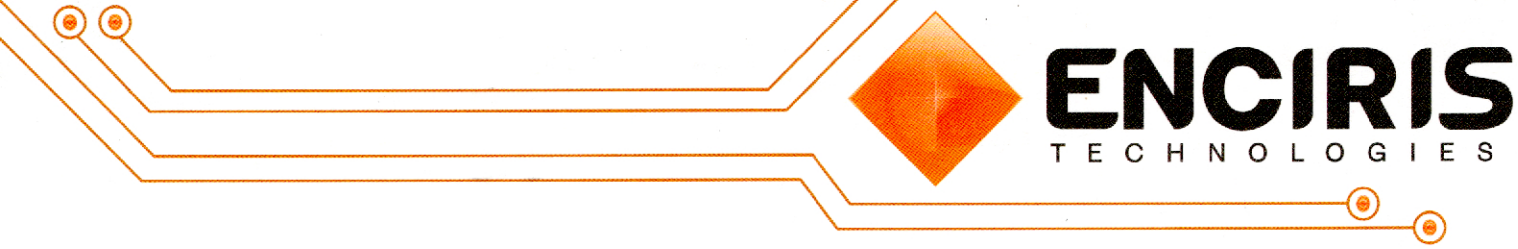


LT310 API

The programmer guide

Enciris Technologies

Version 1.4.0, 28/11/2025

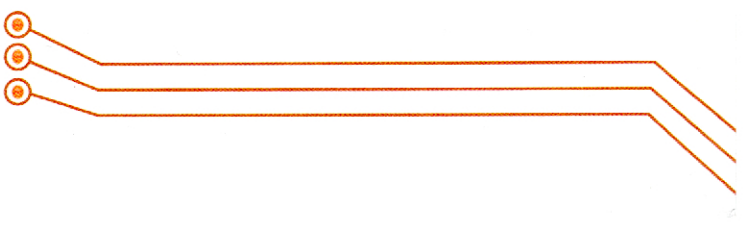


Table of Contents

Chapter 1. Introduction

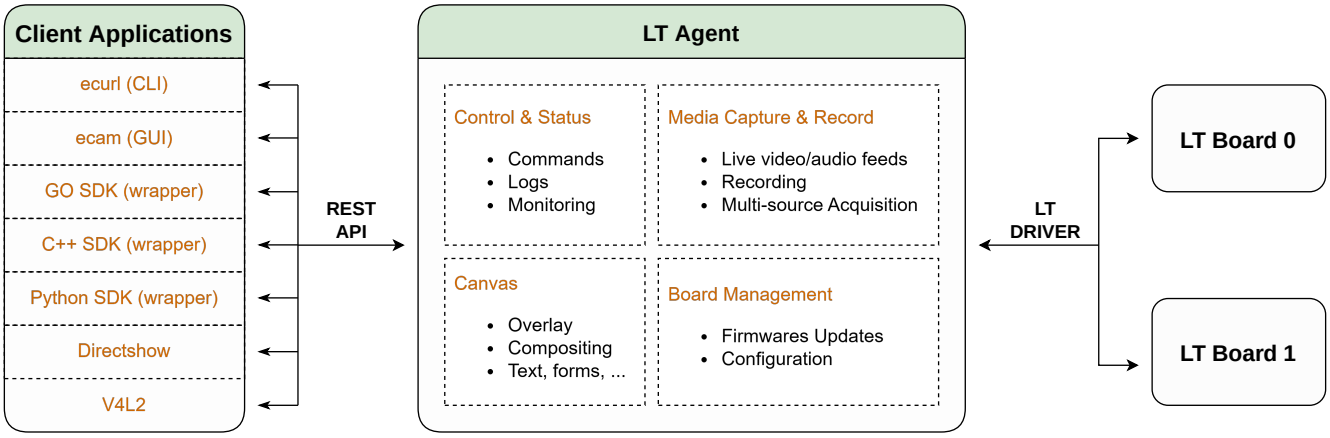
The **LT API** is built around the REST architecture (Representational State Transfer). REST defines a structured approach for exposing system functionalities via a consistent interface.

A REST API is typically accessed using predefined URLs, which represent various resources returned as JSON objects. These resources support standard methods such as `GET`, `POST`, and `DELETE`.

The **LT API** requests are processed by a host service called **lt310agent**, which utilizes standard OS mechanisms such as IPC, SG-DMA, and shared memory. This ensures minimal latency when handling API requests. To optimize performance, video and audio data are shared among consumers using memory segments, allowing large data buffers and concurrent access.

With **LT boards** designed to be seen as *hardware-as-a-service* approach, the **LT API** abstracts hardware-specific implementations behind a unified API. This allows users to focus on core functionalities such as capturing, playing, recording, and streaming audio/video data.

The **LT API** is accessible through various clients, including the **ecurl** command line interface (CLI) and the **ecap** graphical user interface (GUI). The API can also be accessed directly through C++, Python, DirectShow, V4L2, NamedPipe, and UART.



Chapter 2. Installation

2.1. Windows

Download and run the latest **lt310install_x.x.x.exe** to install the **lt310 family** drivers, tools and services. If necessary the previous version will be uninstalled.

Once installed, the **lt310agent** service will be started automatically and will be (re)started automatically with each system (re)boot.

The **lt310agent** can be controlled using the Windows Services Manager or the command line. For more details, see [control_service].

The **LT boards** plugged into the host should appear in the Windows Device Manager under the **Sound, video and game controllers**.

Directshow

All the boards inputs are accessible through directshow filters, and then are available in any directshow compatible application. The directshow filters are managed by the **lt310agent** service, so you can use them only if the service is running.

NOTE	To uninstall the **lt310 family** drivers, tools and services, run the installer and choose "Remove" or use "Apps & features" menu.

2.2. Linux

Download and extract the latest **lt310install_x.x.x.tar.gz**, then run the **lt310install.sh** script to install the **lt310 family** drivers, tools and services. If necessary, the previous version will be uninstalled.

Upon successful installation, the **lt310agent** daemon will start automatically and will also (re)started with each system (re)boot. If the installation fails, contact us and check the **lt310install.log** file for details.

The **lt310agent** can be controlled using the command line. For more details, see [control_service].

V4L2

All the boards inputs are accessible through V4L2 drivers, and then are available in any V4L2/GStreamer compatible application. The V4L2 drivers are managed by the **lt310agent** service, so you can use them only if the service is running.

NOTE	To uninstall the **lt310 family** drivers, tools and services, run lt310_uninstall.sh from the installation directory.

2.3. SDK

Download the latest **lt310sdk_x.x.x.zip** or **lt310sdk_x.x.x.tar.gz** archive and extract it anywhere you want. The SDK contains the API documentation, the API libraries and examples for the following languages **Go**, **C++** and **Python**.

Please navigate through the examples to learn how to program the API.

You can also create scripts using the **ecurl** command-line tool. For more details, see [ecurl].

2.4. Tools

Two tools are installed along with the **lt310agent** service:

- **ecurl** - a command-line tool to send REST API requests to the **lt310agent**.
- **ecap** - a graphical user interface to control and/or test LT boards.

By default, the service and tools are added to the **PATH** environment variable, allowing you to use them from any command line.

2.5. lt310agent Controls

To access the **LT API**, ensure that the **lt310agent** is installed and running on your host system.

You can use the following commands, which require administrative privileges, to control the **lt310agent**.

Start the lt310agent

```
$ lt310agent start
```

Stop the lt310agent

```
$ lt310agent stop
```

Check the lt310agent and boards firmware version

```
$ lt310agent version
```

Check the lt310agent status

```
$ lt310agent status
```

2.6. Board Firmware

If the boards installed in the host require a firmware update, the **lt310agent** service will automatically update them when the service starts. This process may take up to 2 minutes, depending on the board type. The service will be unavailable until the update is completed.

Please use the command below if you want to check the firmware version of the boards.

Check the lt310agent and boards firmware version

```
$ lt310agent version
```

It is also possible to manually update the board firmware.

Update the boards firmware

```
$ lt310agent update
```

NOTE	To update the board firmware, the **lt310agent** must be stopped.

Chapter 3. ecurl (CLI)

The **ecurl** program is a developer tool to help you make requests on the **LT API** directly from your terminal. The tool is deployed along with the **lt310agent** at the installation stage. The tool has been developed with our SDK and is available for both Linux and Windows platforms.

You can use the **ecurl** CLI to:

- Create, retrieve, update or delete **LT API** objects.
- Play and record any video or audio resources.
- Use the multi-channel feature of the **LT boards**.
- Control and test the installed **LT boards**.

NOTE	The **lt310agent** must be running otherwise **ecurl** will not work.

3.1. GET command

```
$ ecurl get <url>
```

Send a GET request to retrieve a specific API object identified by the <**url**>.

▼ **Examples**

Retrieve information about the lt310 family agent

```
$ ecurl get lt310:/
```

Retrieve informaton from lt310 board located at index 0

```
$ ecurl get lt310:/0
```

Make a JPEG capture

```
$ ecurl get lt310:/0/{in}/0/file -d type=image/jpeg
```

3.2. POST command

```
$ ecurl post <url> [-d @file.json] [-d field=value] [-d data=@file.bin]
```

Create or modify the resource designated by the <**url**>.
Arguments may be added to the request with the **-d** optional flags. It is possible to use a file content as input by preceding the filename with the @ character. By preceding the filename with the **$** charater, relative path will be translated to the absolute full path.

▼ **Examples**

Create a virtual video input from a mp4 file

```
$ ecurl post lt310:/canvas/0/init -d source=$video.mp4
```

Load a custom HDMI-IN Edid

```
$ ecurl post lt310:/0/hdmi-in/0/edid -d data=@custom.edid
```

3.3. DELETE Command

```
$ ecurl delete <url>
```

Delete or reset the resource pointed by the <**url**>.

▼ **Examples**

Unplug virtual video input

```
$ ecurl delete lt310:/canvas/0
```

3.4. PLAY Command

```
$ ecurl play <url> [-d]
```

Play video or audio source until `Ctrl` + `c` is pressed.
Arguments may be added to the request with the **-d** optional flags.

▼ **Examples**

Live display of hdmi-in video

```
$ ecurl play lt310:/0/hdmi-in/0 -d media=video/yuyv
```

Live listening of hdmi-in audio

```
$ ecurl play lt310:/0/hdmi-in/0 -d media=audio/pcm
```

3.5. REC Command

```
$ ecurl rec <url> [-d]
```

Record video or audio source until `Ctrl` + `c` is pressed.
Arguments may be added to the request with the **-d** optional flags.

▼ **Examples**

Record hdmi-in video into a mp4 file

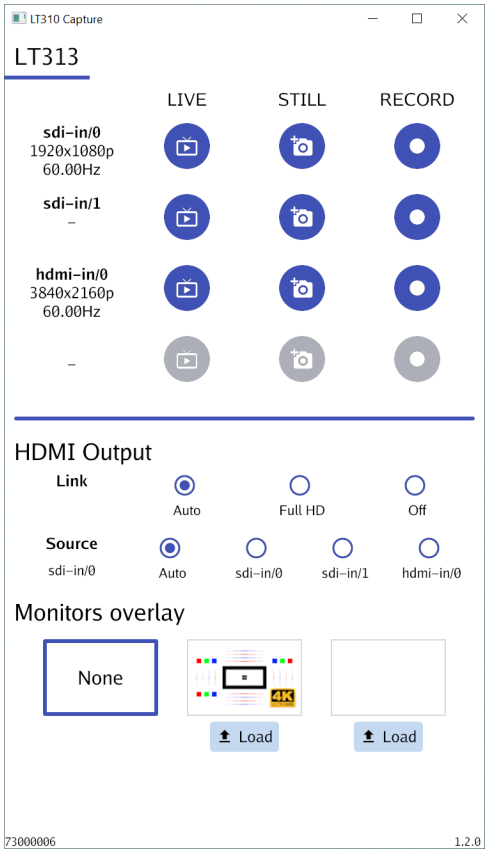
```
$ ecurl rec lt310:/0/hdmi-in/0/file -d media=video/mp4
```

Record camera video using NVENC hardware encoder and hevc codec

```
$ ecurl rec lt310:/0/camera/0 -d media=video/mp4 -d extra.hw=nvenc -d extra.codec=hevc
```

Chapter 4. ecap (GUI)

The **ecap** program is a simple graphical user interface designed to capture and record using the **LT310 boards**. The tool has been developed with our SDK and is available for both Linux and Windows platforms.



You can use the **ecap** GUI to:

- Play and record any video or audio content.
- Use the multi-channel features of the **LT boards**.
- Control and test the installed **LT boards**.

NOTE The **lt310agent** must be running; otherwise, **ecap** will not work.

Chapter 5. API Description

5.1. Endpoint Structure

An API endpoint is a URL where the API processes requests for a specific resource. Endpoints are accessed via URLs using the syntax `scheme:/path`, which consists of:

- A non-empty scheme component followed by a colon `lt310:`
- A path component made up of path segments separated by slashes `/`

For clarity in this documentation, the **scheme `lt310:` is omitted** from endpoint URLs.

5.2. Request and Response Format

All API endpoints use JSON format for requests and responses. Endpoints are divided into two categories:

- **Read-write endpoints**: Support both GET and POST methods to retrieve and modify parameters
- **Read-only endpoints**: Support only GET method to retrieve current state

5.2.1. Retrieve parameters (GET)

To retrieve parameters, send a GET request to the appropriate endpoint. The server returns the complete object with all current parameter values.

Example: GET request

request

```
{
    "method": "GET",
    "url": "lt310:/0/endpoint",
    "body": null
}
```

response

```
{
    "parameter1": "value1",
    "parameter2": 42
}
```

5.2.2. Update parameters (POST)

To update parameters, send a POST request with the attributes you want to modify. You don't have to send the complete object.

The server will:

- Validate the new values
- Apply the requested changes
- Return the complete object with all current values (including unchanged ones)

Example: POST request

request

```
{
    "method": "POST",
    "url": "lt310:/0/endpoint",
    "body": {
        "parameter1": "newValue"
    }
}
```

response

```
{
    "parameter1": "newValue",
    "parameter2": 42
}
```

NOTE Specific examples with actual endpoints and parameters are provided in each endpoint's documentation section.

5.2.3. Error Handling

If a request fails, the server returns an error response with an explicit message describing the cause of the failure.

Common error causes include:

- Invalid parameter values (out of range)
- Missing required parameters
- Invalid endpoint or resource ID
- Hardware not responding

Example error response

```
{
    "error": "invalid parameter value",
    "message": "parameter must be between 0 and 100, got 150",
    "code": 400
}
```

5.3. Audio/Video structures

This section describes the JSON data structures used to represent audio and video stream information. These structures are embedded in various endpoints related to audio/video sources and outputs.

5.3.1. Audio object

The `audio` structure contains detailed information about an audio stream.

Audio object

```
{
    "audio": {
        "description": "",
        "format": "",
        "channels": 0,
        "samplerate": 0,
        "depth": 0,
        "signal": "none"
    },
}
```

Attribute	Type	Description
description	string	A short description of the audio signal.
format	string	The audio sample format `pcm`.
channels	int	The number of audio channels.
samplerate	int	The number of audio samples per second.
depth	int	The number of bits per audio sample.
signal	string	The audio signal status: `none` (not found), or `locked` (ready to use).

5.3.2. Video object

The `video` structure contains detailed information about a video stream.

Video object

```
{
    "video": {
        "description": "",
        "format": "",
        "size": [0, 0],
        "framerate": 0,
        "interlaced": false,
        "signal": "none"
    }
}
```

Attribute	Type	Description
description	string	A short description of the video signal.
format	string	The pixel color format `rgb444`, `yuv444` or `yuv422`.
size	[2]int	The video frame width and height in pixel units.
framerate	float	The number of video frames per second.
interlaced	bool	The video frame interlaced status.

Attribute	Type	Description
signal	string	The video signal status: `none` (not found), or `locked` (ready to use).

5.4. Agent

The **agent** endpoint allows to retrieve the **lt310agent** software version.

Endpoint	Method	Description
/	GET	Retrieve the current agent information.

5.4.1. Agent Object

The **agent** object provides information about the currently running **lt310agent** software.

GET **lt310:/**

```
{
    "version": "1.4.0"
}
```

Attribute	Type	Description
version	string	The current agent version.

5.4.2. Agent Configuration File

The agent can be configured using an optional configuration file named `lt310agent.cfg`. Configuration options:

Option	Description
`WD`	Working directory for the agent (default: current working directory)
`numCanvases`	Number of canvas objects to create (default: 4)
`defaultPixelFormat`	Default pixel format for video input (default: nv12)
`noVideoSignal`	Settings for when no video signal is detected (default: gray background with "NO SIGNAL" text)
`dshowFilters`	Enable/disable DirectShow filters (default: true)
`v4l2Filters`	Enable/disable V4l2 filters (default: false)
`devMode`	Enable/disable development mode (default: false)

If you choose to use it, place the file in the same directory as the application executable. This file is already included in the agent folder. The configuration file is optional and may be omitted if no custom configuration is required.

NOTE	The configuration file is read only when the agent starts. Any changes to the file will take effect only after restarting the agent.

5.5. Board

The **board** endpoint provides access to information about boards installed in the host system.

Endpoint	Method	Description
/:board	GET	Retrieve information about the board installed in the host system.

URL parameters

- `:board` Device position into the host `[0 .. 1]`.

5.5.1. Board Object

The **board** object provides information about a specific board installed in the host system.

Board object

```
{
    "model" : "lt311",
    "sn" : 71000012,
    "cpu" : 0,
    "fpga" : 0,
    "bridge" : 0
}
```

Attribute	Type	Description
model	string	Board model identifier. Could be `lt311`, `lt312`, `lt313`. If the board model is not recognized, the value is left empty.
sn	uint32	Board serial number.
cpu	uint32	Embedded processing cpu tagged time.
fpga	uint32	Processing fpga tagged time.
bridge	uint32	Bridge fpga tagged time.

5.6. HDMI Input

This endpoint allows you to monitor the **hdmi-in** signal status and access the audio and video streams, when available, in various formats.
Data can be retrieved as raw data or files, in different formats and encodings. Native formats are `yuyv` or `nv12` (video) and `pcm` (audio). Depending on the requested format, the host CPU and/or GPU may be used for processing and conversion.

Endpoint	Method	Description
/:board/hdmi-in/:id	GET	Retrieve the current HDMI Input information.
/:board/hdmi-in/:id/data	POST	Retrieve raw data from the HDMI Input.
/:board/hdmi-in/:id/file	POST	Retrieve a file from the HDMI Input.
/:board/hdmi-in/:id/edid	GET, POST	Retrieve or set the EDID data for the HDMI Input.

URL parameters

- `:board` Device position into the host `[0 .. 1]`.
- `:id` hdmi-in index, dependent on the board type: `not applicable` for LT311, `[0 .. 1]` for LT312, `0` for LT313.

5.6.1. HDMI Input Object

The **hdmi-in** object contains the current state of its audio and video signals.

GET **/:board/hdmi-in/:id**


```
{
    "audio": {
        "description": "",
        "format": "",
        "channels": 0,
        "samplerate": 0,
        "depth": 0,
        "signal": "none"
    },
    "video": {
        "description": "",
        "format": "",
        "size": [0, 0],
        "framerate": 0,
        "interlaced": false,
        "signal": "none"
    }
}
```


Attribute	Type	Description
audio	object	Audio object for the HDMI Input.

Attribute	Type	Description
video	object	Video object for the HDMI Input.

5.6.2. EDID Object

The **edid** object provides the Extended Display Identification Data (EDID) which describe capabilities for a specific hdmi-in. A default EDID is provided, it supports standard resolutions up to 4K 60Hz and audio format 2 channels PCM 48kHz. The EDID can be customized by the user, but make sure it matches the board capabilities.

GET, POST **/:board/hdmi-in/:id/edid**

```
{
    "data": " ... 256-byte ... "
}
```

Attribute	Type	Description
data	[256]byte	The 256-byte E-EDID data.

▼ **POST Example**

These code examples demonstrate how to set custom EDID data for the HDMI Input.

▼ *ecurl*

To set a custom EDID using **ecurl**, the custom EDID file must be provided using the `data=@filename` syntax:

```
$ ecurl post lt310:/0/hdmi-in/0/edid -d data=@customEdid.bin
```

▼ *GO*

```
body := lt.JSON{
    "data": "00FFFFFFFFFF00..." // 256-byte EDID data
}
err := lt.Post("lt310:/0/hdmi-in/0/edid", body, nil)
```

▼ *C++*

```
lt::json body = {
    {"data", "00FFFFFFFFFF00..."} // 256-byte EDID data
};
lt::error err = lt::Post("lt310:/hdmi-in/0/edid", body, nullptr);
```

▼ *Python*

```
edid = {
    "data": "00FFFFFFFFFF00..." # 256-byte EDID data
}
response, err = Post("lt310:/0/hdmi-in/0/edid", edid)
```

5.7. SDI Input

This endpoint allows you to monitor the **sdi-in** signal status and access the audio and video streams, when available, in various formats.
Data can be retrieved as raw data or files, in different formats and encodings. Native formats are `yuyv` or `nv12` (video) and `pcm` (audio). Depending on the requested format, the host CPU and/or GPU may be used for processing and conversion.

Endpoint	Method	Description
/:board/sdi-in/:id	GET	Retrieve the current SDI Input information.
/:board/sdi-in/:id/data	POST	Retrieve raw data from the SDI Input.
/:board/sdi-in/:id/file	POST	Retrieve a file from the SDI Input.

URL parameters

- `:board` Device position into the host `[0 .. 1]`.
- `:id` sdi-in index, dependent on the board type: `[0 .. 3]` for LT311, `not applicable` for LT312, `[0 .. 1]` for LT313.

5.7.1. SDI Input Object

The **sdi-in** object contains the current state of its audio and video signals.

GET **/:board/sdi-in/:id**


```
{
    "audio": {
        "description": "",
        "format": "",
        "channels": 0,
        "samplerate": 0,
        "depth": 0,
        "signal": "none"
    },
    "video": {
        "description": "",
        "format": "",
        "size": [0, 0],
        "framerate": 0,
        "interlaced": false,
        "signal": "none"
    }
}
```


Attribute	Type	Description
audio	object	Audio object for the SDI Input.
video	object	Video object for the SDI Input.

5.8. HDMI Output

Allows to configure the physical hdmi outputs on a LT board.

Endpoint	Method	Description
/:board/hdmi-out/:id	GET, POST	Retrieve or configure the specified hdmi-out.

URL parameters

- `:board` Device position into the host `[0 .. 1]`.
- `:id` hdmi-out index `0`.

5.8.1. HDMI Output Object

The **HDMI Output** object provides configuration and status information about a specific hdmi-out.

GET, POST **/:board/hdmi-out/:id**


```
{
    "source": "auto",
    "overlay": "none",
    "overlayMode": "performance",
    "format": "auto",
    "link": "auto",
    "audio": {
        "description": "",
        "format": "",
        "channels": 0,
        "samplerate": 0,
        "depth": 0,
        "signal": "none"
    },
    "video": {
        "description": "none",
        "format": "",
        "size": [0, 0],
        "framerate": 0,
        "interlaced": false,
        "signal": "none"
    }
}
```


Attribute	Type	Description
source	string	The hdmi-out source. Values can be `auto` or any valid board input source.
overlay	string	The overlay source applied on the hdmi-out. Values can be `canvas/:id` or `none` for no overlay.
overlayMode	string	Controls how overlay content is processed on the output device. This parameter affects the quality and performance of overlays. Values can be `performance` or `quality`.

Attribute	Type	Description
format	string	The color format applied to the output. Values can be `auto`, `rgb444`, `yuv444` and `yuv422`.
link	string	The hdmi-out link carrier. Values can be `auto`, `fhd` or `off`.
audio	*object*	The audio object for the HDMI Output (read only).
video	*object*	The video object for the HDMI Output (read only).

▼ **GET Example**

These code examples demonstrate how to retrieve the configuration and status of hdmi-out 0 on board 0.

▼ *ecurl*

```
$ ecurl get lt310:/0/hdmi-out/0
```

▼ *GO*

```
var response lt.Output // struct to store the response
err := lt.Get("lt310:/0/hdmi-out/0", &response)
```

▼ *C++*

```
lt::Output response; // struct to store the response
lt::error err = lt::Get("lt310:/0/hdmi-out/0", response);
```

▼ *Python*

```
response, err = Get("lt310:/0/hdmi-out/0")
```

▼ **POST Example**

These code examples demonstrate how to enable the overlay with canvas 0 on hdmi-out 0.

▼ *ecurl*

```
$ ecurl post lt310:/0/hdmi-out/0 -d overlay=canvas/0
```

▼ *GO*

```
body := lt.JSON{
    "overlay": "canvas/0",
}
var response lt.Output // struct to store the response
err := lt.Post("lt310:/0/hdmi-out/0", body, &response)
```

▼ *C++*

```
lt::json body = {
    {"overlay", "canvas/0"}
};
lt::Output response; // struct to store the response
lt::error err = lt::Post("lt310:/0/hdmi-out/0", body, response);
```

▼ *Python*

```
body = {
    "overlay": "canvas/0"
}
response, err = Post("lt310:/0/hdmi-out/0", body)
```

5.9. Canvas

The **canvas** endpoint is both a virtual audio/video source and a dynamic synthetic image generator which supports draw operations. It could be used to emulate the LT boards video inputs and to send overlay images onto the hdmi and/or sdi outputs.

Data operations

Endpoint	Method	Description
/canvas/:id	GET	canvas **:id** configuration.
/canvas/:id/data	POST	Data stream
/canvas/:id/file	POST	File recording

Draw operations

Endpoint	Method	Description
/canvas/:id/init	POST	Initialize the canvas.
/canvas/:id/text	POST	Draw text on the canvas.
/canvas/:id/line	POST	Draw a line on the canvas.
/canvas/:id/ellipse	POST	Draw an ellipse on the canvas.
/canvas/:id/rectangle	POST	Draw a rectangle on the canvas.

Endpoint	Method	Description
/canvas/:id/image	POST	Put an image on the canvas.
/canvas/:id/video	POST	Put a video on the canvas.
/canvas/:id/clear	POST	Clear a canvas area or a source.
/canvas/:id/op	POST	Perform a single draw operation on the canvas.
/canvas/:id/ops	POST	Perform multiple draw operations on the canvas.

URL parameters

- `:id` canvas index `[0 .. 3]` (configurable with `numCanvases` in agent configuration file)

5.9.1. Canvas Object

The **Canvas** object provides status information about a specific canvas.

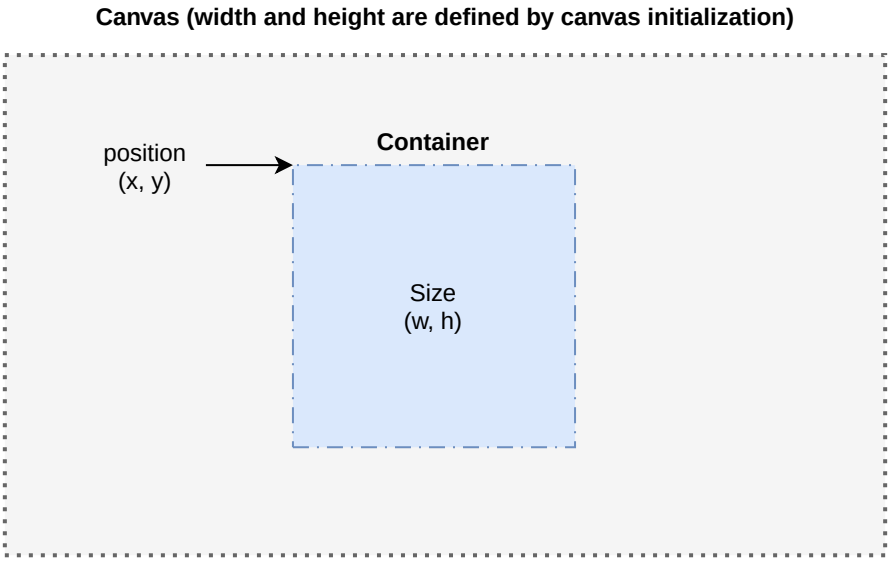
GET **/canvas/:id**


```
{
    "audio": {
        "description": "",
        "format": "",
        "channels": 0,
        "samplerate": 0,
        "depth": 0,
        "signal": "none"
    },
    "video": {
        "description": "",
        "format": "rgba",
        "size": [3840, 2160],
        "framerate": 30,
        "interlaced": false,
        "signal": "locked"
    }
}
```


Attribute	Type	Description
audio	*object*	The audio object for the Canvas (read only).
video	*object*	The video object for the Canvas (read only).

5.9.2. Understanding Canvas Operations

Each drawing operation works within a **container** - a rectangular area that defines where and how the element is drawn. The schema of a container is illustrated below:



All operations share these common parameters:

Parameter	Type	Default	Description
position	[2]int	`[0, 0]`	The top-left corner coordinates `{x, y}` of the container on the canvas.
size	[2]int	varies	The container dimensions `{width, height}` in pixels. **This parameter is mandatory for most operations.**
angle	float	`0`	Rotation angle in degrees, applied around the center of the container.
anchor	[2]int	`[0, 0]`	The pivot point for rotation and scaling transformations, relative to the container's top-left corner.

The mandatory parameters for each operation are highlighted in the respective sections below.

Rendering Order Rules:

- Standard drawing operations (`text`, `line`, `ellipse`, `rectangle`, `image`) are stacked in the order they are sent. Later operations draw over earlier ones.
- Video sources (`video` operations) are always rendered on top of all standard drawing operations.
- Among multiple video sources, the most recently added video appears on top of earlier video sources.

Note: There is no layer system - operations cannot be reordered after being sent.

5.9.3. Initialize Canvas

Canvas initialization is done using the **init** operation. This operation sets up the canvas with the specified background color, size, and framerate. If a file source is provided, the canvas size and framerate are derived from the file content.

POST **/canvas/:id/init**

```
{
    "op": "init",
    "source": "",
    "color": [255,255,255,255],
    "size": [3840,2160],
    "framerate": 30.0
}
```

Attribute	Type	Description
op	string	Operation identifier.
source	string	Path to the file source. Supported formats are `jpeg`, `png`, `bmp` and `mp4` files. If no source is provided, the canvas is filled with the specified color.
color	[4]int	The RGBA background color with transparency. Default `[0,0,0,0]`.
size	[2]int	The video frame width and height in pixel units. Default `[3840,2160]`.
framerate	float	The video frame rate in frames per second. Default `30.0`.

5.9.4. Drawing Operations

5.9.4.1. Text Operation

Draw text onto the canvas with various font styles and attributes. The text can be positioned, rotated and scaled within a defined container.

POST **/canvas/:id/text**

```
{
    "op": "text",
    "text": "hello world!",
    "align": "center",
    "font": "regular",
    "fontSize": 32,
    "italic": false,
    "bold": false,
    "color": [255, 255, 255, 255],
    "angle": 0,
    "position": [0, 0],
    "size": [3840, 2160],
    "anchor": [0, 0]
}
```

Attribute	Type	Description
op	string	Operation identifier.
text	string	Text to draw. **This parameter is mandatory.**

Attribute	Type	Description
align	string	Set the text position into the container. The possible values are `top-left`, `top`, `top-right`, `left`, `center`, `right`, `bottom-left`, `bottom` and `bottom-right`. Default is `center`.
font	string	Font type. Default is `regular`, could also be `mono` and `smallcaps`.
fontSize	int	Font size in pt unit. Default is `32`.
italic	bool	Draw the text with the *italic* attribute. Default `false`.
bold	bool	Draw the text with the **bold** attribute. Default `false`.
color	[4]int	Text color in RGBA format. Default `[0,0,0,255]`.

See Understanding Canvas Operations section for container parameters.

5.9.4.2. Line Operation

Draw a line within the container. The line extends from the top-left corner to the bottom-right corner of the container defined by `position` and `size`.

POST **/canvas/:id/line**

```
{
    "op": "line",
    "width": 1,
    "color": [255, 0, 0, 255],
    "pattern": null,
    "angle": 0,
    "position": [0, 0],
    "size": [3840, 2160],
    "anchor": [0, 0]
}
```

Attribute	Type	Description
op	string	Operation identifier.
width	int	The shape width size in pixel unit. Default `1`.
color	[4]int	The shape RGBA color. Default is `[255,255,255,255]`.
pattern	[]int	The dash size pattern in pixel units. The pattern is repeated. Default no dash pattern: `{}`.

See Understanding Canvas Operations section for container parameters.

5.9.4.3. Ellipse Operation

Draw an ellipse that fits within the container rectangle. The ellipse is inscribed in a bounding box defined by the container's `position` (top-left corner) and `size` (width and height).

POST **/canvas/:id/ellipse**

```
{
    "op": "ellipse",
    "width": 10,
    "color": [255, 0, 0, 255],
    "pattern": null,
    "fill": [0, 255, 0, 255],
    "angle": 0,
    "position": [0, 0],
    "size": [3840, 2160],
    "anchor": [0, 0]
}
```

Attribute	Type	Description
op	string	Operation identifier.
width	int	The shape width size in pixel unit. Default `1`.
color	[4]int	The shape RGBA color. Default is `[255,255,255,255]`.
pattern	[]int	The dash size pattern in pixel units. The pattern is repeated. Default no dash pattern: `{}`.
fill	[4]int	Fill the shape with a RGBA color. Default is `{0,0,0,0}`.

See Understanding Canvas Operations section for container parameters.

5.9.4.4. Rectangle Operation

Draw a rectangle that exactly matches the container boundaries. The rectangle's top-left corner is positioned at `position` coordinates, and its dimensions are defined by `size`.

POST **/canvas/:id/rectangle**

```
{
    "op": "rectangle",
    "width": 1,
    "color": [255, 255, 255, 255],
    "pattern": null,
    "fill": [0, 0, 255, 255],
    "rounded": 0,
    "angle": 0,
    "position": [100, 100],
    "size": [400, 400],
    "anchor": [0, 0]
}
```

Attribute	Type	Description
op	string	Operation identifier.
width	int	The shape width size in pixel unit. Default `1`.
color	[4]int	The shape RGBA color. Default is `[255,255,255,255]`.
pattern	[]int	The dash size pattern in pixel units. The pattern is repeated. Default no dash pattern: `{}`.

Attribute	Type	Description
fill	[4]int	Fill the shape with a RGBA color. Default is `{0,0,0,0}`.
rounded	int	The rectangle corner rounding radius in pixel unit. Default `0`.

See Understanding Canvas Operations section for container parameters.

5.9.4.5. Image Operation

Draw an image within the container. There are two ways to provide the image:

- Using a file path with the `source` parameter. The `format`, `data`, `width` and `height` parameters are ignored as they are derived from the file content.
- Using a data buffer with the `data` parameter. The `format` parameter is mandatory, and for raw formats (`rgba` or `rgb`), the `width` and `height` parameters are also required.

If the `size` parameter is provided, the image is scaled to fit the container while maintaining its aspect ratio. If omitted, the image is drawn at its original dimensions.

POST **/canvas/:id/image**

```
{
    "op": "image",
    "source": "C:\\image.png",
    "format": "",
    "data": null,
    "width": 0,
    "height": 0,
    "angle": 0,
    "position": [0, 0],
    "size": [640, 480],
    "anchor": [0, 0]
}
```

Attribute	Type	Description
op	string	Operation identifier.
source	string	Filepath. Supported formats are `jpeg`, `png` and `bmp` files.
format	string	The image data format. Could be `rgba`, `rgb`, `bmp`, `jpeg` or `png`.
data	[]byte	Image data buffer.
width	int	Image width. Mandatory for `rgba` or `rgb` data buffer.
height	int	Image height. Mandatory for `rgba` or `rgb` data buffer.

See Understanding Canvas Operations section for container parameters.

5.9.4.6. Video Operation

Draw a live video source within the container. The video source can be a board input (SDI/HDMI) or another canvas.

If the `size` parameter is provided, the video is scaled to fit the container while maintaining its aspect ratio. If omitted, the video is drawn at its original dimensions.

Note: The `angle` parameter is not supported for video operations.

POST **/canvas/:id/video**

```
{
    "op": "video",
    "source": "canvas/0",
    "position": [0, 0],
    "size": [1920, 1080],
    "anchor": [0, 0]
}
```

Attribute	Type	Description
op	string	Operation identifier.
source	string	Supported sources are `:board/camera/:id` and `canvas/:id`.

See Understanding Canvas Operations section for container parameters.

5.9.5. Clear Operation

The clear operation can be used for two purposes that can be combined:

1. Remove video sources:

- Specify a `source` parameter to remove a specific video source (`:board/camera/:id` or `canvas/:id`)
- Use `source: "all"` to remove all video sources from the canvas

2. Clear a canvas area:

- Define a rectangular area using `position` and `size` parameters
- The area will be filled with the specified `color` (default: transparent black)
- Use `thickness` to expand the clearing area by a given number of pixels on all sides (acts as padding)

Default behavior: If no parameters are provided, the entire canvas is cleared to transparent black. All drawing operations are removed, but video sources remain untouched.

POST **/canvas/:id/clear**

```
{
    "op": "clear",
    "source": "",
    "color": [0,0,0,0],
    "position": [200,200],
    "size": [512,256],
    "thickness": 0
}
```

Attribute	Type	Description
op	string	Operation identifier.
source	string	Video source to remove. Supported values: `:board/camera/:id`, `canvas/:id`, or `"all"` to remove all video sources. If omitted, no video sources are removed.
color	[4]int	The RGBA background color with transparency. Default `[0,0,0,0]` (transparent black).
position	[2]int	The top-left corner coordinates of the area to clear. If omitted, starts at `[0,0]`.
size	[2]int	Dimensions of the area to clear. If omitted, clears the entire canvas (except video sources).
thickness	int	Expand the clearing area by this number of pixels on all sides (acts as padding). Default `0`.

5.9.6. Single Operation Endpoint

The `op` endpoint is a generic endpoint that accepts any canvas operation (`init`, `text`, `line`, `ellipse`, `rectangle`, `image`, `video`, or `clear`). This provides a unified way to send operations without using operation-specific endpoints.

The operation type is determined by the `op` field in the JSON payload.

POST **/canvas/:id/op**

```
{
    "op": "text",
    "text": "Hello, World!",
    "fontSize": 48,
    "position": [100, 100],
    "size": [800, 100]
}
```

Attribute	Type	Description
op	string	Operation type. Must be one of: `init`, `text`, `line`, `ellipse`, `rectangle`, `image`, `video`, or `clear`. **This parameter is mandatory.**
...	varies	Additional parameters depend on the operation type. See the corresponding operation section for details.

5.9.7. Batch Operations Endpoint

The `ops` endpoint allows sending multiple canvas operations in a single request. Operations are executed sequentially in the order they appear in the array.

This is useful for:

- Atomic updates (all operations succeed or fail together)
- Performance optimization (reduces HTTP overhead)
- Initial canvas setup with multiple elements

Each operation in the array must include an `op` field specifying its type, followed by the operation-specific parameters.

POST **/canvas/:id/ops**

```
{
    "ops": [
        {
            "op": "init",
            "size": [1920, 1080],
            "color": [0, 0, 0, 255]
        },
        {
            "op": "rectangle",
            "position": [100, 100],
            "size": [400, 300],
            "fill": [255, 0, 0, 180]
        },
        {
            "op": "text",
            "text": "Overlay",
            "position": [100, 100],
            "size": [400, 300],
            "align": "center"
        }
    ]
}
```

Attribute	Type	Description
ops	[]object	Array of operation objects. Each object must contain an `op` field and the corresponding parameters for that operation type. **This parameter is mandatory.**

5.9.8. Examples

This section provides practical examples demonstrating common canvas use cases. Additional examples are available in the SDK code samples.

5.9.8.1. Example 1: Simple text banner

This example demonstrates how to create a simple text banner overlay using standard drawing operations. The canvas is assumed to be already initialized with dimensions 1920x1080 and a transparent background.
A semi-transparent dark banner is drawn at the bottom of the screen with white centered text. This is commonly used for displaying information, captions, or alerts over video content.

Step 1: *Draw the banner background using a rectangle with rounded corners and transparency.*

Step 2: *Add centered text on top of the banner.*

POST **lt310:/canvas/0/rectangle**

```
{
    "op": "rectangle",
    "position": [50, 900],
    "size": [1200, 100],
    "fill": [0, 0, 0, 180],
    "rounded": 10
}
```

POST **lt310:/canvas/0/text**

```
{
    "op": "text",
    "text": "Hello, World!",
    "fontSize": 48,
    "color": [255, 255, 255, 255],
    "align": "center",
    "position": [50, 900],
    "size": [1200, 100]
}
```

These operations use the standard drawing endpoints (`/rectangle` and `/text`) which are specific to each operation type.

▼ **Code Examples**

▼ *ecurl*

```
$ ecurl post lt310:/canvas/0/rectangle -d position=50,900 -d size=1200,100 -d fill=0,0,0,180 -d
rounded=10

$ ecurl post lt310:/canvas/0/text -d position=50,900 -d size=1200,100 -d text="Hello, World!" -d
fontSize=48 -d color=255,255,255,255 -d align=center
```

▼ *GO*

```
// Add semi-transparent background bar
body := lt.JSON{
    "position": []int{50, 900},
    "size": []int{1200, 100},
    "fill": []int{0, 0, 0, 180},
    "rounded": 10,
}
err := lt.Post("lt310:/canvas/0/rectangle", body, nil)

// Add text over the background
body = lt.JSON{
    "position": []int{50, 900},
    "size": []int{1200, 100},
    "text": "Hello, World!",
    "fontSize": 48,
    "color": []int{255, 255, 255, 255},
    "align": "center",
}
err = lt.Post("lt310:/canvas/0/text", body, nil)
```

▼ *C++*

```
// Add semi-transparent background bar
lt::json body = {
    {"position", {50, 900}},
    {"size", {1200, 100}},
    {"fill", {0, 0, 0, 180}},
    {"rounded", 10}
};
lt::error err = lt::Post("lt310:/canvas/0/rectangle", body, nullptr);

// Add text over the background
body = {
    {"position", {50, 900}},
    {"size", {1200, 100}},
    {"text", "Hello, World!"},
    {"fontSize", 48},
```

```
    {"color", {255, 255, 255, 255}},
    {"align", "center"}
};
err = lt::Post("lt310:/canvas/0/text", body, nullptr);
```

▼ *Python*

```
# Add semi-transparent background bar
body = {
    "position": [50, 900],
    "size": [1200, 100],
    "fill": [0, 0, 0, 180],
    "rounded": 10
}
resp, err = Post("lt310:/canvas/0/rectangle", body)

# Add text over the background
body = {
    "position": [50, 900],
    "size": [1200, 100],
    "text": "Hello, World!",
    "fontSize": 48,
    "color": [255, 255, 255, 255],
    "align": "center"
}
resp, err = Post("lt310:/canvas/0/text", body)
```

5.9.8.2. Example 2: Side-by-Side SDI Display

Create a 4K canvas displaying two sdi sources side-by-side with labels. This example uses batch operations to set up the entire composition in a single request, including canvas initialization.

Step 1: Initialize a 4K canvas with a black background.

Step 2: Add the first camera video source on the left half of the canvas.

Step 3: Overlay a text label below the first camera feed.

Step 4: Add the second camera video source on the right half of the canvas.

Step 5: Overlay a text label below the second camera feed.

POST **lt310:/canvas/0/ops**

```
{
    "ops": [
        {
            "op": "init",
            "size": [3840, 2160],
            "color": [0, 0, 0, 255],
            "framerate": 30.0
        },
        {
            "op": "video",
            "source": "0/sdi-in/0",
            "position": [0, 600],
            "size": [1920, 1080]
        },
        {
            "op": "text",
            "text": "SDI 0",
            "fontSize": 48,
            "color": [255, 255, 255, 255],
            "align": "center",
            "position": [0, 1680],
            "size": [1920, 100]
        },
        {
            "op": "video",
            "source": "0/sdi-in/1",
            "position": [1920, 600],
            "size": [1920, 1080]
        },
        {
            "op": "text",
            "text": "SDI 1",
            "fontSize": 48,
            "color": [255, 255, 255, 255],
            "align": "center",
            "position": [1920, 1680],
            "size": [1920, 100]
        }
    ]
}
```

Instead of using standard drawing endpoints, all operations are sent via the batch endpoint `/ops` in a single request.

▼ **Code Examples**

▼ *ecurl*

To perform the batch operation using `ecurl`, a JSON file is created containing all the operations as described above, and then sent in a single POST request.

```
$ ecurl post lt310:/canvas/0/ops -d @batch_ops.json
```

▼ *GO*

```
body := lt.JSON{
    "ops": []lt.JSON{
        {
            "op": "init",
            "size": []int{3840, 2160},
            "color": []int{0, 0, 0, 255},
```

```
            "framerate": 30.0,
        },
        {
            "op": "video",
            "source": "0/sdi-in/0",
            "position": []int{0, 600},
            "size": []int{1920, 1080},
        },
        {
            "op": "text",
            "text": "SDI 0",
            "fontSize": 48,
            "color": []int{255, 255, 255, 255},
            "align": "center",
            "position": []int{0, 1680},
            "size": []int{1920, 100},
        },
        {
            "op": "video",
            "source": "0/sdi-in/1",
            "position": []int{1920, 600},
            "size": []int{1920, 1080},
        },
        {
            "op": "text",
            "text": "SDI 1",
            "fontSize": 48,
            "color": []int{255, 255, 255, 255},
            "align": "center",
            "position": []int{1920, 1680},
            "size": []int{1920, 100},
        },
    },
}
err := lt.Post("lt310:/canvas/0/ops", body, nil)
```

▼ *C++*

```
lt::json body = {
    {"ops", {
        {
            {"op", "init"},
            {"size", {3840, 2160}},
            {"color", {0, 0, 0, 255}},
            {"framerate", 30.0}
        },
        {
            {"op", "video"},
            {"source", "0/sdi-in/0"},
            {"position", {0, 600}},
            {"size", {1920, 1080}}
        },
        {
            {"op", "text"},
            {"text", "SDI 0"},
            {"fontSize", 48},
            {"color", {255, 255, 255, 255}},
            {"align", "center"},
            {"position", {0, 1680}},
            {"size", {1920, 100}}
        },
        {
            {"op", "video"},
            {"source", "0/sdi-in/1"},
            {"position", {1920, 600}},
            {"size", {1920, 1080}}
        },
        {
            {"op", "text"},
            {"text", "SDI 1"},
```

```
            {"fontSize", 48},
            {"color", {255, 255, 255, 255}},
            {"align", "center"},
            {"position", {1920, 1680}},
            {"size", {1920, 100}}
        }
    }}
};
lt::error err = lt::Post("lt310:/canvas/0/ops", body, nullptr);
```

▼ *Python*

```
body = {
    "ops": [
        {
            "op": "init",
            "size": [3840, 2160],
            "color": [0, 0, 0, 255],
            "framerate": 30.0
        },
        {
            "op": "video",
            "source": "0/sdi-in/0",
            "position": [0, 600],
            "size": [1920, 1080]
        },
        {
            "op": "text",
            "text": "SDI 0",
            "fontSize": 48,
            "color": [255, 255, 255, 255],
            "align": "center",
            "position": [0, 1680],
            "size": [1920, 100]
        },
        {
            "op": "video",
            "source": "0/sdi-in/1",
            "position": [1920, 600],
            "size": [1920, 1080]
        },
        {
            "op": "text",
            "text": "SDI 1",
            "fontSize": 48,
            "color": [255, 255, 255, 255],
            "align": "center",
            "position": [1920, 1680],
            "size": [1920, 100]
        }
    ]
}
resp, err = lt.Post("lt310:/canvas/0/ops", body)
```

5.10. Workers

All the API processing is based on **Worker objects** created by the multimedia server on behalf of clients requests. A Worker is a software entity that receives, processes, and outputs streams of **Packets**. **Packets** encapsulate video, audio, or control data moving through the system. Workers allow the construction of pipelines that capture data from Enciris boards, process it, or store it in files.

The workers creation endpoints are easily recognizable by their URLs patterns:

Endpoint	Method	Description
/:url/data	POST	Create a data worker for the specified resource.
/:url/file	POST	Create a file worker for the specified resource.

URL parameters

- `:url` URL can be any valid API resource that point toward a `data` or a `file` endpoint.

There are two types of workers available:

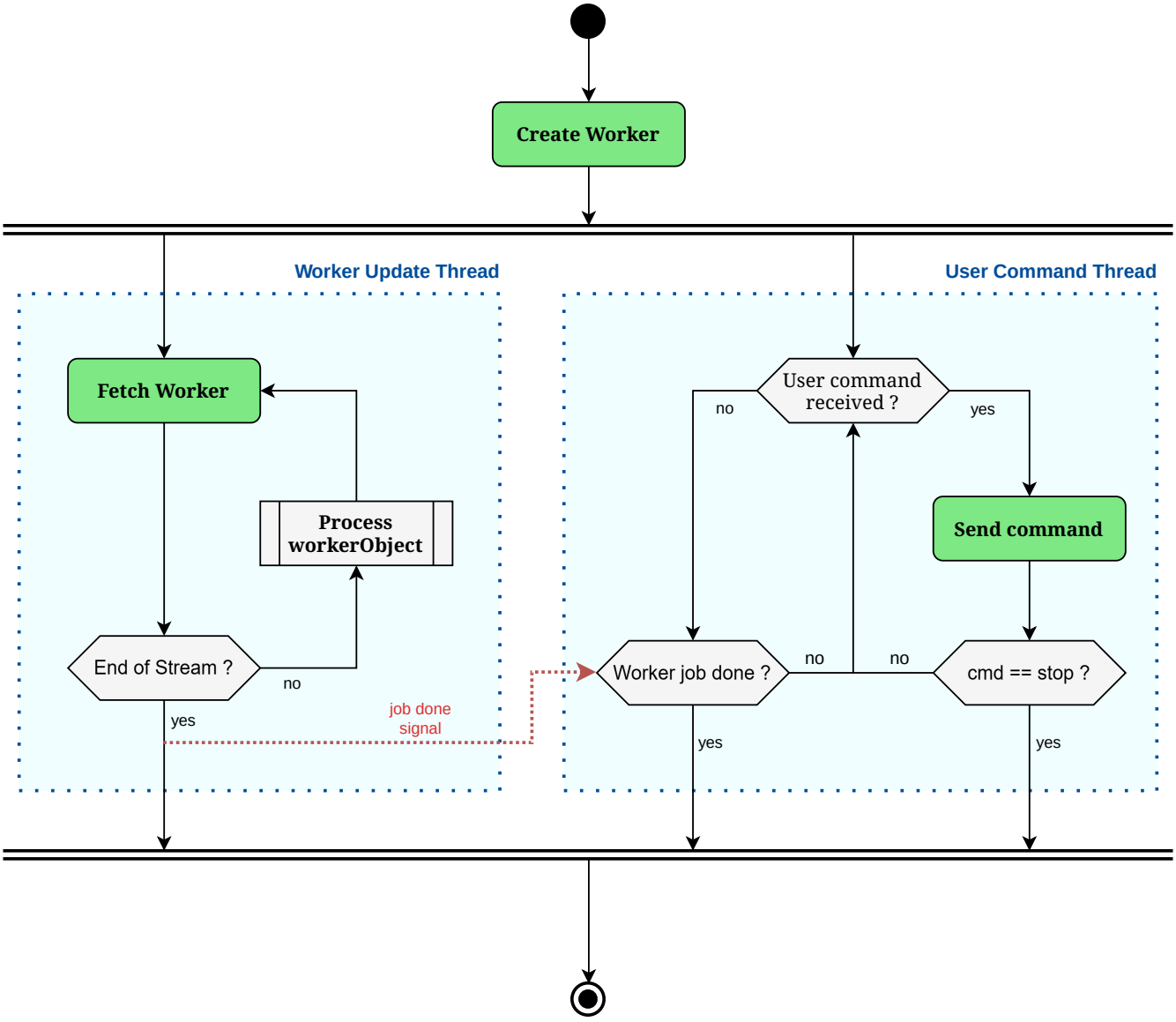
- `data` workers expose media data directly to the client application for real-time analysis or custom processing. The worker object provides packets containing shared memory buffers with raw video or audio data, ensuring minimal latency and zero-copy transfer. Data workers are useful for applications that require direct access to raw frames or audio samples.
- `file` workers record media streams into files on disk, supporting features such as file splitting and multiple container formats. The worker object provides packets containing recording status information instead of raw media buffers. File workers are ideal for applications that need to store streams for later playback or archival.

The general workflow is the same for both worker types:

1. The client creates a worker by sending a POST request to the appropriate endpoint (`/:url/data` or `/:url/file`) with the desired configuration.
2. The API responds with a redirect URL to the newly created worker object.
3. The client follows this redirect to access the worker object, which provides status information and related packets.

The client can also stop the worker at any time by sending a POST request to the `/:url/stop` endpoint.

This workflow is illustrated in the diagram below:



The following sections provide further details on:

- Worker creation and configuration parameters,
- The structure of the worker object,
- The packet and shared packet object models,
- The worker lifecycle and its associated threads,
- Usage examples for common scenarios.

5.10.1. Worker Creation

Workers are created by sending a POST request to the appropriate endpoint (`/:url/data` for data workers or `/:url/file` for file workers) along with the required parameters. If successful, the API responds with a redirect URL that identifies the newly created worker. This URL must be used by the client to fetch updates or to send commands to the worker.

The configuration for each worker type varies based on the media type and processing requirements. The media are separated into distinct categories, each with its own set of parameters

and options.

- Audio workers focus on audio data processing and support parameters like channels, sample rate, and bit depth.
  Available media: `audio/pcm`, `audio/wav`, `audio/aac`.
- Image workers are designed for image data processing and include parameters such as width, height, and pixel format.
  Available media: `image/yuyv`, `image/yuv422`, `image/nv12`, `image/rgba`, `image/rgb`, `image/jpeg`, `image/png`, `image/bmp`.
- Video workers are tailored for video data processing and support parameters like frame rate, resolution, and codec.
  Available media: `video/yuyv`, `video/nv12`, `video/h264`, `video/mp4`.

The following sections detail the parameters and usage for each worker type.

5.10.1.1. Audio Workers

This section describes the parameters used for configuring audio workers. The data worker provides direct access to audio packets for real-time use, while the file worker records audio into a file according to the specified parameters.
The URL used with the POST request must point to an audio source, such as `:board/hdmi-in/:id/data` or `:board/sdi-in/:id/data`.

Common parameters

media `string`

Media type identifier. Supported values: `audio/pcm`, `audio/wav`, `audio/aac`.

channels `int`

Number of audio channels. Default: `2`.

samplerate `int`

Audio sample rate in Hz. Default: `48000`.

depth `int`

Audio sample bit depth. Default: `16`.

Additional file worker parameters

location `string`

The file location to save the recorded audio. Must be a valid file path.

duration `int`

The duration of the recording in seconds. Default: `0` (infinite).

splitSize `int`

The maximum size of each split file in bytes. Default: `0` (unlimited, no splitting).

splitDuration `int`

The duration of each split file in seconds. Default: `0` (unlimited, no splitting).

Response

Returns the location of the worker object onto the form of a **redirect** error.

Data Worker Creation

File Worker Creation

POST /:url/data

request

```
{
    "method": "POST",
    "url": "lt310:/:url/data",
    "body": {
        "media": "audio/pcm",
        "channels": 2,
        "samplerate": 48000,
        "depth": 16
    }
}
```

response

```
{
    "location": "lt310:/client/jobs/...",
    "error": "redirect"
}
```

POST /:url/file

request

```
{
    "method": "POST",
    "url": "lt310:/:url/file",
    "body": {
        "media": "audio/pcm",
        "channels": 2,
        "samplerate": 48000,
        "depth": 16,
        "location": "path_to_directory",
        "duration": 0,
        "splitSize": 0,
        "splitDuration": 0
    }
}
```

response

```
{
    "location": "lt310:/client/jobs/...",
    "error": "redirect"
}
```

▼ **Examples**

▼ *GO*

```go
// Create Data worker
err := lt.Post("lt310:/:url/data", lt.AudioDataWorker{Media: "audio/pcm"}, nil)

// Create File worker
// err := lt.Post("lt310:/:url/file", lt.AudioFileWorker{Media: "audio/wav"}, nil)

if !errors.Is(err, lt.ErrRedirect) {
    log.Fatal("worker creation failed:", err)
}
workerURL := lt.RedirectLocation(err)
```

▼ *C++*

```cpp
// Create Data worker
lt::error err = lt::Post("lt310:/:url/data", lt::AudioDataWorker{ "audio/pcm" }, nullptr);

// Create File worker
// err := lt.Post("lt310:/:url/file", lt.AudioFileWorker{Media: "audio/wav"}, nil);

if (!lt::ErrorIs(err, lt::ErrRedirect)) {
    logFatal("worker creation failed:" + err);
}
string workerURL = lt::RedirectLocation(err);
```

▼ *Python*

```python
# Create Data worker
resp, err = Post("lt310:/:url/data", {'media': "audio/pcm"})

# Create File worker
```

```
# err := lt.Post("lt310:/:url/file", lt.AudioFileWorker{Media: "audio/wav"}, nil);

if not lt.ErrorIs(err, lt.ErrRedirect):
    exit(err)
workerURL = lt.RedirectLocation(err)
```

5.10.1.2. Image Workers

This section describes the parameters used for configuring image workers. The data worker provides direct access to image packets, while the file worker records images into a file according to the specified parameters.
The URL used with the POST request must point to an image source, such as `:board/hdmi-in/:id` `:board/sdi-in/:id` or `canvas/:id`.

Common parameters

media `string`

Media type identifier. Could be `image/yuyv`, `image/yuv422`, `image/nv12`, `image/rgba`, `image/rgb`, `image/jpeg`, `image/png` and `image/bmp`.

size `[2]int`

The image frame size. Let empty to use the default size.

Additional file worker parameters

location `string`

The file location to save the recorded audio. Must be a valid file path.

Response

Returns the location of the worker object onto the form of a **redirect** error.

Data Worker Creation

File Worker Creation

POST /:url/data

request

```
{
    "method": "POST",
    "url": "lt310:/:url/data",
    "body": {
        "media": "image/jpeg",
        "size": [1920, 1080]
    }
}
```

response

```
{
    "location": "lt310:/client/jobs/...",
    "error": "redirect"
}
```

POST /:url/file

request

```
{
    "method": "POST",
    "url": "lt310:/:url/file",
    "body": {
        "media": "image/jpeg",
        "size": [1920, 1080],
        "location": "path_to_directory"
    }
}
```

response

```
{
    "location": "lt310:/client/jobs/...",
    "error": "redirect"
}
```

▼ **Examples**

▼ *GO*

```go
// Create Data worker
err := lt.Post("lt310:/:url/data", lt.ImageDataWorker{Media: "image/jpeg"}, nil)

// Create File worker
// err := lt.Post("lt310:/:url/file", lt.ImageFileWorker{Media: "image/jpeg"}, nil)

if !errors.Is(err, lt.ErrRedirect) {
    log.Fatal("worker creation failed:", err)
}
workerURL := lt.RedirectLocation(err)
```

▼ *C++*

```cpp
// Create Data worker
lt::error err = lt::Post("lt310:/:url/data", lt::ImageDataWorker{ "image/jpeg" }, nullptr);

// Create File worker
// lt::error err = lt::Post("lt310:/:url/file", lt::ImageFileWorker{ "image/jpeg" }, nullptr);

if (!lt::ErrorIs(err, lt::ErrRedirect)) {
    logFatal("worker creation failed:" + err);
}
string workerURL = lt::RedirectLocation(err);
```

▼ *Python*

```python
# Create Data worker
resp, err = Post("lt310:/:url/data", {'media': "image/jpeg"})

# Create File worker
// resp, err = Post("lt310:/:url/file", {'media': "image/jpeg"})

if not lt.ErrorIs(err, lt.ErrRedirect):
    exit(err)
workerURL = lt.RedirectLocation(err)
```

5.10.1.3. Video Workers

This section describes the parameters used for configuring video workers. The data worker provides direct access to video packets for real-time use, while the file worker records video into a file according to the specified parameters.
The URL used with the POST request must point to a video source, such as `:board/hdmi-in/:id`, `:board/sdi-in/:id` or `canvas/:id`.

Common parameters

media `string`

Media type identifier. Could be `video/yuyv`, `video/nv12`, `video/h264`, `video/mp4`.

size `[2]int`

The image frame size. Let empty to use the default size.

framerate `float`

The video frame rate. Let empty to use the default framerate.

Additional file worker parameters

location `string`

The file location to save the recorded video. Must be a valid file path.

duration `int`

The duration of the recording in seconds. Default `0` (infinite).

splitSize `int`

The maximum size of each split file in bytes. Default: `0` (unlimited, no splitting).

splitDuration `int`

The duration of each split file in seconds. Default: `0` (unlimited, no splitting).

Extra `string`

Video encoder configuration parameters that control the encoding process:

- `hw` - Hardware encoder to use: `"qsv"` (Intel), `"nvenc"` (NVIDIA), or `"amf"` (AMD)
- `bitrate` - Target bitrate in bits per second (e.g., 5000000 for 5 Mbps)
- `quality` - Quality/compression level (19-24, lower values = higher quality)
- `gop` - Group of Pictures - keyframe interval in frames
- `codec` - Video codec to use: `"h264"` or `"hevc"`
- `preset` - Preset for the encoder (e.g., `"veryfast"`, `"faster"`, `"fast"`, `"medium"`, `"slow"`, `"slower"`, `"veryslow"`)

> **NOTE** Use either `bitrate` or `quality` for rate control, but not both simultaneously. Using `bitrate` creates a constant bitrate encoding, while `quality` creates variable bitrate encoding with consistent visual quality.

Response

Returns the location of the worker object onto the form of a **redirect** error.

Data Worker Creation **File Worker Creation**

POST /:url/data

request

```
{
    "method": "POST",
    "url": "lt310:/:url/data",
    "body": {
        "media": "video/nv12",
        "size": [1920, 1080],
        "framerate": 30
    }
}
```

response

```
{
    "location": "lt310:/client/jobs/...",
    "error": "redirect"
}
```

POST /:url/file

request

```
{
    "method": "POST",
    "url": "lt310:/:url/file",
    "body": {
        "media": "video/mp4",
        "size": [1920, 1080],
        "framerate": 30,
        "location": "path_to_directory",
        "duration": 0,
        "splitSize": 0,
        "splitDuration": 0,
        "extra": {
            "hw": "",
            "bitrate": 0,
            "quality": 0,
            "gop": 0,
            "codec": "",
            "preset": ""
        }
    }
}
```

response

```
{
    "location": "lt310:/client/jobs/...",
    "error": "redirect"
}
```

▼ **Examples**

▼ *GO*

```go
// Create Data worker
err := lt.Post("lt310:/:url/data", lt.VideoDataWorker{Media: "video/nv12"}, nil)

// Create File worker
// err := lt.Post("lt310:/:url/nv12", lt.VideoFileWorker{Media: "video/nv12"}, nil)

if !errors.Is(err, lt.ErrRedirect) {
    log.Fatal("worker creation failed:", err)
}
workerURL := lt.RedirectLocation(err)
```

▼ *C++*

```cpp
// Create Data worker
lt::error err = lt::Post("lt310:/:url/data", lt::VideoDataWorker{ "video/nv12" }, nullptr);

// Create File worker
// lt::error err = lt::Post("lt310:/:url/file", lt::VideoFileWorker{ "video/nv12" }, nullptr);

if (!lt::ErrorIs(err, lt::ErrRedirect)) {
    logFatal("worker creation failed:" + err);
}
string workerURL = lt::RedirectLocation(err);
```

▼ *Python*

```python
# Create Data worker
resp, err = Post("lt310:/:url/data", {'media': "video/nv12"})

# Create File worker
// resp, err = Post("lt310:/:url/file", {'media': "video/nv12"})

if not lt.ErrorIs(err, lt.ErrRedirect):
    exit(err)
workerURL = lt.RedirectLocation(err)
```

5.10.2. Worker Object

The Worker object is the result of a GET request onto a worker endpoint. It contains the worker status, data packets and metadata. A Worker might process one or multiples tracks and the SDK provides helpers functions to automatically parse the worker into a comprehensive structure with the contained audio and video packets.

name `string`

In case of file workers, the name of the output file. Otherwise, it is empty.

location `string`

In case of file workers, the location of the output file. Otherwise, it is empty.

start `int64`

Unix timestamp at which the worker started.

duration `int64`

Elapsed time since the worker started.

size `int`

Quantity of byte processed since the segment started.

status `string`

Worker current status (`running`, `paused` or `completed`).

packets `map[int]packet`

Packets maps packet or shared packet of video, audio or text data samples and/or metadata samples.

Worker object

```
{
    "name": "",
    "location": "",
    "start": 1644248369455566,
    "duration": 16667,
    "size": 4147200,
    "status": "completed",
    "packets": {
        "0": {
            "... packet object #0 ..."
        }
    }
}
```

5.10.3. Packet and SharedPacket Objects

Packets are the fundamental units delivered by a worker. They may contain raw media data (PCM audio samples, raw video frames), or status information (e.g., file recording progress).
Two packet models exist:

- **Packet**: contains metadata and inline media data.
- **SharedPacket**: references a buffer in shared memory for zero-copy access to larger media payloads.

Clients must release packets after processing them to free system resources.

Common parameters

track `int`

The track ID of the packet if the worker process multiple tracks.

type `string`

The packet type and format.

signal `string`

`none` (not found), or `locked` (ready to use)

timestamp `int64`

Unix timestamp at which the packet has been sampled.

meta `JSON`

The metadata fields for audio and video. See audio and video metadata objects.

Packet object specific parameters

data `[]byte`

The packet plain data buffer.

SharedPacket object specific parameters

ref `string`

The shared memory reference to be deleted once the data has been used.

client `string`

The client id which has made the request.

handle `string`

The handle that allows to access the shared memory.

size `int`

The shared memory block total capacity.

len `int`

The shared buffer length inside the shared memory block.

Packet object

```
{
    "track": 0,
    "type": "video/nv12",
    "signal": "locked",
    "timestamp": 1695815814430318,
    "meta": {
        "size": [1920, 1080],
        "framerate": 30,
        "interlaced": false,
        "keyframe": true
    },
    "data": "bytes_array",
}
```

SharedPacket object

```
{
    "track": 0,
    "type": "video/nv12",
    "signal": "locked",
    "timestamp": 1695815814430318,
    "meta": {
        "size": [1920, 1080],
        "framerate": 30,
        "interlaced": false,
        "keyframe": true
    },
    "len": 16588800,
    "ref": "{lt}:/client/ref/...",
    "client": "client_id",
    "handle": "shared_memory_handle",
    "size": 1275592704,
}
```

To simplify integration, the SDK provides a helper function that automatically parses each packet

type into a unified structure, making it easier for developers to work with the data. This approach streamlines packet handling and ensures consistency across different worker types and media formats. The new packet object model maintains common parameters while standardizing how the data sections are represented, as described below.

data `[]byte`

If the packet is a Packet object, this field contains the packet data. If the packet is a SharedPacket object, the shared memory content is loaded into this field.

ref `string`

If the packet is a Packet object, this field is empty. If the packet is a SharedPacket object, this field contains the shared memory reference, which indicates to the user that the data is loaded from shared memory.

SDK Packet Object

```
{
    "track": 0,
    "type": "video/nv12",
    "signal": "locked",
    "timestamp": 1695815814430318,
    "meta": {
        "size": [1920, 1080],
        "framerate": 30,
        "interlaced": false,
        "keyframe": true
    },
    "ref": "{lt}:/client/ref/...",
    "client": "q5jrzd2OIQuxCq1IJWICuA",
    "handle": "{lt}_global_24",
}
```

5.10.3.1. Metadata

The content of metadata structure present in packet object or shared packet object depends on the packet type. It exposes some fields that are specific to the type of media being processed. The following sections describe the fields for audio, image and video metadata.

Audio Metadata

channels `int`

The number of channels.

samplerate `int`

The number of samples per second.

depth `int`

The number of bits per sample.

Samples `int`

The number of samples contained into the buffer.

Audio metadata

```
{
    "channels": 2,
    "samplerate": 48000,
    "depth": 16,
    "samples": 800,
}
```

Image Metadata

size `[2]int`

The image frame size.

Image metadata

```
{
    "size": [1920, 1080],
}
```

Video Metadata

size `[2]int`

The video frame size.

framerate `float`

The number of video frame per second.

interlaced `bool`

Is the frame interlaced.

keyframe `bool`

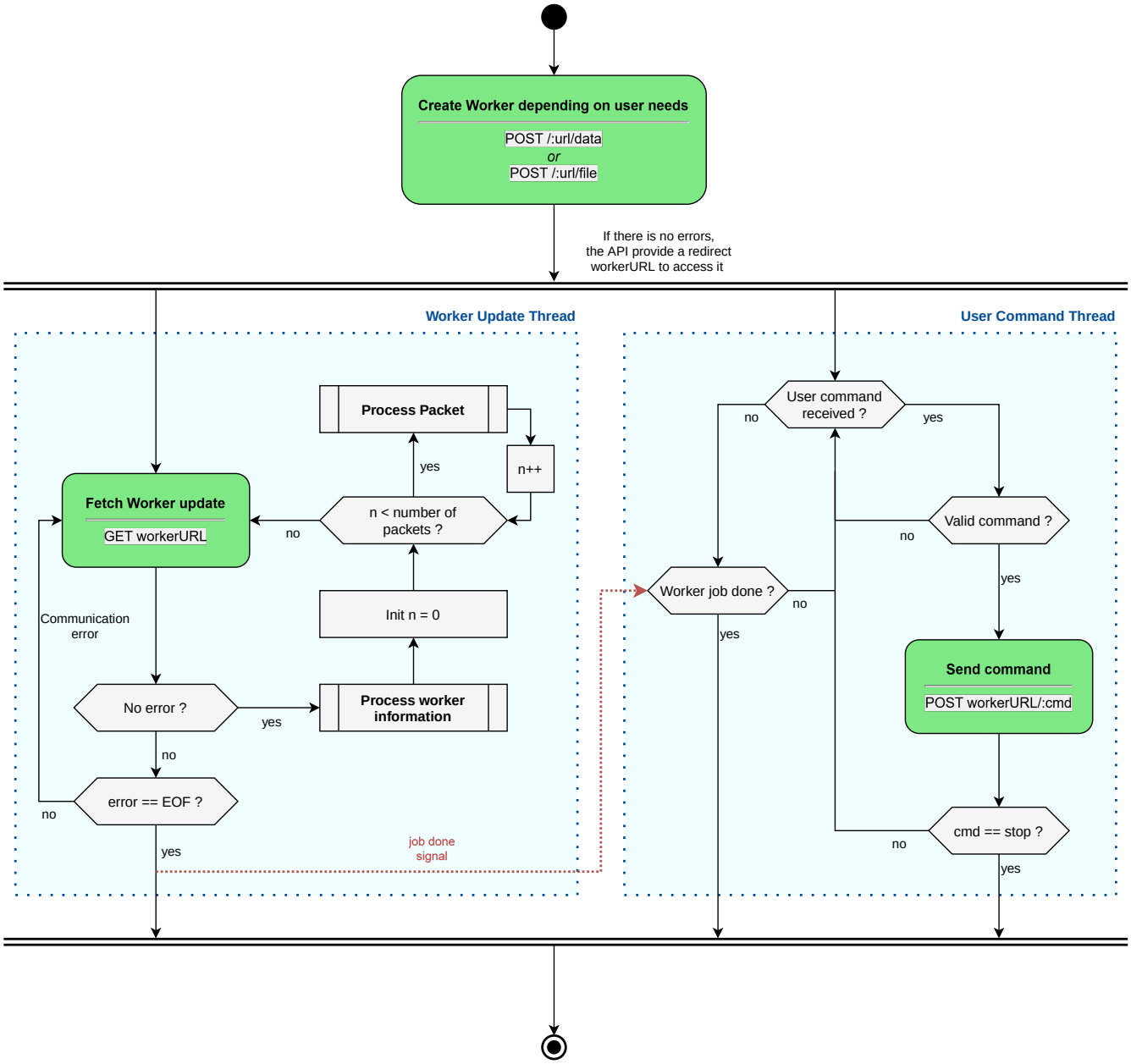
Is the frame intra coded.

Video metadata

```
{
    "size": [1920, 1080],
    "framerate": 60,
    "interlaced": false,
    "keyframe": true
}
```

5.10.4. Worker lifecycle

The worker lifecycle consists of three main stages: creation, processing, and termination. The following diagram provides a detailed view of this workflow:



The first step is to create the worker by sending a POST request to the appropriate endpoint. If the request is successful, the response contains the location of the created worker (`workerURL`), which can be used to interact with the worker during its lifecycle.

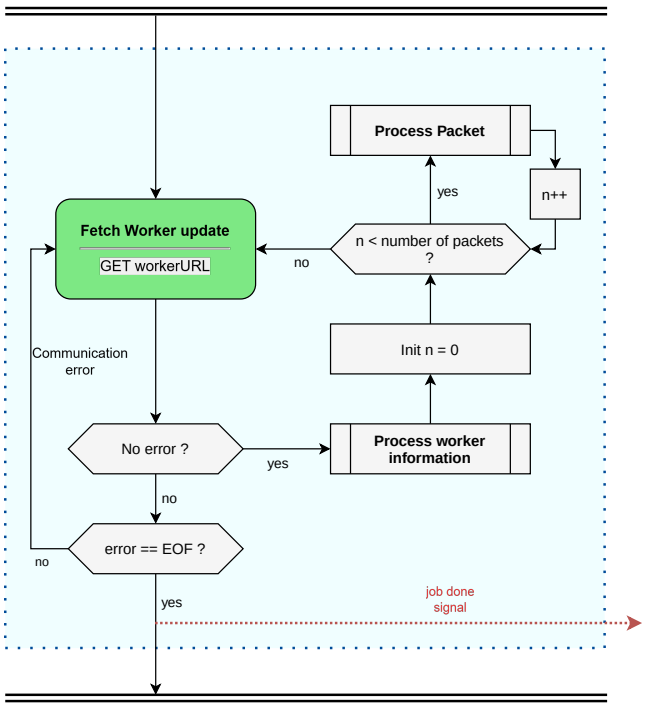
To manage this interaction, two client-side threads are typically used:

- **Worker Update Thread**: continuously fetches the worker object from the server, updating its status and packets. The loop continues until an `EOF` error is returned, indicating that the worker has completed its task.
- **User Command Thread (optional)**: processes user commands and forwards them to the worker. Typical commands include `stop`, `pause`, and `start`. Sending such a command will eventually lead to an `EOF` being returned to the update thread, signaling the end of the worker.

5.10.4.1. Worker Update Thread

This thread is responsible for regularly fetching the worker object from the server, processing the returned packets and worker status, and detecting the EndOfStream (`EOF`) condition. It usually runs concurrently with the main application logic.

Step by step:



1. `Fetch worker update`

   Perform a `GET` request on the `workerURL` to retrieve the latest worker object.

2. `Check for errors`

   - If `EOF` is returned by the server, signal job completion and exit.
   - If another error occurs, handle it appropriately (log, retry, etc.).
   - Otherwise, parse the worker object.

3. `Process worker information`

   Extract status, progress, and available data packets.
   Note: A `completed` status may indicate a finished file segment depending on the worker type.

4. `Process worker packets`

   Loop over packets and process each according to its type.
   Tip: Process asynchronously to avoid blocking the update loop. Release each packet after use with `Close()`.

5. `Repeat`

   Return to step 1 until the server returns `EOF`.

5.10.4.2. User Command Thread

This thread handles user commands and sends them to the worker. It is optional and may not be needed in all applications. Supported commands are: `stop`, `pause`, and `start`.

Step by step:

1. `Wait for command`

   Listen non-blocking for `stop`, `pause`, or `start`.

2. `Verify and send`

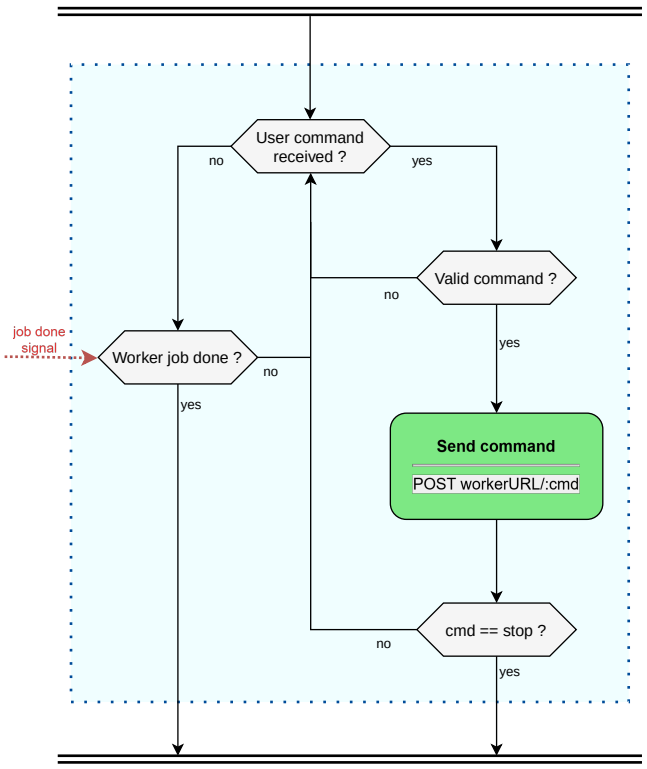
   Ensure the command is valid and POST it to `workerURL`.

3. `Check for completion`

   Exit the loop if the `stop` command is sent, or if a job-done signal is received from the Worker Update Thread.

4. `Repeat`

   Return to step 1 until termination.



5.10.5. Example 1: fetching audio data with a data worker

This example demonstrates how to create an audio data worker, fetch audio packets, and process them using the SDK. The audio data can then be played back or analyzed as needed.
Only the Worker Update Thread is used in this example, since the User Command Thread is optional.

Creating an audio data worker
Send the following request to create a data worker for fetching audio from `0/hdmi-in/0` input:

request

```
{
    "method": "POST",
    "url": "lt310:/0/hdmi-in/0/data",
    "body": {
        "media": "audio/pcm",
        "source": "0/hdmi-in/0",
        "channels": 2,
        "samplerate": 48000,
        "depth": 16
    }
}
```

If successful, the server responds with a redirect containing the worker location (`workerURL`):

response (redirect)

```
redirect: lt310:/client/jobs/VCW90ecK90GwE
```

The user can now start the Worker Update Thread to fetch and process audio packets.

Fetching worker updates
To fetch updates, send a GET request to the worker URL:

request

```
{
    "method": "GET",
    "url": "lt310:/client/jobs/VCW90ecK90GwE"
}
```

Possible outcomes:

- `null`: request successful, worker object returned.
- `EOF`: special error returned by the server indicating EndOfStream - no more packets will be produced.
- Any other value: an error occurred while fetching the update.

Processing the worker object

A successful response contains the worker object, including status, packets, and metadata.
Example parsed by the SDK:

Worker Object with SDK Packet Object


```
{
    "name": "",
    "location": "",
    "duration": 595015,
    "length": 114240,
    "start": 1758272241958140,
    "status": "running",
    "packets": [
    {
        "Track": 0,
        "Media": "audio/pcm",
        "Signal": "locked",
        "Timestamp": 1758616846077989,
        "Ref": "",
        "Data": "AAAAAAAAA...",
        "Meta": {
            "channels": 2,
            "samplerate": 48000,
            "depth": 16,
            "samples": 1020
        }
    }
    ]
}
```


Notes:

- `"name"` and `"location"` are empty (data worker).
- Each packet contains audio data and metadata (channels, sample rate, bit depth, sample count).
- `"ref"` is empty because this is a `Packet` object, not a `SharedPacket`.
- `"data"` contains the raw audio samples in bytes.

Release each packet with `Close()` to free resources. Loop back to fetch the next update until the server returns an `EOF` error.

End of Stream handling

A data worker runs indefinitely until explicitly stopped. To stop it, the client should:

- Send a `stop` command via the User Command Thread (recommended).
- Or terminate the application directly (not recommended, may leave resources inconsistent).

After receiving `stop`, the worker finishes processing any remaining data. The client must continue fetching updates until the server returns an `EOF` error, which signals that the worker has terminated.

5.10.6. Example 2: recording a video stream with a file worker

This example demonstrates how to create a video file worker to record a video stream from an input source and process worker updates.
Only the Worker Update Thread is implemented in this example, as the User Command Thread is optional.

Creating a video file worker
Send the following request to create a file worker for recording video from `0/sdi-in/0` input with this configuration:

- recording format: `video/mp4`
- save files to `C:\Users\A\Videos`
- record 10 seconds, splitting every 5 seconds
- other parameters are left to default values.

request

```
{
    "Method": "POST",
    "URL": "lt310:/0/sdi-in/0/file",
    "Body": {
        "media": "video/mp4",
        "location": "C:\\Users\\A\\Videos",
        "duration": 10,
        "splitSize": 0,
        "splitDuration": 5,
        "size": [
            0,
            0
        ],
        "framerate": 0,
        "extra": {
            "hw": "",
            "bitrate": 0,
            "quality": 0,
            "gop": 0,
            "codec": "",
            "preset": ""
        }
    }
}
```

If successful, the server responds with a redirect containing the worker location (`workerURL`):

response (redirect)

```
redirect: lt310:/client/jobs/NBCgXxNEOsc
```

Recording starts immediately. The client can now start the Worker Update Thread to fetch and process packets.

Fetching worker updates
To fetch updates, send a GET request to the worker URL:

request

```
{
    "method": "GET",
    "url": "lt310:/client/jobs/NBCgXxNEOsc"
}
```

Possible outcomes:

- `null`: request successful, worker object returned.
- `EOF`: special error returned by the server indicating EndOfStream - no more packets will be

produced.

- Any other value: an error occurred while fetching the update.

Processing the worker object

Worker Object with SDK Packet Object


```
{
    "name": "VID_20250919_172235_449.mp4",
    "location": "C:\\Users\\A\\Videos",
    "start": 1758295355449982,
    "duration": 640004,
    "length": 555098,
    "status": "running",
    "packets": [
    {
        "Track": 0,
        "Media": "video/h264",
        "Signal": "locked",
        "Timestamp": 1758295356049986,
        "Ref": "",
        "Data": null,
        "Meta": {
            "size": [
                1920,
                1080
            ],
            "framerate": 60,
            "interlaced": false,
            "keyframe": false
        }
    }
    ]
}
```


Notes:

- `"name"` and `"location"` indicate the name and location of the recorded file. When a split occurs, the name will change to reflect the new file (e.g., `VID_20250919_172235_450.mp4`).
- `"status"` shows `running` while recording. It changes to `completed` when the recording duration is reached or just before a split.
- Each packet (e.g., track ID `"0"`) contains video metadata and encoding information such as frame size, framerate, interlacing status, and keyframe status.
- `"ref"` and `"data"` are empty because the `FileWorker` provides metadata only, not direct video data.

Release each packet with `Close()` and loop back to fetch the next update until the server returns `EOF`.

End of Stream handling

This file worker runs for a specified duration and automatically completes the recording.
The client should continue fetching updates until the server returns an `EOF` error, indicating that no more packets will be produced and the worker is terminated. On the final worker object, `"status"` is typically `completed`, and `"duration"` reflects the total recording time.

Chapter 6. Cheatsheet

Agent

Endpoint	Method	Description
/	GET	Retrieve the current agent information.

Board

Endpoint	Method	Description
/:board	GET	Retrieve information about the board installed in the host system.

HDMI Input

Endpoint	Method	Description
/:board/hdmi-in/:id	GET	Retrieve the current HDMI Input information.
/:board/hdmi-in/:id/data	POST	Retrieve raw data from the HDMI Input.
/:board/hdmi-in/:id/file	POST	Retrieve a file from the HDMI Input.
/:board/hdmi-in/:id/edid	GET, POST	Retrieve or set the EDID data for the HDMI Input.

SDI Input

Endpoint	Method	Description
/:board/sdi-in/:id	GET	Retrieve the current SDI Input information.
/:board/sdi-in/:id/data	POST	Retrieve raw data from the SDI Input.
/:board/sdi-in/:id/file	POST	Retrieve a file from the SDI Input.

HDMI Output

Endpoint	Method	Description
/:board/hdmi-out/:id	GET, POST	Retrieve or configure the specified hdmi-out.

Canvas

Endpoint	Method	Description
/canvas/:id	GET	canvas **:id** configuration.
/canvas/:id/data	POST	Data stream
/canvas/:id/file	POST	File recording

Endpoint	Method	Description
/canvas/:id/init	POST	Initialize the canvas.
/canvas/:id/text	POST	Draw text on the canvas.
/canvas/:id/line	POST	Draw a line on the canvas.
/canvas/:id/ellipse	POST	Draw an ellipse on the canvas.
/canvas/:id/rectangle	POST	Draw a rectangle on the canvas.
/canvas/:id/image	POST	Put an image on the canvas.
/canvas/:id/video	POST	Put a video on the canvas.
/canvas/:id/clear	POST	Clear a canvas area or a source.
/canvas/:id/op	POST	Perform a single draw operation on the canvas.
/canvas/:id/ops	POST	Perform multiple draw operations on the canvas.

Chapter 7. Changelog

```
# 1.4.0 (28/11/2025):
- Add NV12 native support
- Add audio recording
- Add 2K DCI resolution support
- Improve audio acquisition
- Improve QoS
- Improve HDMI input
- Improve SDK clients & examples
- Update documentation

# 1.3.1 (25/08/2025):
- Add audio/video player example (Go & C++)
- Improve audio fetching

# 1.3.0 (29/04/2025):
- Add HEVC codec options
- Add interlace support
- Add python client
- Add video encoder parameters
- Improve agent reliability
- Improve canvas
- Improve HDMI output
- Improve overlay performance
- Minor fixes
- Hardware acceleration on Linux currently works only with Intel QSV (Quick Sync Video)

# 1.2.0 (18/12/2024):
- Improve DirectShow filters
- Update sdk
- Change boards EDID
- Minor fixes

# 1.1.0 (27/09/2024):
- Add DirectShow audio

# 1.0.0 (10/07/2024):
- First official release
```