

**TFlow Web App**

Table of Contents

[Purpose 3](#_Toc195954512)

[Overview 3](#_Toc195954513)

[General requirements 3](#_Toc195954514)

[Application modules 3](#_Toc195954515)

[Application layout 4](#_Toc195954516)

[Camera 5](#_Toc195954517)

[Preview panel 5](#_Toc195954518)

[File thumbnail 10](#_Toc195954519)

[File open 11](#_Toc195954520)

[Camera Sub-modules 15](#_Toc195954521)

# Purpose

The document describes the behavior of TFlow Web application its requirements as well as specification to the back end.

# Overview

TFlow Web application is JS based application that provides user control for TFlow camera. The camera and its control application are not supposed for regular customers, but for other developers, OEM integrators and manufacturers. Accordingly, the application style and logic should be adopted for engineers, programmers, tester, but not for dummy customers.

The Application main functions are:

* Configure camera and its modules parameters.
* Playback and management of previously recorded videos.
* General system info, settings and diagnostics.
* \*Network subsystem configuration.

The main purpose of the application is to simplify Machine Vision algorithms development and debugging based on TFlow camera, therefore the basic functions of Camera itself are quite distinct from existing cameras’ interfaces, while other functions like System settings, File management and Network configuration are quite similar.

All communication between application and client is over a local network with assumption the Internet is not available.

# General requirements

The application should be implemented using HTML Responsive Web Design technics. It should be easy and convenient to use on different platforms such as PC, Tablet and Smart Phone. For Tablets and Smartphones only most popular models can be considered. I.e. no obsolete, rare and antique models support is required.

The Application should be implemented in accordance with the provided design style guide (color scheme, controls, etc.).

# Application modules

The application consists of the following main modules:

* Camera
* \*Network
* \*System
* \*Files

Current document describes the Camera module configuration only. Other modules will be implemented in the next development waves.

## Application layout

Application should contain an extendible side bar like on the pictures below:

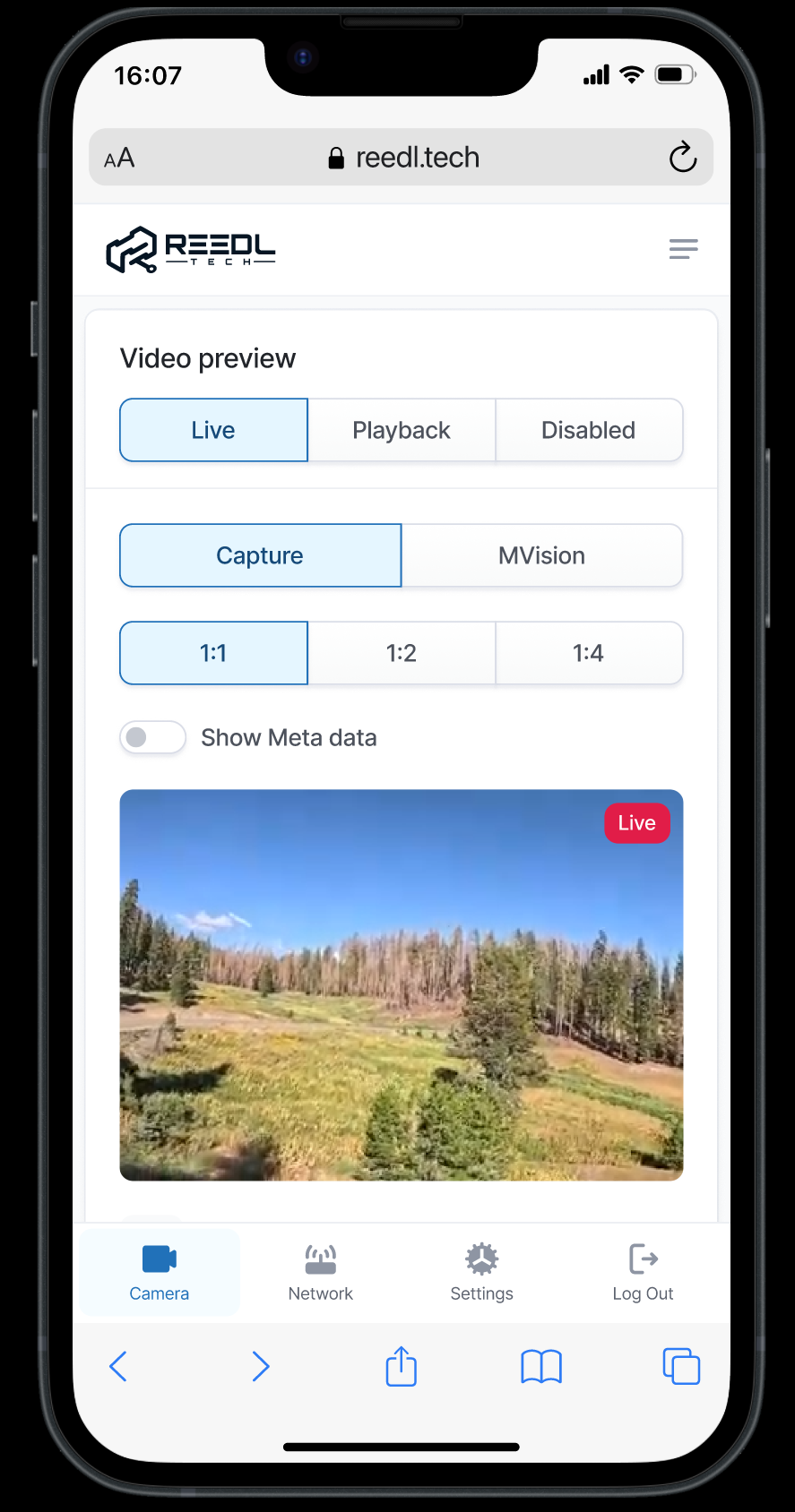
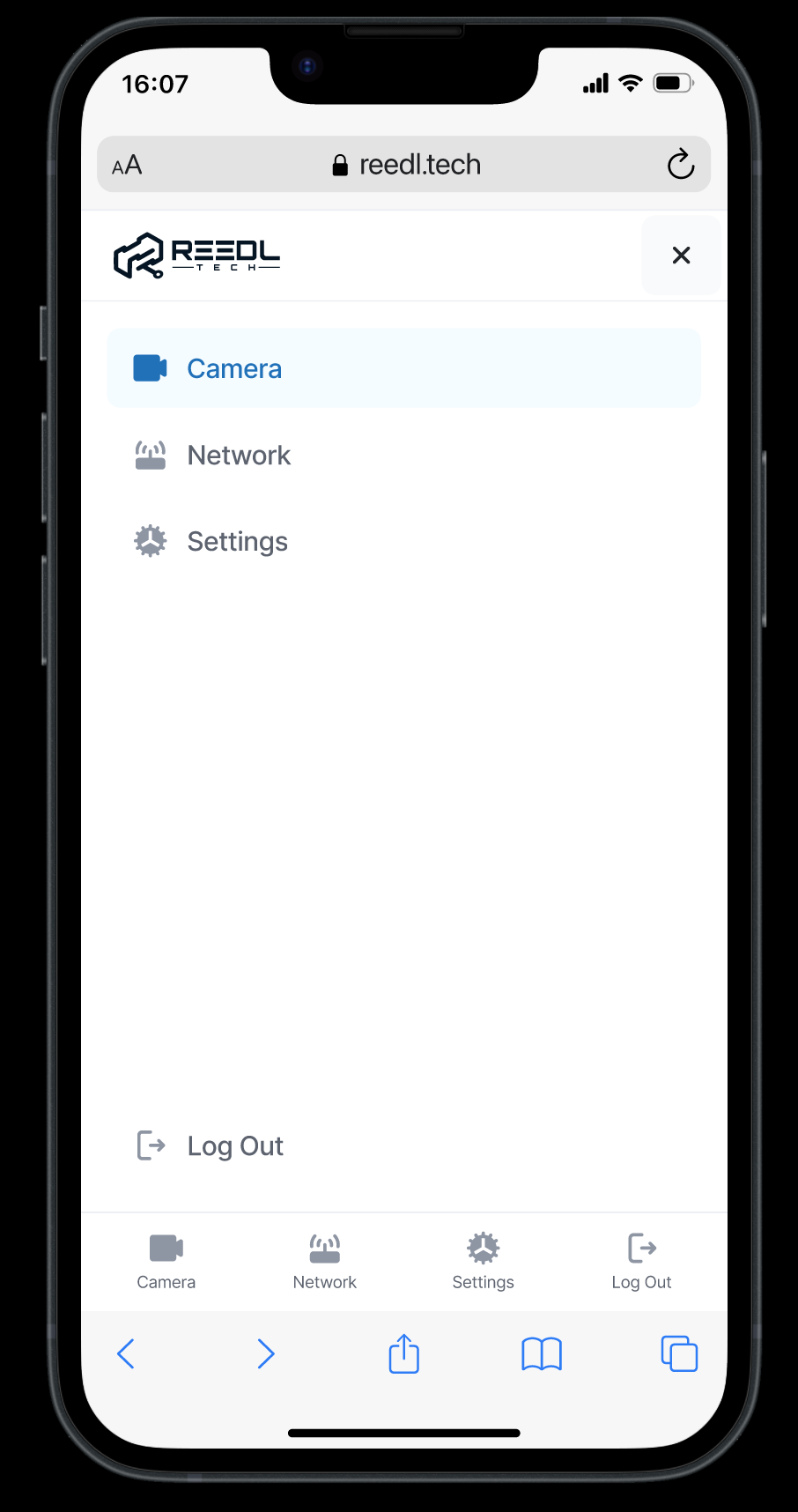
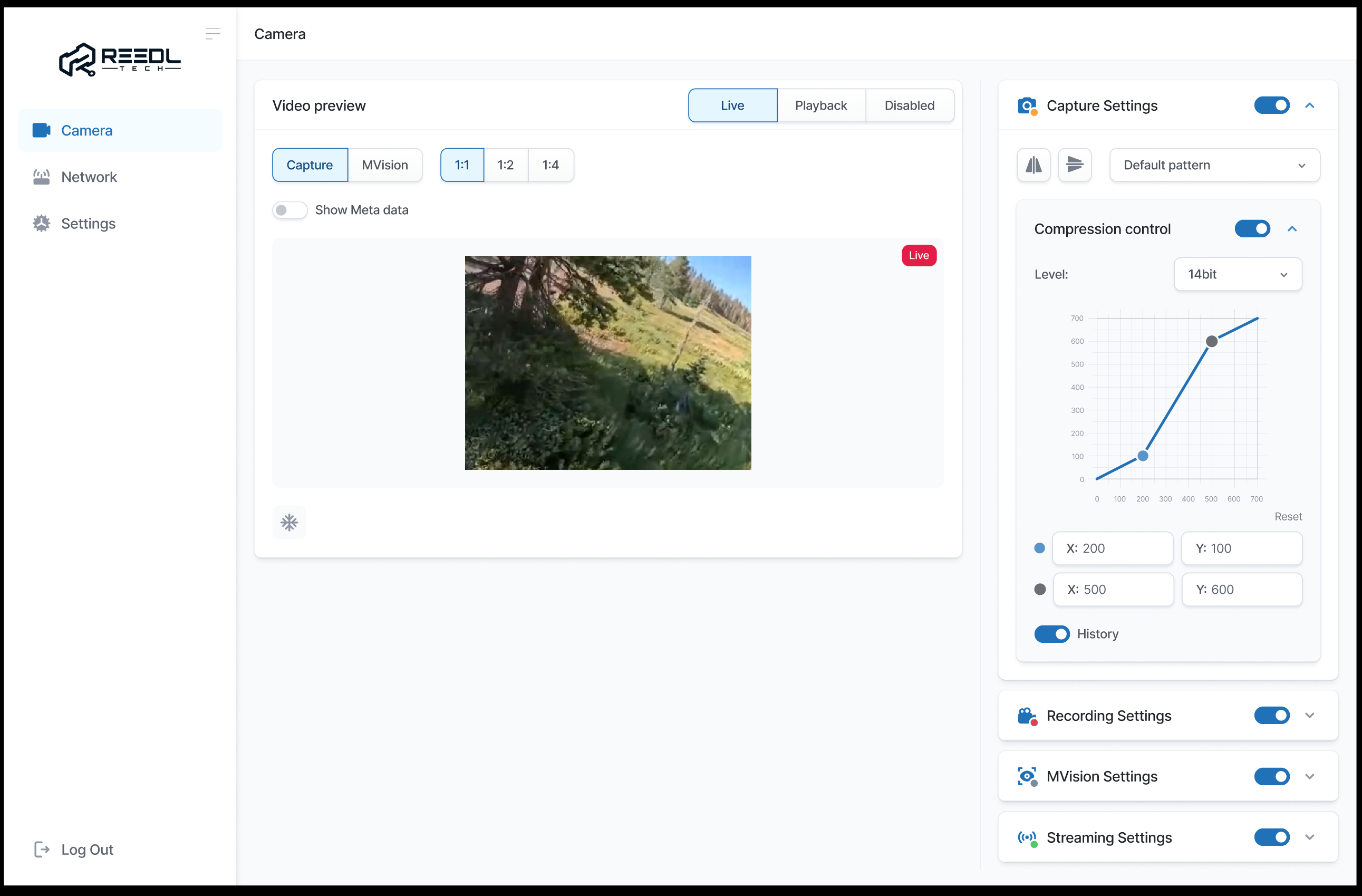
 

Figure 1. Application layout.

The side bar contains buttons or tabs for each particular module of the application:

* Camera
* Network
* System

## Camera

The Camera module is responsible for video image capture, Machin Vision processing algorithms, debug info visualization, stream recording to the local filesystem and streaming over a network.

Also, Camera tab has Video preview block with some control button and switches.

As the Camera preview block should be always in front of user, while all Camera setting may not fit to the view are, it is proposed to split Preview and Camera Settings into two panels and add a scroll bar to the Camera Setting (see Figure 2).

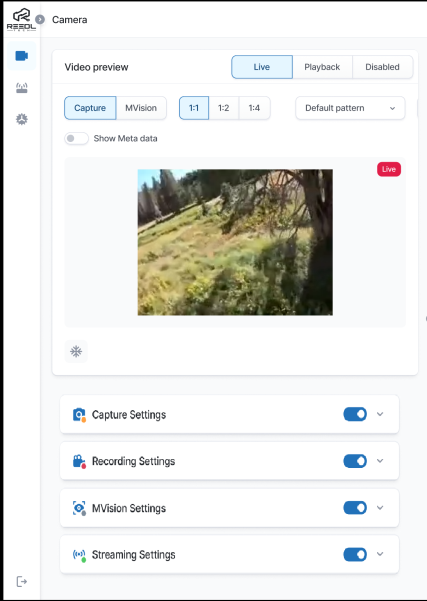
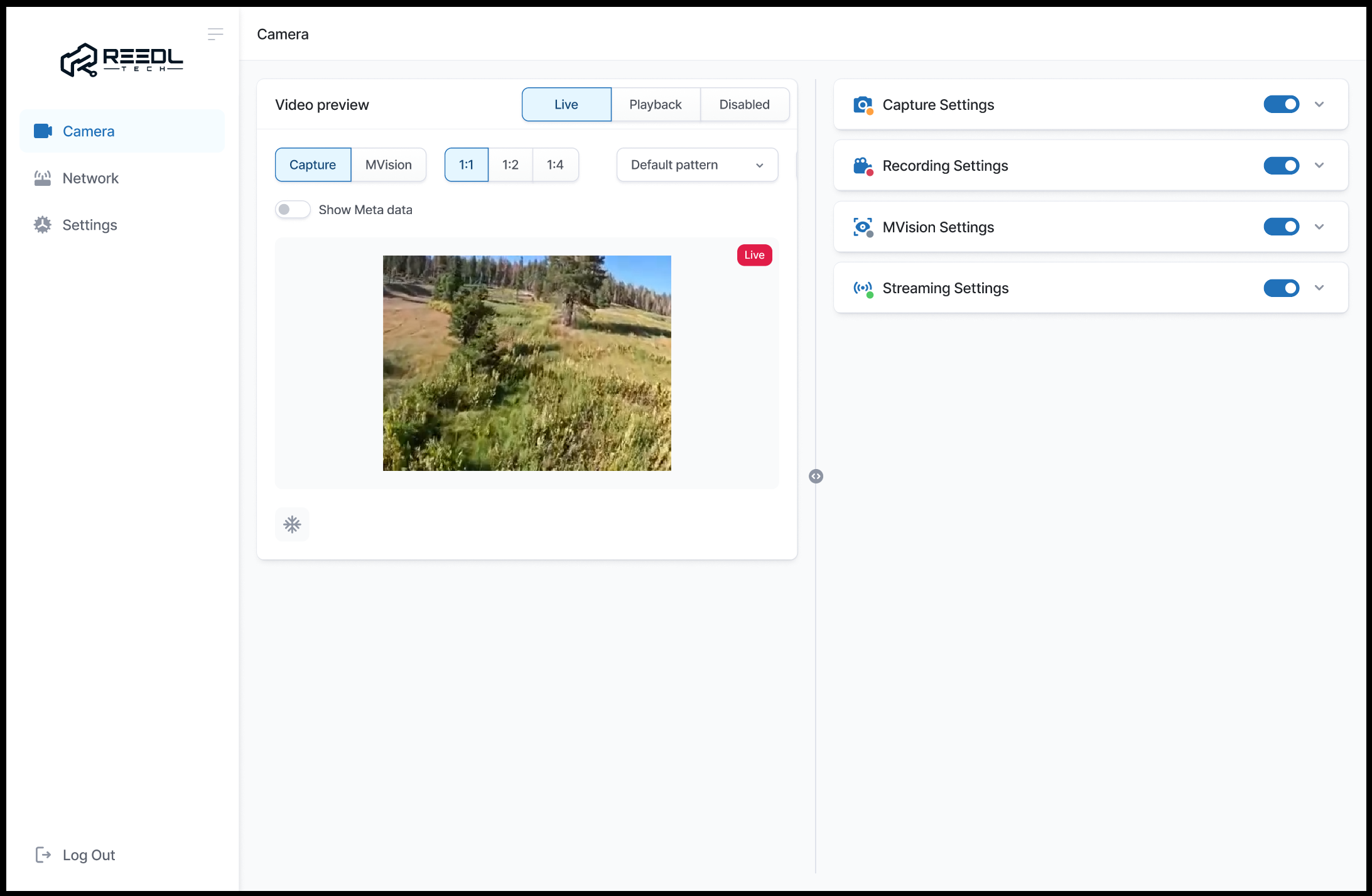
 

Figure 2. Camera module layout and responsiveness behaviour.

### Preview panel

The preview panel consists of the following sub-panels:

* Video Preview.
* Preview control.
* Playback control.

Video Preview

Video Preview is an element that decodes video stream received over WebSocket or regular UDP connection (Node.js dgram.Socket) within a *\_local\_* network.

The stream can be either MPEGTS (preferred) or if not possible, then Fragmented MP4 (ISO variant). The video is encoded by H.265 codec (aka HEVC). The input stream must be decoded and displayed as soon as possible with minimum possible delay – no buffering, dejitter and other smoothing techniques allowed. In case of network bandwidth limitation or delay it may happens that several packets arrive at the same time. In this case only a most recent frame should be displayed.

The pace control of the video stream is fully on the server side.

The Preview performance must be able to proceed @60FPS with max delay <100ms. The target delay is ~40ms.

The MPEG-TS stream along with video data contains so called Meta Data. The Machine Vision processing module uses these Meta Data as a transport for debug info. Accordingly, if configured by a user, the debug info should be extracted from the MPEG-TS stream and depicted over the video image (<https://www.oreilly.com/library/view/html5-canvas/9781449308032/ch06.html>). Another option is to use WebGL. The info is in form of primitives description - circle, rectangle, rhomb, line, dot, arrow and text with specific color and optional text. The detailed Meta Data format description is in the Chapter TBD.

The frame size can differs depends on sensor type in use as well as preview mode selected (Captue/MVision) typical frame sizes are 384x288, 640x512 and 800x600. On a desktop PC a native size video looks too small and thus there should be an option to increase Preview window scale to 1:2 and 1:4.

2024-14-10 using web decode can solve the problem. There is a latency config option as well (https://www.w3.org/TR/webcodecs/#enumdef-latencymode).

https://w3c.github.io/webcodecs/samples/video-decode-display/

Preview Control

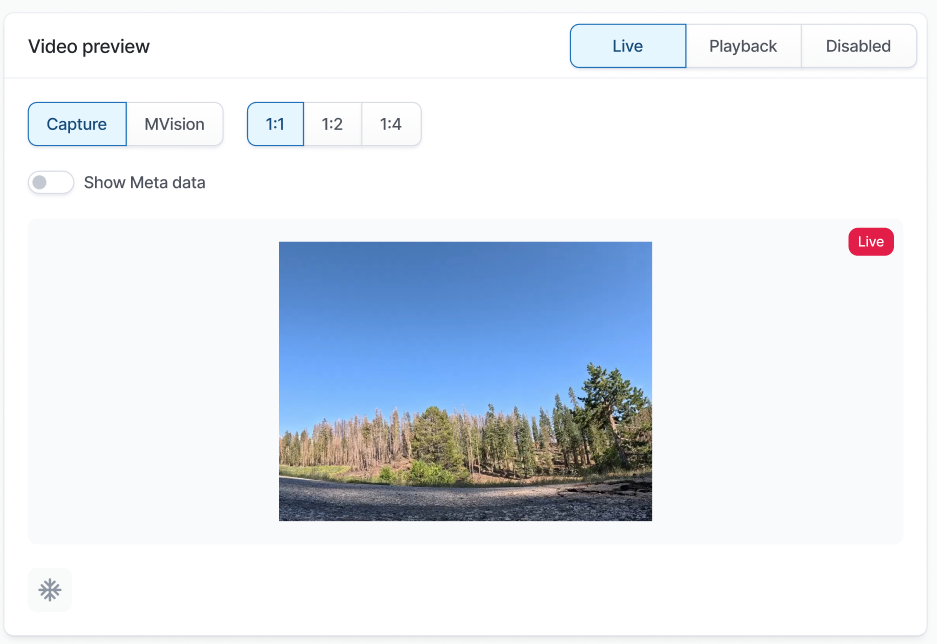


Figure 3. Preview Control panel.

1. Video preview size.

Upon any of Preview Size press the window scale changed internally on the Web Client side. No interaction with the server is supposed.

1. Freeze/Unfreeze video preview window icon. The icon is active in “Live” capture mode only.

Upon Freeze button press the Video Preview window get freeze and not refreshed until the button is unpressed. No interaction with the server is supposed.

1. Show Meta data toggle switch.

“Meta” – is toggle switch that enables/disables the Meta Data rendering over the Video Preview window.

Upon switch toggle its state should be set to disabled and the following request should be sent:

uri: /api/preview/set

method: post

body: {“preview” : { “meta” : <0|1> } }

The response will contain the actual “meta” value.

The toggle switch state should be enabled back with its actual value received in response. In case of timeout or an error the toggle switch enabled back without its value update.

1. MVision Source selector.

Selects the video source for the MVision module. Enabled only in case of Preview selector set to MVision.

* Live. Video frames captured from a camera.
* Playback. Video frames captured from the internal Player (i.e. from a local file).
* Disabled. MVision is disabled.

Request:

uri: /api/mvision/set

method: post

body:

{

“mv\_cfg” : {

“video\_src” : <“live” | “playback” | “disabled” >

}

}

Response:

body:

{

“mv\_cfg” : {

“video\_src” : <“live” | “playback” | “disabled” >

...

}

}

1. Preview Source selector.

- Capture. Preview window shows the output of the Capture module. I.e. raw data from the camera or from the player.

- MVision. Preview window show the output of Machine Vision module.

Request:

uri: /api/preview/set

method: post

body:

{

“preview” : {

“preview\_source” : <“capture” | “mvision”>

}

}

Response:

body:

{

“preview” : {

“preview\_source” : <“capture” | “mvision”>

}

}

Upon initial page load the application sends the api/preview/get and api/mvision/get requests. The backend responds with the current configuration. From the preview response the application gets “preview\_source” value and from the mvision gets the “video\_src ”.

The UI’s state should be updated accordingly. I.e. if “preview\_source” set to “capture”, then MVision selector should be disabled.

In case of video source is set to “playback”, the player’s state should be requested.

uri: /api/playback/get

method: post

body: { “player” : {} }

Response:

body:

{

“player” : {

“state” : <“pause” | “play” | “off”>,

“path”: “/path/to/last\_used\_dir/”

“file\_name” : “abc”,

“frame” : 100500,

“frames\_num” : 100500,

“frame\_rate” : -1.0, // -1.0, 0.5, 1.0, 2.0

}

Where:

“state” – PLAY/PAUSE – two main states. OFF state means the Player isn’t selected as a Video Source and normally that shouldn’t happens.

“path” – Contains the last used directory or directory of the currently selected file.

“file\_name” – File name only, without a path.

“frame” – The frame number where stream is stopped or currently played (should be updated frequently (<1sec) in “play” mode)

The UI should be updated accordingly.

If “file\_name” isn’t specified or empty the application should open empty modal “File Select” dialog (see Figure 4) and proceed with a directory read request (/api/playback/get\_dir) specified in “path”.

If “file\_name” exist and not empty, the application shows the Playback control (Figure 6).

### OpenFile dialog

The dialog opened empty and then application sends the /api/playback/get\_dir request. The backend responds with content of a directory specified in the request. The application shows the directory content in a modal window (see Figure 4). At first the application shows directory’s entries as blank thumbnail with name, date and size only. After that, in the background, the application request thumbnail files for directory entries where thumbnail file path exists.

Request:

uri: /api/playback/get\_dir

method: post

body:

{

“player\_dir” : {

“path” : “\full\path\to\dir”,

“mask” : <basic grep syntax to filter directory’s entries>

}

}

Response:

body:

{

“player\_dir” : {

“state” : “off”,

“error” : “Error message in case of error”

“path” : “\full\path\to\dir”,

“mask” : “mask specified in request”,

“items” : [

{ “name” : “a1”, “type” : <”file”|”dir”|”err”>, “date” : “13/01/2025”, “size” : 100500,

“thmb” : “optional thumbnail file path” },

{ “name” : “a2”, “type” : <”file”|”dir”|”err”>, “date” : “13/01/2025”, “size” : 100500 },

{ “name” : “a3”, “type” : <”file”|”dir”|”err”>, “date” : “13/01/2025”, “size” : 100500 },

…

]

}

}

* If “state” field is present and set to OFF, then other fields will be omitted.
* In case of file / directory open error the “error” field will contain the error description. Otherwise, the “error” field will be omitted.
* “path” - full path of current directory.
* “Items” – array of directory’s entries, where:
  + “name” – file’s or directory’s name without its path.
  + “type” – either “file” or “dir” if can be opened normally or “err” otherwise.
  + “date” – epoch time of last modification.
  + “size” – exist only for files.
  + “thmb” – full path to the thumbnail file. Optional. Can be specified for both – files and directories.

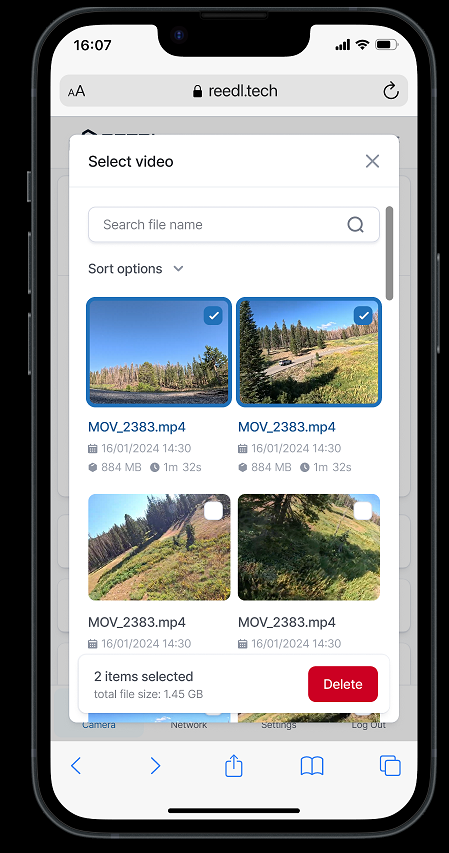
 

Figure 4. Video file selection dialog.

#### File thumbnail

All files are stored and processed on the device only. Optionally they can be downloaded through the typicall “download” user interface protocol, but the application doesn’t process file on its side. The only preview files (aka thumbnails) can be downloaded from the device and depicted on the client side.

*TODO: Define how to transfer preview file. Option1: Using json + base64; Option2: direct files download by web client.*

Thumbnail files are generated automatically by the server and, optionally, thumbnail generation generation can be triggered by a user. During a playback a user can press the button “???” and the Player will generated a thumbnail file from the currently streamed frame. Thumbnail is a regular

JPEG file with resolution ???.

Upon file selection (double click) the application sends a /api/playback/set request with the selected file. If files name syntax is OK, then server replay with “state” = wait.

*TODO: Is it worth to split file name and file path to have the same syntax with directory get?*

Request:

uri: /api/playback/set

method: post

body:

{

“player” : {

“file\_name” : <full\_file\_name>

}

}

Response:

body:

{

“player” : {

“file\_name” : <full\_file\_name>,

“state” : “wait”,

“error” : “”, // Omitted if no error.

}

}

After that the application should periodically (~100ms interval) poll the Player’s status until it switched from “wait” to something else.

*TODO: In case of state not changed within ???ms start “waiting” animation. ??? Add % completed???*

*TODO: In case of “state” and “completed\_percent” not changed within ???ms. Then drop polling and …*

Request:

uri: /api/playback/get

method: post

body: { “player” : {} }

Response:

body:

{

“player” : {

“file\_name” : <file\_name>,

“state” : “pause”,

“error” : “”, // Omitted if no error. In case of error other

// field will be omitted.

“curr\_frame” : 100500,

“frames\_num” : 100500,

“frame\_rate” : 25,

“playback\_speed” : 1.0

}

}

In case of successful response the OpenFile modal window closed and the Playback panel updated with the data received from the response. File duration calculated as frames\_num / frame\_rate, [sec].

### Playback Control

The Playback control (aka Player) is a panel that allows emulate video capturing using previously recorded media files from the local filesystem (i.e. device onboard files only).

The panel becomes active when the Video Preview is switched to Playback mode. While the Preview is in “Live” mode the panel is hidden.

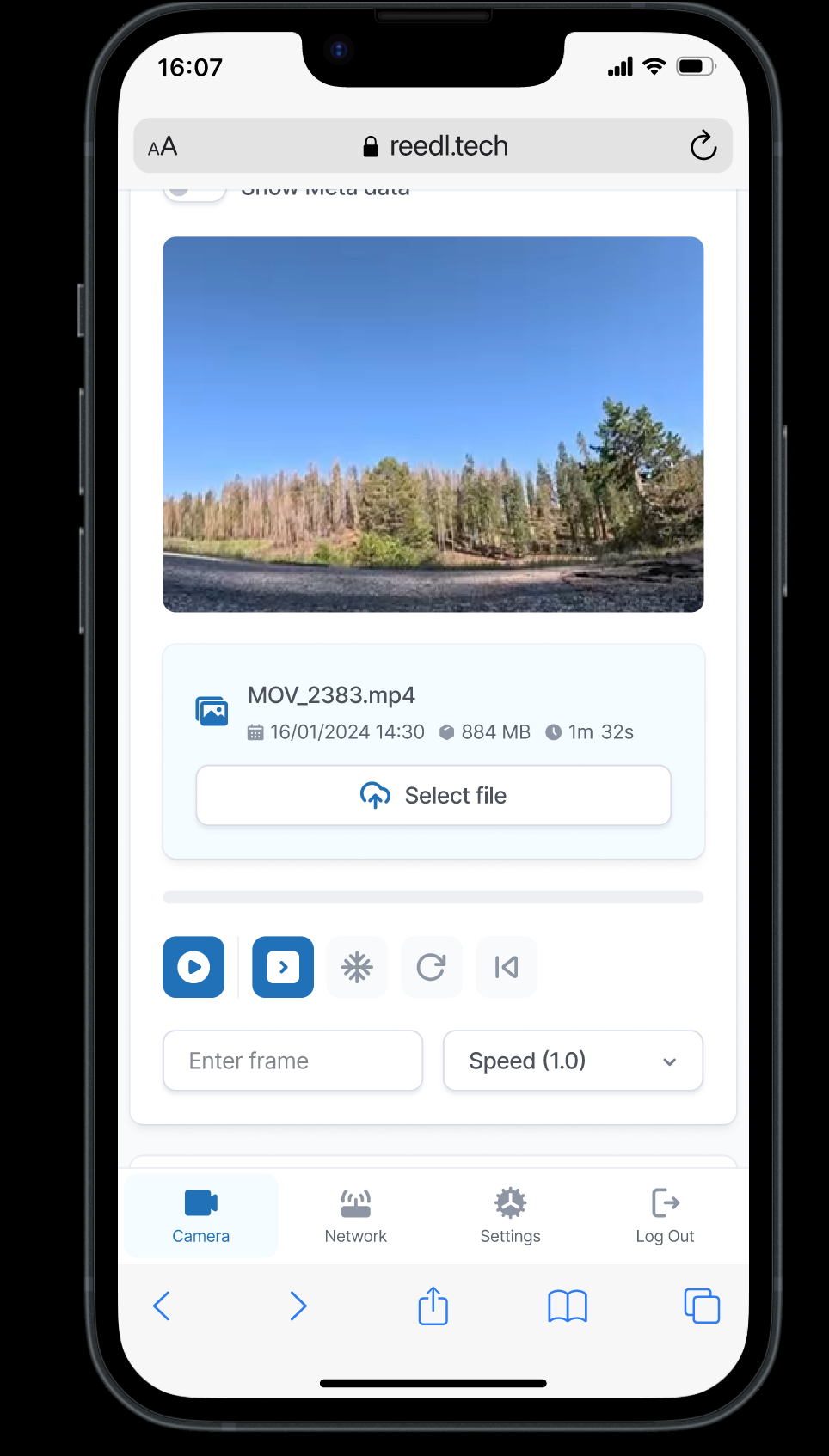
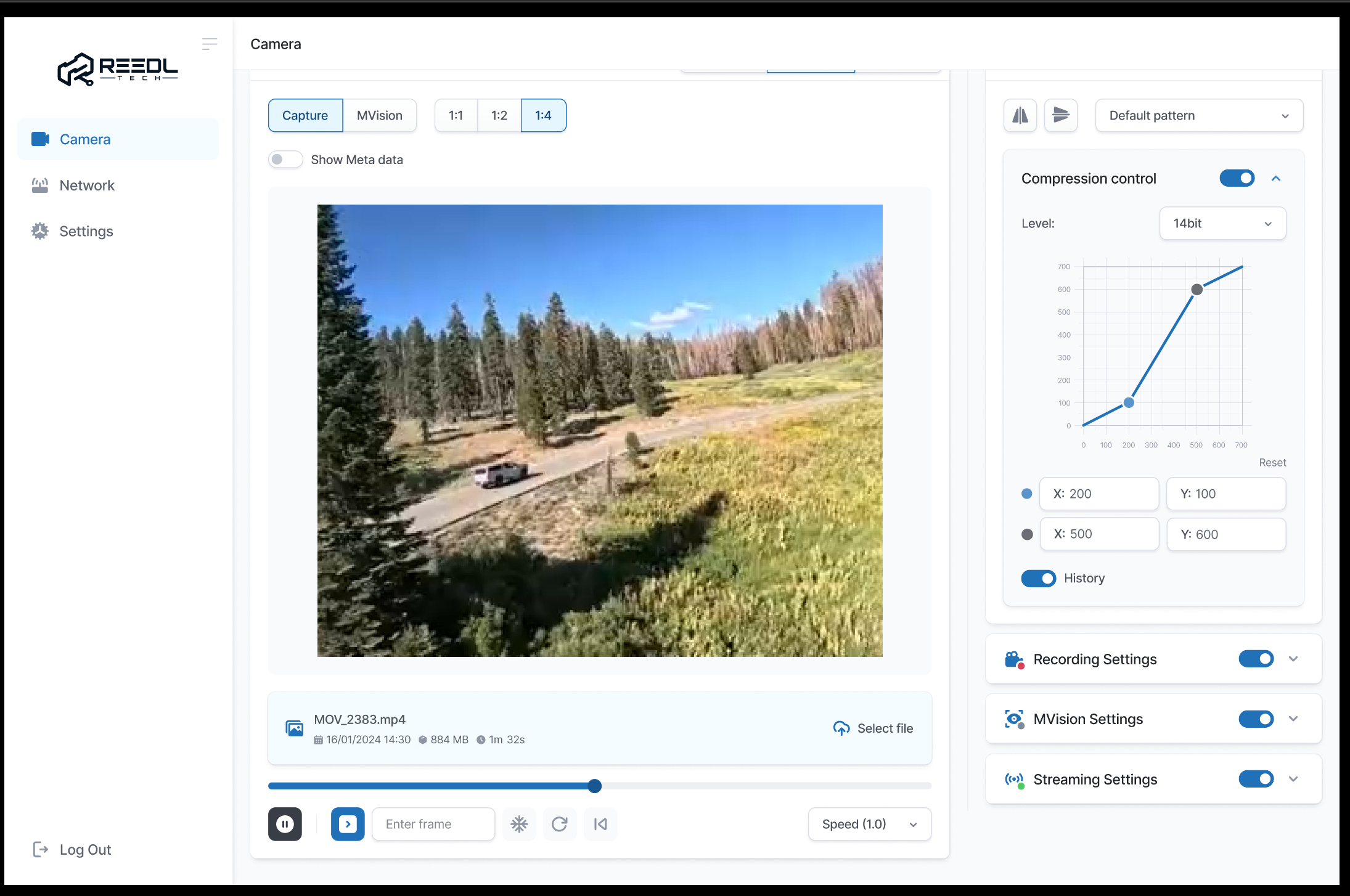
 

Figure 6. Playback panel.

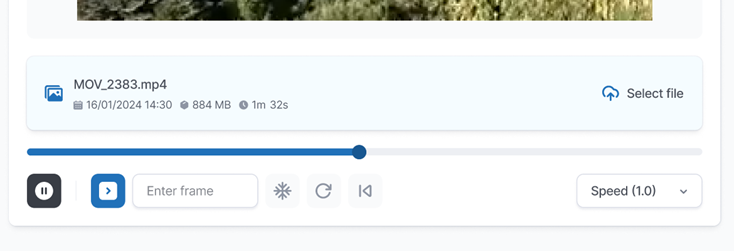
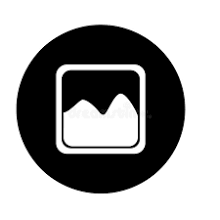


Figure 7. Playback panel layout

1. **Play/Pause.**Single button with dual state “Play” () and “Pause” (). The button is active only when media file is open and no errors. If in “Play” state, upon the button press, the application disables other playback controls and sends { “action” : “play\_pause” } request (the “frame” field should be omitted):

Request:

uri: /api/playback/set

method: post

body:

{

“player” : {

“action” : “play\_pause”,

“curr\_frame” : 100500 // Optional. Included only if Pause pressed in “Pause” state.

}

}

Response:

body:

{

“player” : {

“file\_name” : “abc”,

“error” : “”, // Omitted if no error. In case of error other field will be omitted.

“state” : “pause”,

“curr\_frame” : 100500, // The frame number where stream is stopped

“frames\_num” : 100500,

“frame\_rate” : 25,

“playback\_speed” : -1.0, // -1.0, 0.5, 1.0, 2.0

}

}

Upon response the application updates Play/Pause, Frame#, Navigation bar and Frame rate Controls using the response data.

If in “Pause” state, upon the button press, the application disables other playback controls and sends the { “action” : “play\_pause” } request with optionally “curr\_frame”. If “curr\_frame” specified, then playback will start from the specified frame number.

Request:

uri: /api/playback/set

method: post

body:

{

“player” : {

“file\_name” : “abc”,

“action” : “play\_pause”,

“curr\_frame” : 100500, // The frame number to start from

“playback\_speed” : -1, // -1.0, 0.5, 1.0, 2.0

}

}

Response:

Same as for “Pause”, except the state set to “Play”

During the playback the application should periodically send empty /api/playback/get requests and update the Playback control from the response info. The response format is the same as for /api/playback/set request.

While in “Play”mode the “Frame#” (3) field should be disabled for edit.

1. **Step.**If in “Play” state the functionality is the same as for “Pause” button. If in ”Pause” state, then sends another “Pause” request with frame number (3) +1.
2. **Frame#**  
   The field is an edit box. The field is disabled while in “Play”state. In “Pause” state the field is editable. Upon field update the application sends the same “Pause” request as specified above with the newly entered frame number value.
3. **Freeze**  
   TBD
4. **Reset**   
   The button reset the algorithm’s state and it is part of MVision module rather than player. Just send the following request. No any data expected in response.

Request:

uri: /api/mvision /set

method: post

body: { “mv\_algo” : { “reset” : 1 } }

1. **Restart**  
   Just send the request.

Request:

uri: /api/playback/set

method: post

body:

{

“player” : {

“restart” : 1

}

}

1. **Navigation bar**

Upon a click of slider move the application calculates new current frame number and sends the Pause request.

1. **Create thumbnail**

Creates a thumbnail still picture from the current frame for the currently opened file.

Request:

uri: /api/playback/set

method: post

body:

{

“player” : {

“thumb\_create” : 1

}

}

1. **Speed**

The button toggles over the states which are correspond to the following values:  
[-1.0, 0.5, 1.0, 2.0]

Where:

-1 – As fast as possible.

0.5…2.0 – Normalized playback speed.

In “Pause” state – do nothing, just updates the icon state.

In “Play” state – sends the “Play” request with omitted “frame” field.

### Camera Sub-modules

The TFlow backend side consists from the several modules which are connected by native linux IPC means (fifo, pipes). The simplified diagram is depicted below (Figure 8).

Due to different reasons the modules may not be always available and thus the application must follow their states.



Figure 8. Camera module internal diagram.

The Camera consists of following sub-modules:

* Capture

Captures input video stream and distribute it over other sub-modules. The input source for the capture can be either TFlow hardware sensor or previously recorded video. In case of capture from the sensor the dedicated Capture Setting panel allows configure the video driver’s parameters. In case

* Processing  
  Receives video frames from the Capture module, process them and optionally sends to the streaming module.

The result of processing algorithm can be either rendered directly to the frame of send to the Streaming module in form of OSD Meta data.

OSD Meta data is in form primitive description – triangle, circle, rectangle, text, etc. with some attributes like color and size.

* Streaming  
  The Streaming module, depend on configuration, receives data either from the Capture or from the Processing module. The input frames optionally mixed with rendered OSD Metadata, encoded and packed into MPEG-TS stream. Another option – OSD Metadata is not rendered, but added to the MPEG-TS stream according to the standard.

Finally the prepared stream uni/multi-casted to the local network using UDP/IP and to the connected web clients over WebSocket or UDP connection.

Optionally the prepared MPEG-TS can be stored on a local filesystem.

* Control

The Control module is the HTTP server for the Web Clients and IPC client for other modules which are acting as IPC servers.

Control module receives WEB request, parse them, forward to the corresponding sub-module and reply with the received result.

Settings for each module combined into a collapsable panel (Figure 9).

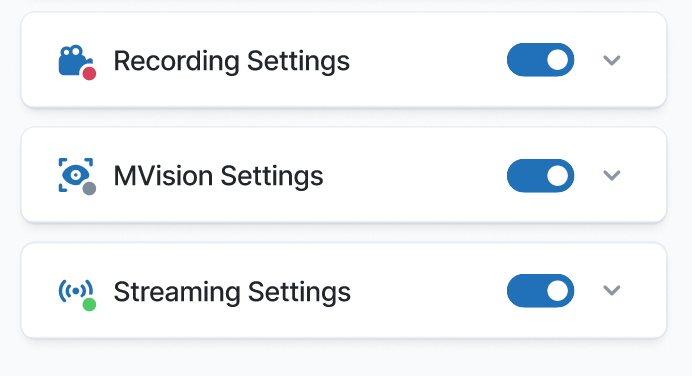


Figure 9. Sub-module control panel.

**Module State Dot**

 – Error. Module is not connected.

 – Grey. Connected but not active.

 – Red blinking. Recording.

 – Green. For streaming only.

Blank – Connected and active.

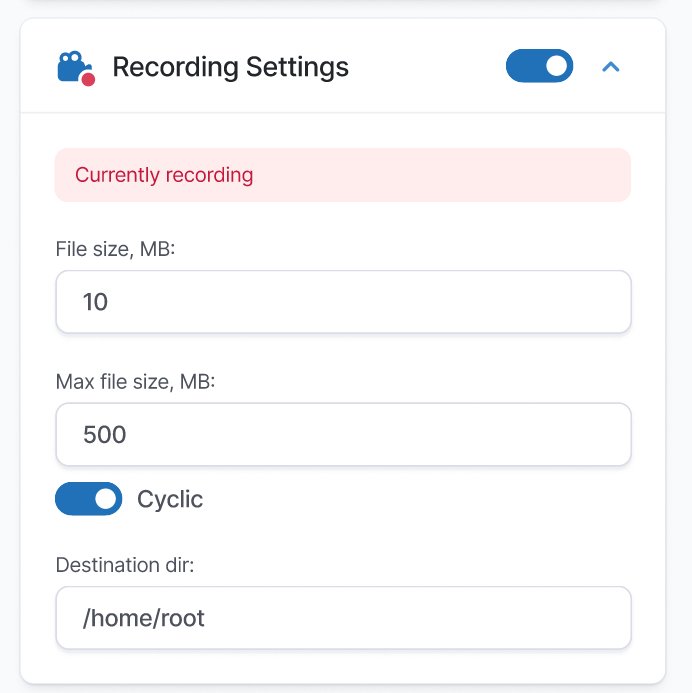
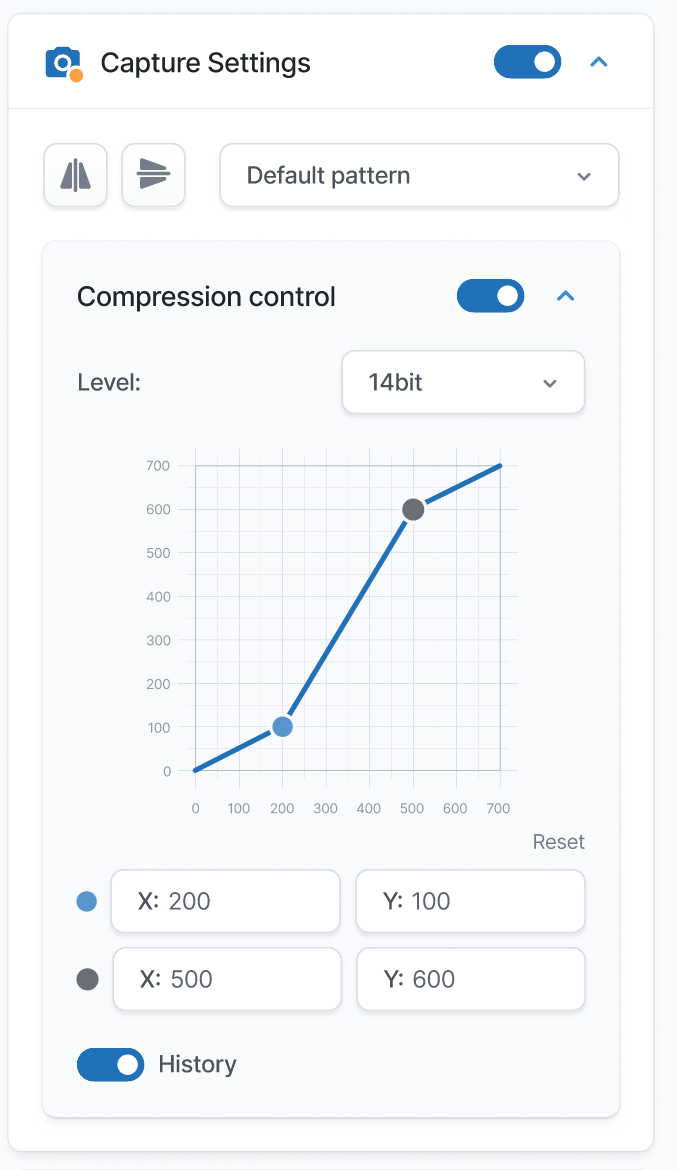
**Quick Access**

The function depends on the module.

**Extend/Collapse panel**

Self evident.

Sub-modules settings



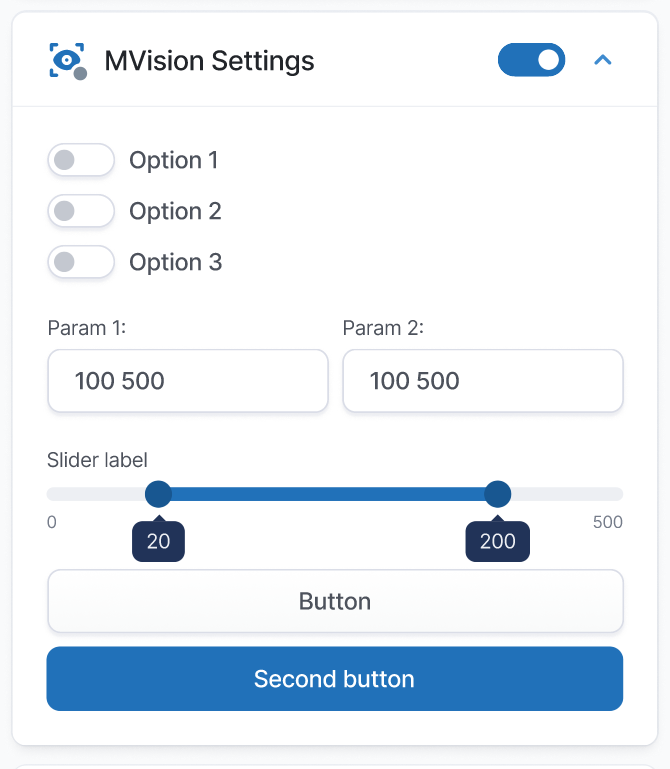


Figure 10. Modules controls.

Each of submodules are defined on the device side and all required information for its control elements rendering should *be obtained from the corresponding module Response* in <ctrl\_N> fields.

Each control <ctrl\_N> object defined with the following JSON description.

{

“name” : <literal>,

“label” : <literal>,

“type” : <literal>,

“state” : <number>,

“value” : <depends on type>,

“size” : <number>

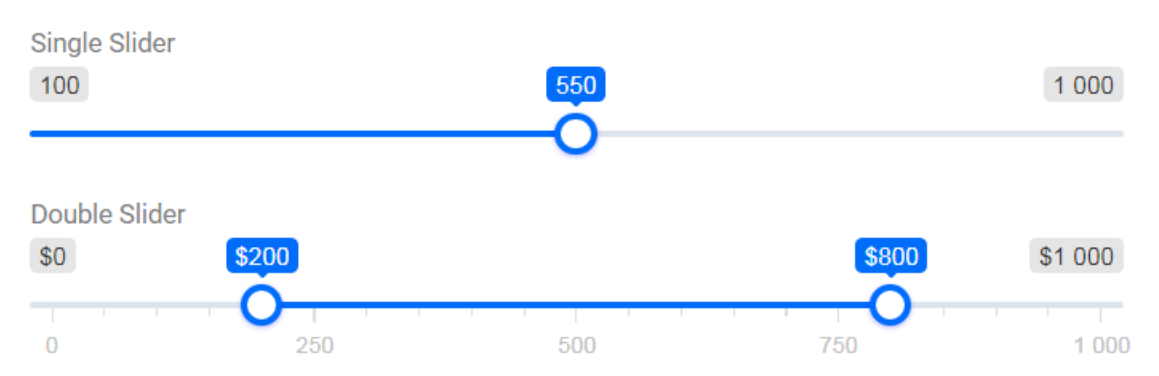
}

“name” - Control name. Not exposed to a user. For internal use only.

“label” - Text label shown in settings panel.

“type” - Type of the control:

* “edit” – edit box the value passed and stored as literals.
* “switch” – a regular switch (checkbox). The value is 0 or 1.
* “button” – The value is 1 if button is pressed.
* “dropdown“ – dropdown list. The value is an array with literals, where 1st element contains current control value, while other elements are the list members.
* “slider\_h” – horizontal slider bar. The value is an array of integer [current, min, max, size]
* “slider2\_h” – dual horizontal slider bar. The value is an array of integer [current\_1, current\_2, min, max].



“state” - State of the control. 1 – enabled, 0 – disabled.

“value” - see above in type description

“size” – in symbols. If = 0, then 100%

For example: The Capture module request & response.

Request:

uri: /api/capture/get

method: post

Response:

body:

{

“capture” : {

“state” : “red”,

“hdr\_control” : { <ctrl\_0> },

“controls” : [ <ctrl\_1>, <ctrl\_2>, … ]

}

}

Where:

“controls” - items are JSON objects in format described above.

“hdr\_control” – the control that placed in panel upper side and accessible when a panel is collapsed.

Upon parameters change the App sends the set request with name and a current value only.

??? Response also contains only values of the controls. ???

Request:

uri: /api/capture/set

method: post

body: {

“capture” : {

“hdr\_control” : {

“value” : 1,

},

}

}

Response:

body:

{

“capture” : {

“hdr\_control” : { “value” : 1 },

“controls” : [ {“option1” : {“value” : 1}}, {“option2” : {“value” : 2}}, ... ]

}

}

In addition, in case of complex control, its description can be defined on WEB side and device will send only input parameters referencing the control by its name.

For example, the Capture module setting panel has a dedicated control – “Compression” which can be allocated in a separate collapsible panel inside the Capture settings.

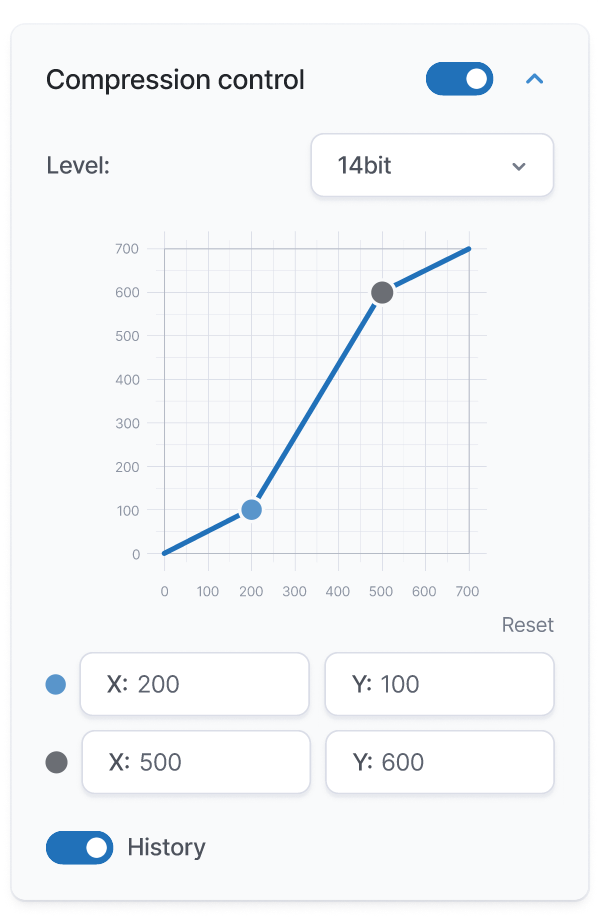


Figure 11. Compression control.

The Compression panel consist of the following controls:

Header control – ON/OFF switch. If compression is disabled, then there are no other controls.

Histogramm – checkbox. If enabled, then the histogram should be displayed on the graph under the compression line.

“Reset”– Compression reset button. Returns the compression handlers back to initial state.

Grid – Shows current values of left and right handlers. The X axis - min 0, max 16384. The Y axis – min 0, max 256.

Handlers – Movable handlers. Each handler has X and Y coordinates according to Grid axes. The handlers’ coordinates also shown in the editable fields.

During the handlers dragging the following rules must be followed:

1. Left X < Right X
2. LeftY < RightY
3. Left X > 0
4. Right X < 16382
5. Right handler pushes blue to the left if Left X == Right X + 1
6. Left handler pushes right to the right if Left X == Right X – 1
7. Same as 5, 6 but on Y axis. Right handler pushes the Left one down, Left handler pushes the right one up.

Compression control response:

{

“name” : “compression\_v0”,

“label” : “Compression”,

“type” : “custom”,

“state” : 1,

“value” :

[

{

“name” : “compr\_en”,

“label” : “Enable”,

“type” : “switch”,

“state” : 1,

“value” : 0

},

{

“name” : “hist\_en”,

“label” : “Enable”,

“type” : “switch”,

“state” : 1,

“value” : 1

},

{

“name” : “hist”,

“label” : “Hist”,

“type” : “custom”,

“value” : [0, 10, 20, 10, 5...] // array of 16 integers

},

{

“name” : “reset”,

“label” : “Reset”,

“type” : “button”,

“state” : 1,

“value” : 1,

},

{

“name” : “h\_left”,

“type” : “custom”,

“value” : [<x>, <y>],

},

{

“name” : “h\_right”,

“type” : “custom”,

“value” : [<x>, <y>],

}

]

}

Compression control request:

{

“name” : “compression\_v0”,

“value” :

[

{

“name” : “compr\_en”,

“value” : 0

},

{

“name” : “hist\_en”,

“value” : 1

},

{

“name” : “reset”,

“value” : 1,

},

{

“name” : “h\_left”,

“value” : [<x>, <y>],

},

{

“name” : “h\_right”,

“value” : [<x>, <y>],

}

]

}

MVision module settings

Streaming module settings

Recording module settings