# RF_VIDEO_TRACKER

VIDEO TRACKING SOFTWARE LIBRARY
(Programmer manual)

Software library version: 2.3
Software library release date: 19.09.2017
Document version: 2.3
Document release date: 21.09.2017

**www.rmtvision.com**

**ОГЛАВЛЕНИЕ**

## DOCUMENT VERSIONS

Table 1 – Document versions.

| Version | Description |
|---|---|
| 1.9 | Description of the software library RF_VIDEO_TRACKER_CA version 1.9. |
| 1.10 | Description of the software library RF_VIDEO_TRACKER version 1.10. |
| 1.11 | Description of the software library RF_VIDEO_TRACKER version 1.11. |
| 2.0 | Description of the software library RF_VIDEO_TRACKER version 2.1. |
| 2.1 | Description of the software library RF_VIDEO_TRACKER version 2.2. |
| 2.2 | Description of the software library RF_VIDEO_TRACKER version 2.2.1. |
| 2.3 | Description of the software library RF_VIDEO_TRACKER version 2.3. |

## SOFTWARE LIBRARY VERSIONS

Table 2 – Software library versions.

| Version | Description |
|---|---|
| 1.0 | Modified correlation tracking algorithm is implemented. |
| 1.1 | Tracking failure algorithm is implemented. |
| 1.2 | Computation of object position with a sub-pixel accuracy is implemented. |
| 1.3 | Optimized performance for execution by single-thread computing. |
| 1.4 | Masking of background elements is implemented. Influence of background components of the reference image on tracking effectiveness is reduced. |
| 1.5 | Multithreaded computing is implemented for better performance. |
| 1.6 | Algorithm is adapted for 64bit operating systems. |
| 1.7 | Optimized CPU usage in computing. |
| 1.8 | Enhanced performance. |
| 1.9 | Enhanced performance. |
| 1.10 | Simplified interface. |
| 1.11 | Alternative tracking algorithm is added that uses additional features. Methods are used of obtaining the intermediary results. |
| 2.1 | The software library is rewritten in C language (C99 standard) using OpenMP code parallelization standard (standard 1.0). Optimized performance. |
| 2.2 | Optimized structure and performance. Reduced amount of statically allocated memory. |
| 2.2.1 | Automatic adjustment of the position and size of the tracking rectangle is added. The algorithm is changed of the detection of the object bounding box in the tracking rectangle to improve stability of results with changing dimensions of the tracking rectangle. The capability is added of organizing the STOP-FRAME mode and delay compensation in transmission of object capture commands for tracking via communication channels. |
| 2.3 | Enhanced performance. Improved tracking stability for small sizes of the tracking rectangle (strobe). Reduced amount of statically allocated memory. |

## WHAT’S NEW

- Developed initially for class **C++** video. The **C** language version is available on request;
- The amount of statically allocated memory for the class object is reduced to 128 Kbyte;
- Calculation speed is increased by 5% as compared with the version 2.2.1;
- Improved tracking stability for small objects;
- Capability is added of using various video sources for one class instance.

## DESCRIPTION

The **RF_VIDEO_TRACKER** software library implements the algorithm of automatic tracking of objects in video stream and computation of their parameters. In addition, with this software library it is possible to implement the STOP-FRAME mode when capturing objects for tracking and compensate for time delays in communication channels when sending commands to capture objects for tracking. The source code for the software library is written in C ++. Its C language version is available on request. The software library is supplied as source code files containing the description of the RfVideoTracker software class of the automatic tracking algorithm. One copy of the software class allows implementation of one tracking channel. In this case, you can alternately use different video sources for one instance of the tracking software class. To build multi-channel tracking systems, you need to create several instances of the class. The software library includes the following files: **RfDataStructures.h** (description of data structures and constants), **RfVideoTracker.h** (description of the tracking algorithm software class) and **RfVideoTracker.cpp** (or ***.c**, on request) (tracking algorithm implementation file). To use the RF_VIDEO_TRACKER software library in his projects, the developer should include the above source code files in the project.

## BASIC FEATURES AND CAPABILITIES

Table 3 summarizes the main characteristics of the software library.

Table 3 – Main features of the software library and tracking algorithm.

| Parameter | Value and notes |
|---|---|
| Software library language | C++ (C++98 standard) using OpenMP code parallelization standard (standard 1.0) with no use of third-party software libraries and OS-dependent functions. The C language version (C99 standard) is available on request. |
| Compatibility with operating systems | Compatible with all 32 and 64bit operating systems for C++ projects. It is also compatible with real-time systems for signal processors. |
| Compatibility with compilers | Compatible with any compilers that support C++98 and OpenMP 1.0 standards. |
| Statically allocated memory required | No more than 128 Kbyte of statically allocated memory in RAM for one tracking channel. When STOP-FRAME function is used the amount of allocated memory can be significantly larger. |
| Number of tracking channels | The software class implements 1 tracking channels. The library allows implementation of multi-channel systems by creating several objects of the tracking algorithm class. |
| Maximum dimensions of tracked object | Maximum dimensions of the tracked object are 128x128 pixels. Tracking is carried out in the search area where the object image is located. |
| Minimum dimensions of tracked object | 8x8 pixels for minimum dimensions of the tracking rectangle of 16x16 pixels. The algorithm is adapted for tracking small-sized low-contrast objects in the presence of occluders. It is recommended to set the tracking rectangle dimensions larger than the dimensions of the tracked object image. |
| Maximum dimensions of tracking rectangle | 128x128 pixels. Any tracking rectangle aspect ratio is possible within minimum and maximum allowed values. |
| Minimum dimensions of tracking rectangle | 16x16 pixels. |
| Maximum allowed translation of tracked object at one frame | Allowed translation of the object at one frame without tracking collapse is 52 pixels in any direction (horizontal and(or) vertical). |
| Discreteness of computation of the object coordinates | No less than 1/16 pixel. The algorithm calculates the object position at every frame on the video. The computed object coordinates are presented in two forms: integer (1 pixel accuracy) and float (floating point value and discreteness no less than 1/16 pixel). |
| Automatic object's size estimation | The algorithm automatically estimates the position and size of the object within the tracking rectangle for subsequent control of its dimensions. The sizing error for high-contrast object against a uniform background is no more than 16 pixels both horizontally and vertically. The object size estimate with the above accuracy is available after no less than 50 frames of video following the size change or capture for tracking. |
| Automatic computing of the object translation speed | The algorithm automatically calculates the object motion in the video frames with an accuracy of no less than 1/16 pixel/frame. Speed calculation is performed within a maximum of 128 video frames from the beginning of the motion or change of direction. Until 128 video frames elapse, the error is possible that is more than 1/16 pixel/frame. |

| Parameter | Value and notes |
|---|---|
| Changing tracking algorithm parameters | The software library allows changing of the algorithm parameters, including in times of automatic tracking. |
| Format of the input video data | The library accepts video frames in **mono_8** format (1 byte per pixel in grayscale). |
| Input video frame dimensions | Dimensions of video frames allowable for the processing range from 240x240 (width and height, respectively) to 2048x2048 pixels. |
| Computation speed | Calculations are made for each video frame independently of the previous frame after the relevant method of the algorithm software class is called. The software library does not contain any time functions and does not control the period of calling of processing functions. No requirement is imposed on the duration (time interval between frames). The processing time of one video frame does not depend on its size (it must lie within the minimum and maximum limits), it rather depends on the specific features and parameters of the C ++ (C) project which makes use of the library. The performance also depends on the computing platform where calculations are performed. To evaluate the library performance on the user's platform, a demonstration program is supplied. |
| Automatic detection of tracking failure | The algorithm automatically evaluates the quality of tracking and decides about its collapse (loss of object). If the object is lost, the algorithm continues tracking based on the object motion parameters calculated before the loss (its horizontal and vertical velocity components). When the object is detected again, it is automatically recaptured for tracking. If detection of the object does not occur within 256 video frames (this value can be changed by the developer), an automatic tracking is reset. |
| Automatic adjustment of position and dimensions of the tracking rectangle | The capability is implemented of the adjustment of the position and dimensions of the tracking rectangle on command. When the command arrives (calling the corresponding method of the software class), the algorithm analyzes the position of the object in the tracking rectangle, moves the tracking rectangle so that the object is in its center and sets its optimal (in terms of the algorithm) dimensions. **ATTENTION:** it is recommended to use this feature as needed. Calling the adjustment method for each processed frame in a complex background environment can lead to a non-optimal (in terms of the algorithm) setting of the size of the tracking rectangle, which in turn can lead to increased probability of tracking failure. |
| Reduced probability of automatic capture of similar objects near the tracked object | The algorithm evaluates the trajectory of the object and minimizes the probability of capture of similar objects near the tracked object. The algorithm selects several similar objects and analyzes their positions relative to the extrapolated position of the object. If any of the selected objects comes closer to the extrapolated position and has more similarity than the others, this object is taken as the tracked one provided that some additional criteria are met. |
| Shape and configuration of tracked objects | The algorithm enables tracking of any types and shapes of objects. The tracking algorithm implemented in the library does not recognize and identify objects, but rather performs monitoring of any specified objects including monitoring over parts of their image. |
| Allowed rate of change of the shape and size of the tracked object | The allowed change of the object area does not exceed 1% over 1 video frame. The allowed change of the object shape (the ratio of maximum width and height) is not more than 1% per 1 video frame. |
| Allowed rate of change of the mean brightness of the tracked object | The allowed change of the mean brightness of the object or the tracking area near it bounded by the tracking rectangle is 1% over 1 per frame. |
| Maximum partial overlap of the object by an occluder (obstacle) | The allowed partial overlap of the tracked object by an occluder (obstacle) is not more than 50% of its area over not more than 40 video frames. In this case there shall be no tracking collapse. |

| Parameter | Value and notes |
|---|---|
| Object search area | In each video frame, the object is searched in an area sizing up to (**105 pixels + width of the tracking rectangle**) **x** (**105 pixels + height of the tracking rectangle**) whose center coincides with the position of the tracking rectangle center in the frame processed previously. In this case, it is possible to shift the object search area to the required position. Also, the developer may change these values. |
| Type of the implemented tracking algorithm | Modified correlation tracking algorithm with filtering of background components of the frame. |
| STOP-FRAME mode and delay compensation in communication channels | The software library allows compensation of the time delays occurring in the communication channels in the transmission of control commands to the tracking device. Also, the software library allows the implementation of the STOP-FRAME mode to assist the operator in capturing dynamic objects for tracking. |

**NOTE:** constants changing of the tracking algorithm by the developer may result in changing of some characteristics of the algorithm as well as the amount of allocated memory.

The basic functions performed by the RF_VIDEO_TRACKER software library are as follows:

1. capture of an object in a video frame for tracking by the defined capture rectangle;
2. position computation of the captured object in subsequent video frames;
3. computation of the object motion speed in video frames;
4. computation of the object position and dimensions inside the tracking rectangle;
5. changing the algorithm parameters by a command, including in times of tracking;
6. resetting of tracking by a command or automatically when automatic reset criteria are met.

Quality of the automatic tracking depends on the conditions of observation and parameters of the tracked object (its shape and contrast relative to background, and others). To assess the quality of tracking in a variety of situations, a demonstration program is provided.

## OPERATION PRINCIPLE OF THE SOFTWARE LIBRARY

### Operation principle of the video tracking algorithm

The working principle of the algorithm is based on the correlation search method (comparing fragments of a video frame with the reference image of an object formed at the time of capture for tracking and updated in the process of tracking). At the time of capture of the object for tracking, the rectangular area of the video frame specified in the capture settings (position and dimensions) is taken as a reference image of the object. In subsequent frames, the object is searched in the area bounded by the algorithm parameters (rectangular area) whose center coincides with the center of the tracking rectangle calculated in the previous video frame (or with the center of the capture rectangle if the first frame after the capture is processed). The most likely computed location of the object (calculated center of the tracking rectangle) is taken as the coordinates of the tracked object in the current video frame. Figure 1 shows the principle of searching the object in video frames.

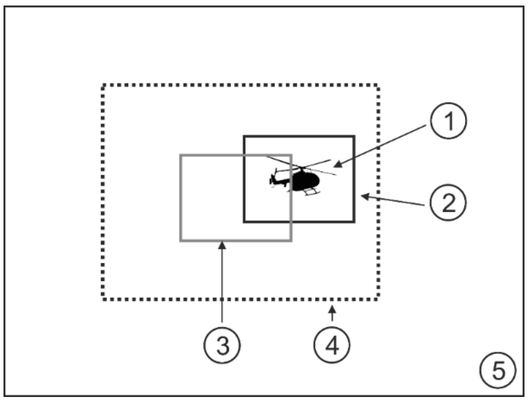

**Figure 1** – Principle of object searching in video frames.
(1 – object image in a frame, 2 – tracking rectangle computed for the current frame, 3 – tracking rectangle position in the previous frame, 4 –search area of the object in the current frame relative to the position of the tracking rectangle in the previous frame, 5 – current frame)

Once the object is captured, the tracking algorithm does not make distinctions between pixels belonging to background or object within the tracking rectangle. Over time (as a certain number of frames are processed), the algorithm evaluates whether a pixel inside the rectangle belongs to the object or background. Based on this information, the tracking quality is further increased and the algorithm evaluates the size and position of the object (its image) within the tracking rectangle which enables subsequent adjustment of the algorithm parameters.

Motion parameters of the object (horizontal and vertical velocity components) are computed for each processed video frame by using a filter whose parameters can set by developer. For each processed frame, the algorithm returns the position of the tracking rectangle center, the position and size of the object rectangle (the rectangle specifies the object dimensions) in the tracking rectangle, as well as the velocity components of the tracked object in the video frames. Search of the object in each video frame passed to the processor is carried out in all possible object positions inside the search area. The computation process results in the formation of the surface of spatial distribution of the likelihood of the object's location of the tracked object in the search area on the current frame. When said surface is formed it is analysed to determine the most probable position of the tracked object in the processed frame. The analysis considers not only the estimated probabilities but also the motion trajectory of the tracked object. **NOTE:** all coordinate and translation values given in this document should be considered in relation to the window coordinate system (with the origin in the upper left corner of the frame).

## STOP-FRAME mode destination

The effectiveness of automatic tracking (accuracy, stability, etc.) depends on the algorithm implemented in the system. In this case, the effectiveness of the tracking system (in the absence of any algorithms for automatic detection and capture) largely depends on the operator's ability to capture an object for tracking. A typical scenario of the operator's actions is the search and detection on the of the workstation monitors of target objects and their capture by "lassoing" the capture strobe on their image. With a fixed optical system and slowly moving (flying, floating) objects, the requirements for the operator's capture skills may be low, but under conditions of a moving camera (rotary platform, target load of unmanned aerial vehicle, etc.) and a complex target situation (fast moving objects), the requirements to the skills of the operator increase many times. The absence of effective algorithms to help the operator (as a part of the system that takes key decisions) in capturing objects for tracking significantly reduces the probability that the system will successfully accomplish the assigned tasks in a complex background-and-target environment. If the operator can't capture the necessary object for tracking, the system will not be able to perform actions required for "handling" the object (target drop-out will occur). An effective capture of a moving object for tracking by operator with any skill level can only be ensured on a still frame (freeze frame). The library provides the capability of object capture for tracking on a still frame by giving the operator some time (a few seconds) to effectively position the capture rectangle over the image of the object. In this case, each video frame is processed without any time "gaps" in the results.

## Purpose of the time delay compensation function in command transmission via data links

Figure 2 illustrates the process of error occurrence in the presence of delays in communication channels.

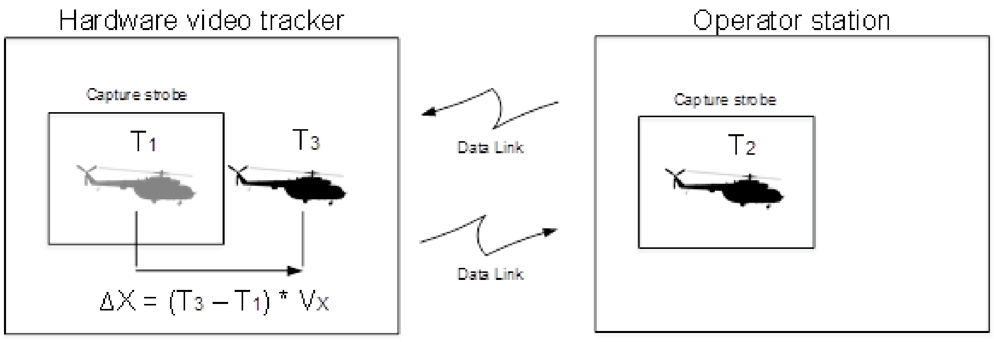

**Figure 2** – Illustration for time delay compensation function.

With remote control of the automatic tracking system, there may be delays in the communication channels during the transmission of commands for capturing the object for tracking, which in turn lead to incorrect capturing and target drop-out. The following situation is considered: at the point of time **T1**, the tracking device receives the next video frame containing the target object. After that, the tracking system transmits the frame data via communication channels to the operator's workstation. The received frame is displayed for the operator at time **T2**. Let us assume that at this point of time the operator captures the object for tracking ("lassoes" it with the capture rectangle). At this moment, a command is created that contains the position and dimensions of the capture rectangle. The generated command is transmitted via communication channels to the tracking device and arrives therein at time **T3**. So, in the time interval from **T1** to **T3**, the object has moved on the video frame whereas the generated command contains capture parameters that are relevant for the time point **T1**.

For large time delays and a moving target, capture of background may occur instead of the object capture, which will result in target missing. Under such conditions, the only possible way to ensure correct capture of the object for tracking is perform the comparison of the capture commands with some unique frame identifier (ID). In this case, the tracking system should capture the object on that very frame where the command was formed on the operator's side (in accordance with the frame ID). The library provides the capability of organizing such functionality.

## Operation algorithm of the STOP-FRAME mode and time delay compensation

The operation algorithm of the STOP-FRAME mode and of the time delay compensation is the same. The principle of the algorithm is as follows: the software library contains a sufficiently large cyclic frame buffer (by default the buffer size holds 256 frames) into which the incoming video frames are sequentially recorded. When free cells (numbers) of the frame buffer run out, the software library starts recording from the beginning of the buffer (it overwrites the earliest frames of the video). The position (index) of the frame in the buffer is the frame identifier. When the next frame is sent to the consumer, it is assigned an ID (frame index in the buffer). This ID is transmitted with the frame data. When an object capture command is formed, the capture parameters should be transmitted along with the ID of the frame on which the object was captured for tracking. The software library carries out capture of the object for tracking on the frame specified in the command. In subsequent processing of the frames for a period corresponding to one video frame, the library processes several frames. Thus, in a short while, the tracking system will be able to "catch up” with the real time" and proceed to the processing of the current video frames. In compensating for the time delays, the principle of operation is the same as in the case of the STOP-FRAME mode: binding the object capture command for tracking to the frame ID and subsequent processing of several frames for each new incoming frame. **ATTENTION:** when using tracking systems using pivoting devices it is recommended to prohibit the rotation control until the tracking algorithm "catches up” with the real time. Otherwise, an attempt to control the executive devices based on the results obtained by processing of the previous frames may lead to incorrect operation of the system.

## Principles of use of the software library

To use the software library the developer should include the library files (**RfDataStructures.h**, **RfVideoTracker.h** and **RfVideoTracker.cpp)** into the project to be designed. Then, the developer should create an object of the class **RfVideoTracker**, declared in file **RfVideoTracker.h**. A typical sequence of use of the software library without implementation of the STOP-FRAME mode or delay compensation in the communication channels of the project can be as follows:

1. create an object of the class RfVideoTracker;
2. set racking algorithm parameters by calling the respective class method if necessary;
3. to add an object for tracking call the Capture(…) method by passing the frame data and capture parameters to it;
4. perform computation of tracking for subsequent video frames through calling the Execute (…) method by passing the frame data to it;
5. after object tracking is completed call the Reset(...) method which will reset the tracking algorithm to the initial state.

A typical sequence of use of the software library with the implementation of the STOP-FRAME mode or time delay compensation can be as follows:

1. create an object of the class RfVideoTracker;
2. before calling any other methods from the software library, call the method Timelapse_Init(…) and pass dimensions of the video frames to be processed as the parameters. In this case, the library will dynamically allocate memory for frame buffer;

3. add each incoming video frame to the frame buffer of the library through calling the Timelapse_AddFrameToBuf(…) method;
4. for each incoming video frame, call the processing Timelapse_Execut(…) method by passing the maximum value of frames to be processed to it;
5. when the object is captured for tracking, call the Timelapse_Capture(…) Method by passing there the ID of frame to be captured.

In the scenario described above, the incoming video frames are always added to the cyclic frame buffer, and the processing method is called for each frame. In this case, the value of maximum number of frames to be processed needs to be passed to the processing method. The processing methods return the filled-in tracking data structures and algorithm parameters. The developer must ensure the integrity of the transmitted data during the execution of the called methods. The software library does not perform any background tasks. All calculations and actions are only performed by calling the appropriate methods and ended by the return of control to the calling thread. **ATTENTION:** when using the STOP-FRAME mode and time delay compensation, the developer should only call the Timelapse_Execut(...), Timelapse_Capture(...), and Timelapse_AddFrameToBuf(...) methods for processing frames.

## OPERATION MODES

### Description of the tracking algorithm operation modes

The automatic tracking algorithm implemented in the RF_VIDEO_TRACKER software library can operate in several different modes (states). Each of the operation modes implemented has own conditions of switching into and out of the mode. Table 4 lists the operating modes of the tracking algorithm for each of the channels.

Table 4 – Operation modes of the automatic tracking algorithm.

| Mode name | Description and switching condition |
|---|---|
| **FREE** | Free mode. No computation is executed in this mode. This is a default mode of the algorithm. Switching to this mode can be effected from any other mode through the reset Reset() method or automatically when the tracking reset criteria are met. |
| **TRACKING** | Automatic tracking mode. In this mode, automatic tracking is computed and all computed parameters of the object are updated. Switching to this mode is effected by a command (through a call of the Execute() method) from the **FREE** mode and can also be performed automatically from the **LOST** mode if the criteria for automatic re-capture for tracking are met. When the reset criteria are fulfilled, automatic changeover to the **FREE** mode occurs. |
| **LOST** | Object loss mode. In this mode, the tracked object coordinates are updated based on the object motion parameters computed before switching to this mode (object's horizontal and vertical velocity components). Changeover to this mode occurs automatically from the **TRACKING** mode when the object loss criteria are met. Switching to this mode is also possible by a command through a call of the Lost() method from the **INERTIAL** mode. In this mode, the search of the object occurs on the video frames and when the criteria of re-capture for tracking are met the tracker automatically jumps to the **TRACKING** mode. When the reset criteria are fulfilled, the tracker automatically switches to the **FREE** mode. |
| **INERTIAL** | Inertial tracking mode. In this mode, the tracked object coordinates are updated based on the object motion parameters calculated before changing to this mode (object's horizontal and vertical velocity components on the video frames). No search for the object is performed in this mode. Switching to this mode is only possible by a command through a call of the Inertial(...) method from the **TRACKING** and **LOST** modes. When the reset criteria are fulfilled, automatic resetting and changeover to the **FREE** mode occur. |
| **STATIC** | Static mode. In this mode, no operations are performed whereas all parameters of the tracked object computed prior to switching to this mode are retained (coordinates, parameters of the tracking rectangle, etc.). The difference from the **INERTIAL** mode is that the tracking rectangle coordinates are not changed. Switching to this mode is only possible by a command through a call of the Static () method. Switching from this mode |

| Mode name | Description and switching condition |
|---|---|
| | can be performed to the **LOST** mode by calling the Lost() method or to the **FREE** mode by calling the Reset() method. |

Figure 3 shows an operating mode switching diagram for the tracking algorithm (tracking channel) and the class methods designed for control of the operating modes. The commands available for the designer include adding the object for tracking (Capture(...)), changing to the inertial tracking mode (Inertial(...)) and to the object loss mode (Lost(...)) from the INERTIAL mode, changing to the STATIC mode (Static(...)) as well as reset of tracking (Reset(...)) from any mode. Also possible is an automatic mode switching when the algorithm parameters are changed, if new parameters of the algorithm cause fulfillment of the reset criteria.

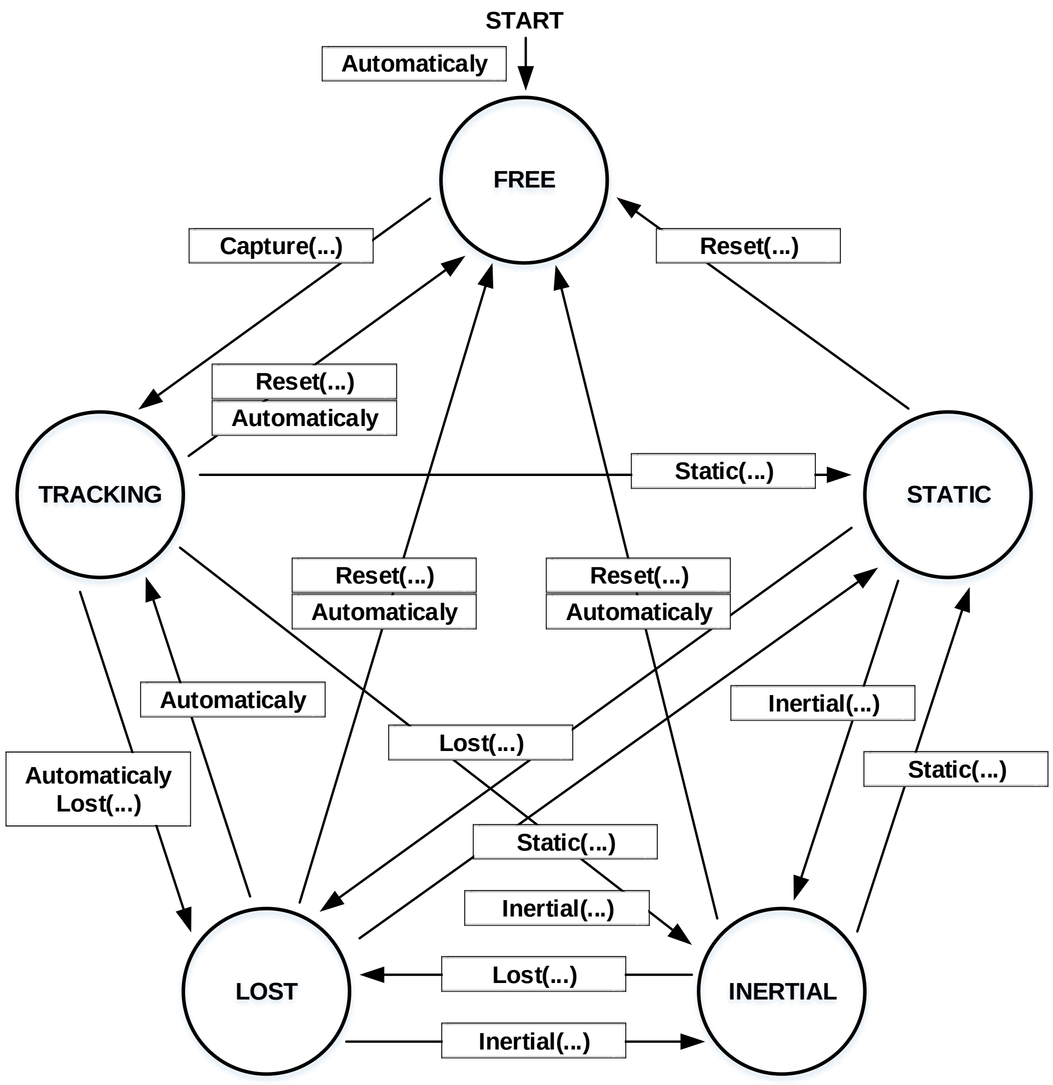

**Figure 3** – Diagram of switching between operation modes.

## Criteria for automatic changing of the algorithm operation modes

As can be seen in the diagram (Figure 3), mode changing occurs both by a command (through a call of the relevant class method) and automatically. Automatic switching of the modes may occur in cases where the following criteria are fulfilled: object loss criteria, criteria of automatic object re-capture for tracking, and reset criteria.

Automatic changeover from the TRACKING mode to the LOST mode takes place in cases where the greatest computed probability of the tracked object's location in the search area of the current processed frame is less than the algorithm adaptive threshold. In this case, the algorithm decides that the object is lost since the likelihood of the object's image location in some of the positions in the search area is too low (below the threshold).

Automatic changeover from the LOST mode to the TRACKING mode occurs in cases where the greatest computed probability of the tracked object's location in the search area of the current processed frame is higher than the algorithm adaptive threshold.

Automatic switching to the FREE mode from the LOST mode can be found in two cases: when the LOST mode persists for more than 256 video frames (the value can be changed by the developer) or when one of the sides of the tracking rectangle comes close to the image bounds at a space of 2 pixels.

Automatic switching to the FREE mode from the other modes occurs if only one of the sides of the tracking rectangle comes close to the image bounds at a space of 2 pixels.

## TRACKING ALGORITHM CONSTANTS

The algorithm constants are declared in the file **RfDataStructures.h** and can be changed by the developer if necessary. Table 5 lists the constants of the tracking algorithm that define the permissible limits of its parameters.

Table 5 – Constants of the tracking algorithm.

| Constant name and description | Define value |
|---|---|
| **RF_MAX_STROBE_W** – maximum width of the tracking rectangle. When the software library is initialized, the width of the tracking rectangle is set to this value. This parameter determines the amount of memory statically allocated by the software library. When this value is increased more memory will be allocated and calculation speed will be lowered. | 128 pixels |
| **RF_MAX_STROBE_H** – maximum height of the tracking rectangle. When the software library is initialized, the height of the tracking rectangle is set to this value. This parameter determines the amount of memory statically allocated by the software library. When this value is increased more memory will be allocated and calculation speed will be lowered. | 128 pixels |
| **RF_MIN_STROBE_W** – minimum width of the tracking rectangle. The width of the tracking rectangle during the work of the application cannot be set lower than this value. | 16 pixels |
| **RF_MIN_STROBE_H** – minimum height of the tracking rectangle. The height of the tracking rectangle during the work of the application cannot be set lower than this value. | 16 pixels |
| **RF_MAX_FRAME_W** – maximum width of the processed video frames. The value of dimensions of the processed frames transmitted as parameters of the methods cannot exceed the specified value. | 2048 pixels |
| **RF_MAX_FRAME_H** – maximum height of the processed video frames. The value of dimensions of the processed frames transmitted as parameters of the methods cannot exceed the specified value. | 2048 pixels |
| **RF_MIN_FRAME_W** – minimum width of the processed video frames. The value of dimensions of the processed frames transmitted as parameters of the methods cannot exceed the specified value. | 240 pixels |
| **RF_MIN_FRAME_H** – minimum height of the processed video frames. The value of dimensions of the processed frames transmitted as parameters of the methods cannot exceed the specified value. | 240 pixels |
| **RF_MAX_CORR_W** – maximum number of possible positions of the object along horizontal checked in the current video frame. This parameter determines the maximum possible translation for this algorithm of the tracked object over one frame along horizontal as (**RF_MAX_CORR_W / 2**). | 105 pixels |
| **RF_MAX_CORR_H** – maximum number of possible positions of the object along vertical checked in the current video frame. This parameter determines the maximum possible translation for this algorithm of the tracked object over one frame along vertical as (**RF_MAX_CORR_H / 2**). | 105 pixels |
| **RF_MIN_CORR_W** – minimum number of possible positions of the object along horizontal checked in the current video frame. This parameter determines the lower boundary of the parameter which can be set by the user during the work of the application. | 27 pixels |

| Constant name and description | Define value |
|---|---|
| **RF_MIN_CORR_H** – minimum number of possible positions of the object along vertical checked in the current video frame. This parameter determines the lower boundary of the parameter which can be set by the user during the work of the application. | 27 pixels |
| **RF_VELX_COEFF** – value of the exponential filter smoothing coefficient for the object's horizontal velocity component. The parameter has the following meaning: the value 0 indicates that the current calculated translation in the frame will be regarded as the current calculated speed of the object on the video frames, i.e., the calculated speed will not be subjected to any smoothing. In turn, the parameter value set at 1 will exclude speed calculation. The current value of the horizontal velocity component of the object's speed is calculated according to the following expression:<br><br>$velX_t = RF\_VELX\_COEFF * velX_{t-1} + (1 - RF\_VELX\_COEFF) * (X_t - X_{t-1})$,<br><br>where: $velX_t$ – the current calculated horizontal component of the object speed; $velX_{t-1}$ – the calculated horizontal component of the object speed on the previous video frame; $X_t$ – the current calculated horizontal coordinate of the center of the tracking rectangle on the video frame; $X_{t-1}$ – the calculated horizontal coordinate of the tracking rectangle center on the previous video frame. | 0.95 |
| **RF_VELY_COEFF** – value of the filter smoothing coefficient for the object's horizontal velocity component. The parameter has the following meaning: the value 0 indicates that the current calculated translation in the frame will be regarded as the current calculated speed of the object on the video frames, i.e., the calculated speed will not be subjected to any smoothing. In turn, the parameter value set at 1 will exclude speed calculation. The current value of the vertical velocity component of the object's speed is calculated according to the following expression:<br><br>$velY_t = RF\_VELY\_COEFF * velY_{t-1} + (1 - RF\_VELX\_COEFF) * (Y_t - Y_{t-1})$,<br><br>where: $velY_t$ – the current calculated vertical component of the object speed; $velY_{t-1}$ – the calculated vertical component of the object speed on the previous video frame; $Y_t$ – the current calculated vertical coordinate of the tracking rectangle center on the video frame; $Y_{t-1}$ – the calculated horizontal coordinate of the center of the tracking rectangle on the previous video frame. | 0.95 |
| **RF_PAT_COEFF** – value of the filter coefficient for smoothing the values of each pixel of the reference image. When the object is captured for tracking, a reference image of the object is formed. In accordance with this reference objects are searched for on subsequent video frames. Following the calculation of the object's position on the current video frame, the reference image of the object is updated. The parameter has the following meaning: the value 1 means that the first image of the object will be accepted as the reference at the time of object capture for tracking and in the future this reference will not be updated. In turn, the value set at 0 means that the object's image on the previous video frame will be used as the reference image. The current value of each of the pixels of the reference image is updated for each processed frame according to the following expression:<br><br>${Pat_{(i,j)}}^t = RF\_PAT\_COEFF * {Pat_{(i,j)}}^{(t-1)} + (1 - RF\_PAT\_COEFF) * {Object_{(i,j)}}^t$,<br><br>where: ${Pat_{(i,j)}}^t$ – the calculated value of the pixel of the reference image of the object with coordinates $(i,j), 0 \le i < RF\_MAX\_STROBE\_H, 0 \le j < RF\_MAX\_STROBE\_W$; ${Pat_{(i,j)}}^{(t-1)}$ – the value of the pixel of the reference image of the object with the coordinates $(i,j)$ on the previous video frame; ${Object_{(i,j)}}^t$ – the value of pixel of the current image of the tracked object with coordinates $(i,j)$. | 0.95 |
| **RF_MAX_ABS_DELTA** – the maximum absolute displacement (module) of the tracking rectangle by command on the video frame horizontally and vertically. The | 64 pixels |

| Constant name and description | Define value |
|---|---|
| displacement of the tracking rectangle is used to adjust its position in the TRACKING mode. | |
| **RF_NUM_THREADS** – the number of threads into which the computations will be divided. The developer sets the parameter value depending on the hardware platform used. It is recommended that this parameter be set equal to the number of computing cores in the system (physical or logical). | 8 computation threads |
| **RF_MAX_ABS_DELTA_SEARCH_WINDOW** – the maximum possible displacement of the search area in any direction. By default, the search for the tracked object on the video frame is performed in the area whose center coincides with the calculated position of the object on the previous video frame. In this case, the user can forcibly shift the position of the search area by a certain value for one processed frame or permanently. In this case, the absolute value of the displacement (module) horizontally and vertically should not exceed this parameter. | 320 pixels |
| **RF_LOST_NUM_FRAMES_LIMIT** – the maximum value of the processed frame counter in the LOST mode which, when reached, leads to the automatic switching to the FREE mode. | 256 frames |
| **RF_RESET_BORDER** – the minimum value of the offset of any of the edges of the tracking rectangle from the frame edge below which automatic reset of tracking occurs (switching to the FREE mode). | 2 pixels |
| **RF_FRAME_BUF_SIZE** – the size of the bounded frame buffer. Each cell in the frame buffer contains the frame data in accordance with the frame dimensions sent to the software library. The size of the bounded buffer determines the time interval available for the operator to capture the object for tracking in the STOP-FRAME mode, and determines the length of time delays in the communication channels that can be compensated. The available time interval is calculated as the frame buffer size multiplied by the video frame scan period. | 256 frames |
| **FREE_MODE_INDEX** – FREE mode identifier. | 0 |
| **TRACKING_MODE_INDEX** – TRACKING mode identifier. | 1 |
| **LOST_MODE_INDEX** – LOST mode identifier. | 2 |
| **INERTIAL_MODE_INDEX** – INERTIAL mode identifier. | 3 |
| **STATIC_MODE_INDEX** – STATIC mode identifier. | 4 |

## DATA STRUCTURES USED IN THE SOFTWARE LIBRARY

### Review of data structures

The data structures used in the software library are described in the header file **RfDataStructures.h**. The class methods receive the parameters and return filled-in data structures. The software library contains a description of only one structure **RF_CHANNEL_DATA** for storage of the results of work of the algorithm. Below is the description of the data structure declaration.

```
typedef struct {
    int strobe_x;
    int strobe_y;
    int strobe_w;
    int strobe_h;
    int substrobe_x;
    int substrobe_y;
    int substrobe_w;
    int substrobe_h;
    int search_wind_dx;
    int search_wind_dy;
    int lost_frame_count;
    int frame_count;
    int corr_w;
    int corr_h;
    float f_strobe_x;
    float f_strobe_y;
    float vel_x;
    float vel_y;
    unsigned char corr_p;
    unsigned char mode;
    unsigned char permanent_search_wind_dxy;
} RF_CHANNEL_DATA;
```

Table 6 describes the fields of the structure RF_CHANNEL_DATA.

Table 6 – Description of RF_CHANNEL_DATA structure fields.

| Field name | Description |
|---|---|
| strobe_x | Horizontal coordinate of the tracking rectangle center. |
| strobe_y | Vertical coordinate of the tracking rectangle center. |
| strobe_w | Width of the tracking rectangle. |
| strobe_h | Height of the tracking rectangle. |
| substrobe_x | Horizontal coordinate of the object bounding box center in the tracking rectangle. In the process of automatic tracking, the software library calculates the position and size estimates of the object in the tracking rectangle to enable subsequent correction. The origin of the coordinate center for the position of the object box center is in the upper left corner of the tracking rectangle. |
| substrobe_y | Vertical coordinate of the object bounding box center in the tracking rectangle. In the process of automatic tracking, the software library calculates the position and size estimates of the object in the tracking rectangle to enable subsequent correction. The origin of the coordinate center for the position of the object box center is in the upper left corner of the tracking rectangle. |
| substrobe_w | Width of the object bounding box in the tracking rectangle. |
| substrobe_h | Height of the object bounding box in the tracking rectangle. |
| search_wind_dx | Horizontal displacement of the object search window in the frame relative to the calculated position of the center of the tracking rectangle. |
| search_wind_dy | Vertical displacement of the object search window in the frame relative to the calculated position of the center of the tracking rectangle. |
| lost_frame_count | Processed frame counter in the LOST mode. |
| frame_count | Processed frame counter starting from the moment of object capture for tracking. |
| corr_w | The number of checked positions of probable location of the object along horizontal. This parameter determines the width of the object search area. The width of the object search area is calculated as (strobe_w + corr_w - 1). |
| corr_h | The number of checked positions of probable location of the object along vertical. This parameter determines the width of the object search area. The height of the object search area is calculated as (strobe_h + corr_h - 1). |
| f_strobe_x | Horizontal coordinate of the center of the tracking rectangle with subpixel accuracy. |
| f_strobe_y | Vertical coordinate of the center of the tracking rectangle with subpixel accuracy. |
| vel_x | Calculated estimate of the object’s horizontal velocity component on video frames in pixels per frame. |
| vel_y | Calculated estimate of the object’s vertical velocity component on video frames in pixels per frame. |
| corr_p | Estimate of the probability of detecting an object on a processed frame. The value ranges from 0 (minimum probability) to 255 (maximum probability). |
| mode | Identifier of the tracking algorithm working mode in accordance with the value of the macros (see CONSTANTS OF THE TRACKING ALGORITHM). |
| permanent_search wind_dxy | Flag of the permanent offset of the search area. When the user sets the search area offset, the permanent offset flag is set (0 – only for one frame, 1 - permanent offset). If the offset is set for one frame, then after the frame has |

| Field name | Description |
|---|---|
| | been processed, the value of the offsets “search_wind_dx” and “search_wind_dy” will be automatically set to 0. |

## DESCRIPTION OF SOFTWARE LIBRARY INTERFACE METHODS

### List of the RfVideoTracker class methods

The software class interface RfVideoTracker is a set of methods available to the user. Table 7 lists the interface methods for the class.

Table 7 – Methods list of the RfVideoTracker class.

| Method name | Description |
|---|---|
| Capture(…) | Method of object capture for tracking. |
| Execut(…) | Method of tracking calculation. |
| Reset() | Method of tracking reset. |
| Lost() | Method of switching of the algorithm to the LOST mode. |
| Inertial() | Method of switching of the algorithm to the NERTIAL mode. |
| Static() | Method of switching of the algorithm to the STATIC mode. |
| SetStrobeSize(…) | Method of setting the dimensions of the tracking rectangle. |
| SetStrobeAutoSize(…) | Method of automatic adjustment of the position and dimensions of the tracking rectangle. |
| MoveSearchWindow(…) | Method of setting the search area offset. |
| SetCorrSize(…) | Method of setting the number of checked positions of the object in video frame. |
| MoveStrobe(…) | Method of displacement of the tracking rectangle. |
| GetChannelData() | Method of obtaining the data structure with current calculation results and algorithm parameters. |
| GetPatImage(…) | Method of obtaining the current reference image of the selected channel. |
| GetMaskImage(…) | Method of obtaining the current image of the mask. |
| GetCorrImage(…) | Method of obtaining the image of the distribution surface of the probability of the tracking object’s location in this or that position of the search area. |
| Timelapse_Init(…) | Method of software library initialization in case of using the STOP-FRAME mode or compensation of time delays. The method allocates memory for the frame buffer and initializes the variables. |
| Timelapse_Execut(…) | Method of tracking calculation in case of using the STOP-FRAME mode or compensation of time delays. Calculation is carried out for several frames at a time. |
| Timelapse_AddFrameToBuf(…) | Method of adding the next frame into the frame buffer in case of using the STOP-FRAME mode or compensation of time delays. |
| Timelapse_Capture(…) | Method of object capture for tracking in case of using the STOP-FRAME mode or compensation of time delays. |

### Capture(…) method

The **Capture(…)** method is intended for indicating an object for the algorithm to monitor on subsequent processed video frames.

*Declaration:*

```
RF_CHANNEL_DATA Capture(
unsigned char *frame,
int frameW,
int frameH,
int strobeX,
int strobeY,
unsigned char *captureMask = 0);
```

*Parameters:*

| | |
|---|---|
| *frame | Pointer to the frame data in mono_8 format. The size of the frame buffer must be equal to frameW * frameH. |
| frameW | Width of the processed video frame. |
| frameH | Height of the processed video frame. |
| strobeX | Horizontal coordinate of the tracking rectangle center. |
| strobeY | Vertical coordinate of the tracking rectangle center. |
| *captureMask = 0 | A pointer to the object mask in the capture rectangle. When capturing an object tracking algorithm does not differentiate between the pixels in the capture rectangle belonging to the background and belonging to the object. Over time, the tracking algorithm constructs hypotheses about the belonging of a particular pixel in the tracking rectangle to the object or background. If at the time of capture, it is known which pixels of the capture rectangle belong to the object and which to the backgrounds it is possible to transmit this information to the tracking algorithm for more stable capture. The mask data should be in the format of one byte per pixel and lie in the range of values from 0 (the pixel most likely belongs to the background) to 255 (the pixel most likely belongs to the object). In this case, the size of the mask (its width multiplied by the height of the capture rectangle) should be equal to the size of the capture rectangle. |

*Return value:*
The method returns the current data structure RF_CHANNEL_DATA after executing the command to capture the object for tracking. The object capture for tracking can’t be performed in the following cases: the frame dimensions specified are larger or smaller than the permissible values determined by the algorithm constants; when capturing is performed, the criteria for automatic reset of tracking are met (see "Criteria for automatic changing of the algorithm operation modes").

## Execute(…) method

The **Execute(…)** method is intended for tracking calculation. The method is called for processing the next video frame.

*Declaration:*

```
RF_CHANNEL_DATA Execute(unsigned char *frame, int frameW, int frameH);
```

*Parameters:*

| | |
|---|---|
| *frame | Pointer to the frame data in mono_8 format. The size of the frame buffer must be equal to frameW * frameH. |
| frameW | Width of the processed video frame. |
| frameH | Height of the processed video frame. |

*Return value:*
The method returns the data structure RF_CHANNEL_DATA after the calculations are completed. When calculations are performed, it is possible to automatically change the operation modes of the selected (see "Criteria for automatic changing of the algorithm operation modes").

## Reset() method

The **Reset()** method is intended for resetting of tracking. The method switches the algorithm to the FREE mode.

*Declaration:*

```
RF_CHANNEL_DATA Reset();
```

*Return value:*
The method returns the data structure RF_CHANNEL_DATA after tracking reset is completed.

## Lost() method

The **Lost()** method is intended for switching the tracking algorithm to the LOST mode from the INERTIAL and STATIC modes.

*Declaration:*

```
RF_CHANNEL_DATA Lost()
```

*Return value:*
The method returns the data structure RF_CHANNEL_DATA after execution of the command.

## Inertial() method

The **Inertial()** method is intended for switching the tracking algorithm to the INERTIAL mode from the LOST, TRACKING and STATIC modes.

*Declaration:*

```
RF_CHANNEL_DATA Inertial();
```

*Return value:*
The method returns the data structure RF_CHANNEL_DATA after execution of the command.

## Static() method

The **Static()** method is intended for switching the tracking algorithm to the STATIC mode from the TRACKING, INERTIAL and STATIC modes.

*Declaration:*

```
RF_CHANNEL_DATA Static();
```

*Return value:*
The method returns the data structure RF_CHANNEL_DATA after execution of the command.

## SetStrobeSize(…) method

The **SetStrobeAutoSize(…)** method is intended for setting the dimensions of the tracking rectangle.

*Declaration:*

```
int SetStrobeSize(int strobeW, int strobeH, int frameW, int frameH);
```

*Parameters:*

| | |
|---|---|
| strobeW | Width of the tracking rectangle for setting. |
| strobeH | Height of the tracking rectangle for setting. |
| frameW | Width of the processed video frames. |

| | |
|---|---|
| frameH | Height of the processed video frames. |

*Return value:*
The method returns **1** if the execution is successful, or **-1** if new values of the tracking rectangle dimensions can’t be set or if the auto-reset criteria is met.

## SetStrobeAutoSize(…) method

The **SetStrobeAutoSize(…)** method is designed to automatically adjust the size of the tracking rectangle. The method moves the tracking rectangle so that the tracked object is in its center. After that, the method changes the dimensions of the tracking rectangle to the optimal ones (from the point of view of the algorithm).

*Declaration:*

```
void SetStrobeAutoSize(int frameW, int frameH);
```

*Parameters:*

| | |
|---|---|
| frameW | Width of the processed video frames. |
| frameH | Height of the processed video frames. |

## MoveSearchWindow(…) method

The **MoveSearchWindow(…)** method is intended for setting the offset of the search area center relative to the calculated position of the object on the previous frame.

*Declaration:*

```
int MoveSearchWindow(int dX, int dY, unsigned char permanent_flag);
```

*Parameters:*

| | |
|---|---|
| dX | Displacement (offset) of the search area along horizontal. |
| dY | Displacement (offset) of the search area along vertical. |
| permanent_flag | Flag of the permanent offset of the search area. If the parameter value exceeds 0, then, on all subsequent video frames the search area displacement will be according to the set values. If the parameter value equals 0, then, the search area displacement to the set values will be performed only for one subsequent processed frame. |

*Return value:*
The method returns **1** if the execution is successfull, and **-1** if the transmitted offset values are larger than the preset limits (see TRACKING ALGORITHM CONSTANTS).

## SetCorrSize(…) method

The **SetCorrSize(…)** method is intended for setting the number of checked positions of the object on the video in times of the search.

*Declaration:*

```
int SetCorrSize(int corrW, int corrH);
```

*Parameters:*

| | |
|---|---|
| corrW | The number of checked positions along horizontal. |
| corrH | The number of checked positions along vertical. |

*Return value:*
The method returns **1** if the execution is successful, and **-1** if the transmitted values of the number of checked positions is more or less than the preset limits (see TRACKING ALGORITHM CONSTANTS).

## MoveStrobe(…) method

The **MoveStrobe(…)** method is intended for displacing the tracking rectangle along horizontal and (or) vertical in the TRACKING mode. The method allows one to correct the position of the tracking rectangle to ensure quality tracking when the object's aspect angle and dimensions are changed.

*Declaration:*

```
int MoveStrobe(int dX, int dY);
```

*Parameers:*

| | |
|---|---|
| dX | Displacement of the tracking rectangle along horizontal. |
| dY | Displacement of the tracking rectangle along vertical. |

*Return value:*
The method returns **1** if the execution is successful, and **-1** if the transmitted values of the offsets (displacements) are larger than the preset limits (see TRACKING ALGORITHM CONSTANTS).

## GetChannelData(…) method

The **GetChannelData(…)** method is intended for obtaining the current data RF_CHANNEL_DATA.

*Declaration:*

```
RF_CHANNEL_DATA GetChannelData();
```

*Return value:*
The method returns the current values of the tracking data.

## GetPatImage(…) method

The **GetPatImage(…)** method is designed to obtain the reference image of the tracking object in the mono_8 format (1 byte per pixel in grayscale). The method fills the transmitted array with the reference image data.

*Declaration:*

```
void GetPatImage(unsigned char *img);
```

*Parameters:*

| | |
|---|---|
| *img | The pointer to the image data buffer to be filled. The buffer size should be equal to (RF_MAX_STROBE_W * RF_MAX_STROBE_H). |

## GetMaskImage(…) method

The **GetMaskImage(…)** method is designed to obtain the image of the filter mask of the tracking object in the mono_8 format (1 byte per pixel in grayscale). The filter mask is used by the algorithm to estimate the size of the tracking object. The method fills the array with filter mask data transmitted by the pointer.

*Declaration:*

```
void GetMaskImage(unsigned char *img);
```

*Parameters:*

| | |
|---|---|
| *img | The pointer to the image data buffer to be filled. The buffer size should be equal to (RF_MAX_STROBE_W * RF_MAX_STROBE_H). |

## GetCorrImage(…) method

The **GetCorrImage(…)** method is used to obtain the image of the probability distribution surface in the mono_8 format (1 byte per pixel in grayscale). The probability distribution surface characterizes the probability of location of the object's image in any of the positions of the search area on the current video frame. The pixel value of the image of the probability distribution surface lies in the range from 0 (minimum probability) to 255 (full similarity).

*Declaration:*

```
void GetCorrImage(unsigned char *img);
```

*Parameters:*

| | |
|---|---|
| *img | The pointer to the image data buffer to be filled. The buffer size should be equal to (RF_MAX_CORR_W * RF_MAX_CORR_H). |

## Timelapse_Init(…) method

The **Timelapse_Init(…)** method is used to initialize the software library when using the STOP-FRAME mode or to compensate for the time delays in communication channels when sending control commands. The main purpose of this class method is the dynamic allocation of memory for the frame buffer. This method must necessarily be called first.

*Declaration:*

```
void Timelapse_Init(int frameW, int frameH)
```

*Parameters:*

| | |
|---|---|
| frameW | Width of the processed video frames. |
| frameH | Height of the processed video frames. |

## Timelapse_AddFrameToBuf(…) method

The **Timelapse_AddFrameToBuf(…)** method is used to add the next frame to the frame buffer when using the STOP-FRAME mode or to compensate for the time delays in communication channels when sending control commands.

*Declaration:*

```
int Timelapse_AddFrameToBuf(unsigned char *frame)
```

*Parameters:*

| | |
|---|---|
| *frame | Pointer to the frame data. The size of the transmitted frame buffer should match the sizes of the processed frames preset by the Timelapse_Init(…) method. |

*Return value:*
The method returns the added frame index in the frame buffer.

## Timelapse_Execut(…) method

The **Timelapse_Execut(…)** method is used for tracking calculation when using the STOP-FRAME mode or for compensation of the time delays in communication channels when sending control commands.

The method performs processing of several frames from the frame buffer, if the current processed video frame is a few frames away from the last one added to it. In this case, the number of simultaneously processed frames can't exceed the preset limit.

*Declaration:*

```
RF_CHANNEL_DATA Timelapse_Execute(int *channelFrameID, int maxNumFrames)
```

*Parameters:*

| | |
|---|---|
| *channelFraemID | Pointer to the Return value of the current processed video frame. |
| maxNumFrames | The value of the maximum number of frames processed by the method. If the index of the current processed video frame is more than one position away from the index of the current frame added to the buffer, then, the method will process several frames until the indices are equal, and the number of processed frames will not exceed the user-defined value. |

*Return value:*
The method returns the tracking data structure RF_CHANNEL_DATA corresponding to the last processed frame in the frame buffer.

To clarify peculiarities of using the algorithm, consider the following situation. The current recorded video frame has the index of 100 in the frame buffer. At the same time, the capture of an object for tracking occurred on a frame with the index 50. Thus, the software library starts tracking the object on the frame which is 50 periods (frames) away from the actual one. When the Execute(...) method is called with the maximum number of frames to be processed equal to 10, the method will process 10 consecutive frames in the buffer at a time and return the result of processing the latter one. Thus, after the first iteration, the index of the last frame processed by the software library becomes 60. After the arrival of the next frame, when the Reset () method is called, it will be written to the buffer cell with the index 101. The subsequent call of the The Execute(...) method will bring the index of the last processed frame closer by another 10 positions (the index will be equal to 70). Hence, for several periods of time, the software library will "catch up" with real time. After the indices of the processed frame and the current frame added to the buffer become equal, the Execute(...) method will process only one frame (the last one added to the buffer). **ATTENTION:** in view of the fact that the method can process several frames at a time, the working time of the method (the time from the call to the return of control) increases proportionally. This may hinder the use of time delay compensation in building some tracking systems due to the lack of execution time stability. If it is necessary to independently control the calculation time, it is recommended to call this method several times with the parameter maxNumFrames equal to 1. In this case, it is necessary to control the time spent on computation.

## Timelapse_Capture(…) method

The **Timelapse_Capture(…)** method is intended for capturing an object for tracking when using the STOP-FRAME mode or compensation of time delays in transmission of control commands via communication channels.

*Declaration:*

```
RF_CHANNEL_DATA Timelapse_Capture(int strobeX, int strobeY, int frameID)
```

*Parameters:*

| | |
|---|---|
| strobeX | Horizontal coordinate of the capture (tracking) rectangle center. |
| strobeY | Vertical coordinate of the capture (tracking) rectangle center. |
| frameID | Identifier of the frame on which capture for tracking must be executed. |

*Return value:*
The method returns the tracking data structure RF_CHANNEL_DATA corresponding to the frame on which capture for tracking was executed.

## EXAMPLE OF USE OF THE SOFTWARE LIBRARY

The example below represents a source code of the console program that opens video file, sends it to display and controls tracking of the object on the first channel. The example was designed for Windows operating systems. If you want to use the example for Linux operating systems, delete the operating system-dependent functions for opening video file and calculating time. An OpenCV computer vision library is used to capture and display video frames and to perform drawing tasks. Computer keyboard and mouse are used for automatic tracking control. The following example uses the STOP-FRAME mode by default.

*Listing:*

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <fstream>
#include <opencv2/opencv.hpp>
#include <Windows.h>
#include <shobjidl.h>
#include <time.h>
#include <RfVideoTracker.h>

// Video capture structure
typedef struct {
    cv::VideoCapture cap;
    cv::Mat frame;
    cv::Mat mono_frame;
    cv::Mat buf_frame;
    int frame_w;
    int frame_h;
    int fps;
} VIDEOCAPTURE;

// Global constants
int mouse_x;
int mouse_y;
VIDEOCAPTURE cap;
rf::RfVideoTracker tracker;
bool record_mode_flag;
bool stop_execut_flag;
cv::VideoWriter *writer;
RF_CHANNEL_DATA channel_data;
LARGE_INTEGER timerFrequency, timerStart, timerStop;
float processingTime;
int video_clip_num;
int current_frame_id;
const int calculatedFramesPerPeriod = 10;

// Functions declaration
bool OpenVideoFileDialog();
void VideoProcessingFunc();
bool KeyboardEventFunc(const int key);
void MouseCallBackFunc(int event, int x, int y, int flags, void* userdata);
void DrawInfo(cv::Mat &frame_img);


// Entry point
int main(int argc, char **argv) {

    std::cout << "####### RF_VIDEO TRACKER v2.3 DEMO APP #######" << std::endl;
    std::cout << "CONTROLL BUTTONS:" << std::endl;
    std::cout << "  Q - set auto size strobe                    " << std::endl;
    std::cout << "  W - increase vertical size of the tracking rectangle" << std::endl;
    std::cout << "  S - reduce vertical size of tracking rectangle" << std::endl;
    std::cout << "  D - increase horizontal size of tracking rectangle" << std::endl;
    std::cout << "  A - reduce horizontal size of tracking rectangle" << std::endl;
    std::cout << "  T - move tracking rectangle up              " << std::endl;
    std::cout << "  G - move tracking rectangle down            " << std::endl;
    std::cout << "  F - move tracking rectangle left            " << std::endl;
    std::cout << "  H - move tracking rectangle right           " << std::endl;
    std::cout << "  P - enable/disable INERTIAL mode           " << std::endl;
    std::cout << "  R - record video                           " << std::endl;
    std::cout << "  SPACE - enable/disable STOP frame mode      " << std::endl;
    std::cout << "  ESC - exit                                 " << std::endl << std::endl;

    // Open video/image file
    if (OpenVideoFileDialog()) {
        VideoProcessingFunc();
    }
    else {
        std::cout << "ERROR! Video file not loaded. Exit..." << std::endl;
        exit(0);
    }//if...

    return 0;

}//int main...

bool OpenVideoFileDialog() {

    IFileOpenDialog *pFileOpen;
    HRESULT hr;
    PWSTR file;
    char* filename;

    hr = CoInitializeEx(NULL, COINIT_APARTMENTTHREADED | COINIT_DISABLE_OLE1DDE);
    if (SUCCEEDED(hr)) {
        hr = CoCreateInstance(CLSID_FileOpenDialog,
            NULL, CLSCTX_ALL,
            IID_IFileOpenDialog,
            reinterpret_cast<void**>(&pFileOpen));
        if (SUCCEEDED(hr)) {
            pFileOpen->SetTitle(L"OPEN VIDEO FILE");
            hr = pFileOpen->Show(NULL);
            if (SUCCEEDED(hr)) {
                IShellItem *pItem;
                hr = pFileOpen->GetResult(&pItem);
                if (SUCCEEDED(hr)) {
                    hr = pItem->GetDisplayName(SIGDN_FILESYSPATH, &file);
                    if (SUCCEEDED(hr)) {
                        pItem->Release();
                        int count = WideCharToMultiByte(CP_ACP, 0, file, static_cast<int>(wcslen(file)), 0, 0, NULL, NULL);
                        filename = new char[count + 1];
                        WideCharToMultiByte(CP_ACP, 0, file, count, filename, count + 1, NULL, NULL);
                        filename[static_cast<size_t>(count)] = '\0';
                        std::cout << filename << std::endl;
                        cap.cap.open(filename);
                        if (!cap.cap.isOpened()) {
                            return false;
                        }
                        else {
                            cap.frame_w = static_cast<int>(cap.cap.get(CV_CAP_PROP_FRAME_WIDTH));
                            cap.frame_h = static_cast<int>(cap.cap.get(CV_CAP_PROP_FRAME_HEIGHT));
                            cap.fps = static_cast<int>(cap.cap.get(CV_CAP_PROP_FPS));
                            cap.cap >> cap.frame;
                            int num_frames = static_cast<int>(cap.cap.get(CV_CAP_PROP_FRAME_COUNT));
                            cap.mono_frame = cv::Mat(cv::Size(cap.frame_w, cap.frame_h), CV_8U);
                            cap.buf_frame = cv::Mat(cv::Size(cap.frame_w, cap.frame_h), cap.frame.channels() == 1 ? CV_8U
: CV_8UC3);
                            return true;
                        }//if...
                    }//if...
                }//if...
                pItem->Release();
            }//if...
        }//if...
    }//if...
    return false;
```

```
}//bool OpenVideoFileDialog...

void VideoProcessingFunc() {

    // Init vars
    cv::Mat display_frame(cap.frame.size(), cap.frame.channels() == 1 ? CV_8U : CV_8UC3);
    cv::Mat pat_img(cv::Size(RF_MAX_STROBE_W, RF_MAX_STROBE_H), CV_8U);
    cv::Mat mask_img(cv::Size(RF_MAX_STROBE_W, RF_MAX_STROBE_H), CV_8U);
    cv::Mat corr_img(cv::Size(RF_MAX_CORR_W, RF_MAX_CORR_H), CV_8U);
    std::string str;
    int video_start_time;
    int video_delta_time;
    int wait_time;
    int frame_period_time;
    int temp_frame_id;

    record_mode_flag = false;
    stop_execut_flag = false;
    mouse_x = 0;
    mouse_y = 0;
    processingTime = 0.002f;
    QueryPerformanceFrequency(&timerFrequency);

    // Init windows
    cv::namedWindow("RF VIDEO TRACKER v2.3 DEMO APP", CV_WINDOW_AUTOSIZE);
    cv::moveWindow("RF VIDEO TRACKER v2.3 DEMO APP", 20, 20);
    cv::setMouseCallback("RF VIDEO TRACKER v2.3 DEMO APP", MouseCallBackFunc, nullptr);
    cv::namedWindow("PATTERN", CV_WINDOW_AUTOSIZE);
    cv::moveWindow("PATTERN", 35 + cap.frame_w, 20);
    cv::namedWindow("MASK", CV_WINDOW_AUTOSIZE);
    cv::moveWindow("MASK", 35 + cap.frame_w, 180);
    cv::namedWindow("CORRELATION SURFACE", CV_WINDOW_AUTOSIZE);
    cv::moveWindow("CORRELATION SURFACE", 35 + cap.frame_w, 340);

    if (cap.fps > 0) frame_period_time = static_cast<int>(1000 / cap.fps);
    else frame_period_time = 33;

    // Init video tracking lib
    tracker.Timelapse_Init(cap.frame_w, cap.frame_h);
    channel_data = tracker.GetChannelData();

    // main loop
    current_frame_id = 0;
    while (true) {

        video_start_time = static_cast<int>(clock());

        // Capture frame
        cap.cap >> cap.frame;
        if (cap.frame.empty()) {
            cap.cap.set(CV_CAP_PROP_POS_FRAMES, 0);
            continue;
        }//if...
        if (stop_execut_flag) cap.buf_frame.copyTo(display_frame);
        else cap.frame.copyTo(display_frame);

        // Convert to grayscale
        if (cap.frame.channels() == 1) memcpy(&cap.mono_frame.data[0], &cap.frame.data[0], cap.frame_w * cap.frame_h);
        else cv::cvtColor(cap.frame, cap.mono_frame, CV_RGB2GRAY);

        // Add frame data to buf
        temp_frame_id = tracker.Timelapse_AddFrameToBuf(cap.mono_frame.data);
        if (!stop_execut_flag) current_frame_id = temp_frame_id;

        // Calculate
        QueryPerformanceCounter(&timerStart);
        channel_data = tracker.Timelapse_Execute(&temp_frame_id, calculatedFramesPerPeriod);
        QueryPerformanceCounter(&timerStop);
        if (channel_data.mode != FREE_MODE_INDEX) {
            processingTime = processingTime * 0.95f + (static_cast<double>(timerStop.QuadPart -
                timerStart.QuadPart) / timerFrequency.QuadPart) * 0.05f;
        }//if...

        // Get additional images
        tracker.GetPatImage(&pat_img.data[0]);
        tracker.GetMaskImage(&mask_img.data[0]);
        tracker.GetCorrImage(&corr_img.data[0]);

        // Draw info
        DrawInfo(display_frame);

        // Display info
        cv::imshow("RF VIDEO TRACKER v2.3 DEMO APP", display_frame);
        cv::imshow("CORRELATION SURFACE", corr_img);
        cv::imshow("PATTERN", pat_img);
        cv::imshow("MASK", mask_img);

        // Record video
        if (record_mode_flag) {
            writer->write(display_frame);
        }//if...

         // Wait keyboard events
        video_delta_time = static_cast<int>(clock()) - video_start_time;
        wait_time = frame_period_time - video_delta_time;
        if (wait_time < 0 || wait_time > frame_period_time) wait_time = frame_period_time;
        if (wait_time == 0) wait_time = 1;
        if (KeyboardEventFunc(cv::waitKey(wait_time))) {
            cv::destroyAllWindows();
            break;
        }//if...

    }//while...

}//void VideoProcessingFunc...

void DrawInfo(cv::Mat &frame_img) {

    if (channel_data.mode != FREE_MODE_INDEX) {
        cv::Scalar strobColor;
        if (channel_data.mode == 1) strobColor = cv::Scalar(0, 0, 255);          // TRACKING
        if (channel_data.mode == 2) strobColor = cv::Scalar(255, 0, 0);          // LOST
        if (channel_data.mode == 3) strobColor = cv::Scalar(0, 255, 0);          // INERTIAL
        if (channel_data.mode == 4) strobColor = cv::Scalar(255, 255, 0);        // STATIC
        cv::rectangle(frame_img, cv::Rect(channel_data.strobe_x - channel_data.strobe_w / 2,
            channel_data.strobe_y - channel_data.strobe_h / 2, channel_data.strobe_w, channel_data.strobe_h), strobColor,
1);
        cv::rectangle(frame_img, cv::Rect(channel_data.strobe_x - (channel_data.strobe_w / 2) + channel_data.substrobe_x
- (channel_data.substrobe_w / 2),
            channel_data.strobe_y - (channel_data.strobe_h / 2) + channel_data.substrobe_y - channel_data.substrobe_h /
2,
            channel_data.substrobe_w, channel_data.substrobe_h), cv::Scalar(255, 255, 255), 1);
        cv::rectangle(frame_img, cv::Rect(channel_data.strobe_x + channel_data.search_wind_dx - (channel_data.strobe_w
+ channel_data.corr_w - 1) / 2,
            channel_data.strobe_y + channel_data.search_wind_dy - (channel_data.strobe_h + channel_data.corr_h - 1) / 2,
            channel_data.strobe_w + channel_data.corr_w - 1, channel_data.strobe_h + channel_data.corr_h - 1),
cv::Scalar(0, 0, 0), 1);
    }
    else {
        cv::rectangle(frame_img, cv::Rect(mouse_x - channel_data.strobe_w / 2, mouse_y - channel_data.strobe_h / 2,
            channel_data.strobe_w, channel_data.strobe_h), cv::Scalar(255, 255, 255), 1);
    }//if...

    std::string str;
    switch (channel_data.mode) {
    case FREE_MODE_INDEX: str += "MODE: FREE"; break;
    case TRACKING_MODE_INDEX: str += "MODE: TRACKING"; break;
    case LOST_MODE_INDEX: str += "MODE: LOST"; break;
    case INERTIAL_MODE_INDEX: str += "MODE: INERTIAL"; break;
    case STATIC_MODE_INDEX: str += "MODE: STATIC"; break;
    default: str = "MODE: UNDEFINED"; break;
    }//switch...
    cv::putText(frame_img, str, cv::Point(10, cap.frame_h - 45), cv::FONT_HERSHEY_SIMPLEX, 0.5, cv::Scalar(255, 255,
0));
    str = "POSITION: " + ((channel_data.mode == FREE_MODE_INDEX) ? std::to_string(mouse_x) :
std::to_string(channel_data.strobe_x)) + "/" +
        ((channel_data.mode == FREE_MODE_INDEX) ? std::to_string(mouse_y) : std::to_string(channel_data.strobe_y)) +
"  STROBE SIZE: " +
        std::to_string(channel_data.strobe_w) + "/" + std::to_string(channel_data.strobe_h);
    cv::putText(frame_img, str, cv::Point(10, cap.frame_h - 30), cv::FONT_HERSHEY_SIMPLEX, 0.5, cv::Scalar(255, 255,
0));
    str = "OBJECT VEL X: " + std::to_string(channel_data.vel_x) + "  VEL Y: " + std::to_string(channel_data.vel_y);
    cv::putText(frame_img, str, cv::Point(10, cap.frame_h - 15), cv::FONT_HERSHEY_SIMPLEX, 0.5, cv::Scalar(255, 255,
0));
    str = "FPS: " + std::to_string(static_cast<int>(1.0f / processingTime));
    cv::putText(frame_img, str, cv::Point(cap.frame_w - 100, 15), cv::FONT_HERSHEY_SIMPLEX, 0.5, cv::Scalar(255, 255,
0));
    if (record_mode_flag) cv::circle(frame_img, cv::Point(15, 15), 7, cv::Scalar(0, 0, 255), CV_FILLED);

}//void DrawInfo...

bool KeyboardEventFunc(const int key) {

    switch (key) {

        // ESC
    case 27:
        std::cout << "EXIT" << std::endl;
        if (record_mode_flag) {
            writer->release();
        }//if...
        return true;
        // "P"
    case 112:
        if (channel_data.mode != INERTIAL_MODE_INDEX) channel_data = tracker.Inertial();
        else channel_data = tracker.Lost();
        break;
        // W
    case 119:
        channel_data.strobe_h += 4;
        tracker.SetStrobeSize(channel_data.strobe_w, channel_data.strobe_h, cap.frame_w, cap.frame_h);
        break;
        // S
    case 115:
        channel_data.strobe_h -= 4;
        tracker.SetStrobeSize(channel_data.strobe_w, channel_data.strobe_h, cap.frame_w, cap.frame_h);
        break;
        // D
    case 100:
        channel_data.strobe_w += 4;
        tracker.SetStrobeSize(channel_data.strobe_w, channel_data.strobe_h, cap.frame_w, cap.frame_h);
        break;
        // A
    case 97:
        channel_data.strobe_w -= 4;
        tracker.SetStrobeSize(channel_data.strobe_w, channel_data.strobe_h, cap.frame_w, cap.frame_h);
        break;

        // T
    case 116:
        tracker.MoveStrobe(0, 4);
        break;
        // G
    case 103:
        tracker.MoveStrobe(0, -4);
        break;
        // H
    case 104:
        tracker.MoveStrobe(-4, 0);
        break;
        // F
    case 102:
        tracker.MoveStrobe(4, 0);
        break;

        // I
    case 105:
        channel_data.corr_h += 4;
        tracker.SetCorrSize(channel_data.corr_w, channel_data.corr_h);
        break;
        // K
    case 107:
        channel_data.corr_h -= 4;
        tracker.SetCorrSize(channel_data.corr_w, channel_data.corr_h);
        break;
        // J
    case 106:
        channel_data.corr_w -= 4;
        tracker.SetCorrSize(channel_data.corr_w, channel_data.corr_h);
        break;
        // L
    case 108:
        channel_data.corr_w += 4;
        tracker.SetCorrSize(channel_data.corr_w, channel_data.corr_h);
        break;

        // SPACE
    case 32:
        stop_execut_flag = !stop_execut_flag;
        if (stop_execut_flag) memcpy(&cap.buf_frame.data[0], &cap.frame.data[0], (cap.frame.channels() == 1 ?
cap.frame_w * cap.frame_h : cap.frame_w * cap.frame_h * 3));
        break;

        // Q
    case 113:
        tracker.SetStrobeAutoSize(cap.frame_w, cap.frame_h);
        break;

        // R
    case 114:
        if (record_mode_flag) {
            writer->release();
            record_mode_flag = false;
        }
        else {
            std::string videoFileName = "record_" + std::to_string(video_clip_num) + ".avi";
            writer = new cv::VideoWriter(videoFileName, -1, 25, cv::Size(cap.frame_w, cap.frame_h), true);
            assert(writer != 0);
            record_mode_flag = true;
            video_clip_num++;
        }//if...
        break;

    }//switch (key)...

    return false;

}//bool KeyboardEventFunc...

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

    mouse_x = x;
    mouse_y = y;

    switch (event) {
```

```
    case cv::EVENT_LBUTTONDOWN:
        if (channel_data.mode == 0) {
            channel_data = tracker.Timelapse_Capture(mouse_x, mouse_y, current_frame_id);
            if (stop_execut_flag) stop_execut_flag = false;
        }
        else {
            channel_data = tracker.Reset();
        }//if...
        break;
    case cv::EVENT_RBUTTONDOWN:    break;
    case cv::EVENT_MBUTTONDOWN:    break;
    case cv::EVENT_MOUSEMOVE:    break;

    }//switch...

}//void MouseCallBackFunc...
```