# LT300 API

## *The programmer guide*

Enciris Technologies

Version 1.5.0, 24/04/2026

# Table of Contents

# Chapter 1. Introduction

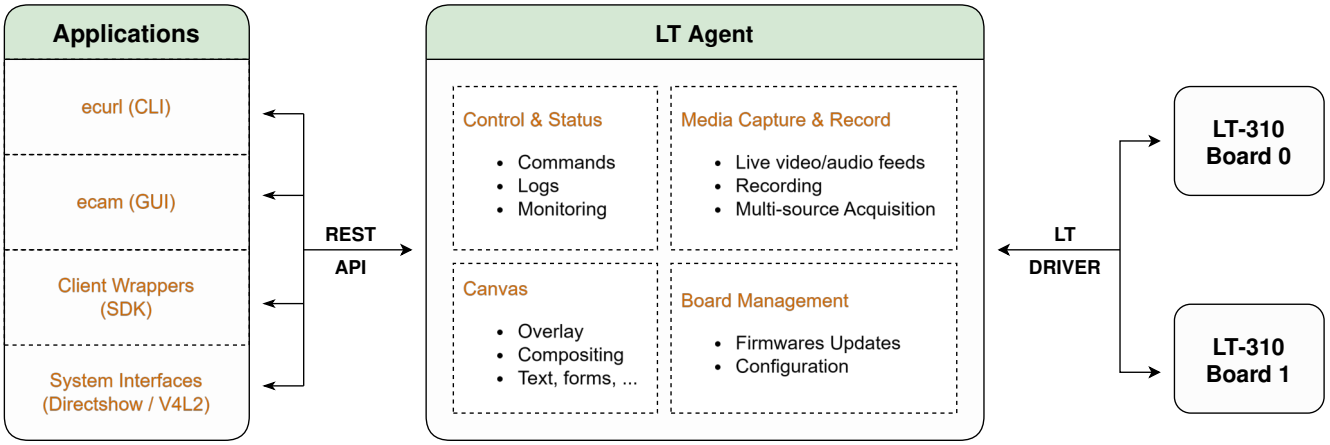
The **LT API** is built around the REST architecture (Representational State Transfer). REST defines a structured approach for exposing system functionalities via a consistent interface.

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 **lt300agent**, 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 Go, C++, C#, Python, DirectShow, V4L2, NamedPipe, and UART.



# Chapter 2. Installation

Two downloadable packages are available for the **LT300** family boards:

- Installer - installs the runtime environment. It is required to operate the boards and access the API.
- SDK - optional, provides development resources such as API documentation, client libraries, and programming examples.

Both packages can be downloaded from the product page.

## 2.1. Installer

The installer provides a complete runtime environment for LT300 boards. It installs the required drivers, deploys the **lt300agent** service for board management and API access, and provides two user tools: ecurl (command-line interface) and ecap (GUI).

Once installed, the system is automatically configured for immediate use. The **lt300agent** service starts at boot, the command-line tools are made available through the system PATH, and the boards are exposed through standard interfaces (DirectShow on Windows and V4L2 on Linux).

The **lt300agent** service will check the board firmware and perform an automatic update if needed. This process may take a few minutes and the board should not be powered off during the update.

The **lt300agent** service can be controlled from the command line. Refer to lt300agent Service for more information.

If you encounter any issues during installation or update, please contact our support team for assistance.

### 2.1.1. Windows

**Supported versions:** Windows 10 and later.

Download and run the latest `lt300install_x.x.x.exe` from our website.

Administrator privileges are required; the installer will prompt for elevation if needed. If a previous version is installed, it will be automatically removed.

The installed boards should appear in Device Manager under **Sound, video and game controllers**.

**Directshow**

All board inputs are accessible as DirectShow filters, available in any DirectShow-compatible application.
The filters are managed by the **lt300agent** service and require the service to be running.

| NOTE | To uninstall, run the installer and choose "Remove" or use "Apps & features" menu. |
|---|---|

### 2.1.2. Linux

**Supported distributions:** Ubuntu 20.04 or later, Debian 12 or later.

Download and extract the latest **lt300install_x.x.x.tar.gz** from our website, then run:

```
./lt300install.sh
```

Elevated privileges are required and handled automatically by the installer. If a previous version is installed, it will be automatically removed.

**V4L2**

All board inputs are accessible through V4L2 drivers, available in any V4L2/GStreamer-compatible application. The V4L2 drivers are managed by the **lt300agent** service and require the service to be running.

| NOTE | To uninstall, run **lt300_uninstall.sh** from the installation directory. |
|---|---|

## 2.2. SDK

The LT300 SDK provides development resources for building applications that interact with the LT API.
Download one of the following archives from our website:

```
lt300sdk_x.x.x.zip
lt300sdk_x.x.x.tar.gz
```

Extract the archive to any directory. The SDK includes:

- API documentation
- client libraries, providing a ready-to-use layer on top of the REST API
- complete source code examples

Currently supported languages include:

- Go
- C++
- C#
- Python

The examples are described in the SDK Examples chapter and can be used as a starting point for

your own applications.

# Chapter 3. lt300agent Service

The **lt300agent** service is the core component of the **LT300** software platform.
It manages the **LT300** boards and exposes the **LT API** used by applications and tools.

The service must be running to:

- access the **LT API**
- expose board inputs through DirectShow or V4L2
- control devices using tools such as ecurl or ecap

## 3.1. Configuration File

The **lt300agent** can be configured using an optional configuration file named `lt300agent.cfg`. Configuration options:

| Option | Default | Description |
|---|---|---|
| `wd` | *(current directory)* | Working directory where the agent saves data. |
| `numCanvases` | `4` | Number of video processing canvases to create. Each canvas handles one video stream. Range: `2` - `16` |
| `defaultPixelFormat` | `nv12` | Pixel format used by HDMI/SDI input capture devices before any conversion. Values: `nv12` (recommended for H.264/HEVC encoding) \| `yuyv` (better compatibility with deinterlacing). Individual streams can still request a different format via the data API. |
| `noVideoSignal.keepLastFrame` | `false` | Behaviour when a video signal is lost. `true` = freeze on the last valid frame. `false` = display the no-signal screen (background + text defined below). |
| `noVideoSignal.text` | `NO SIGNAL` | Text displayed in the centre of the screen when no signal is present. |
| `noVideoSignal.fontSize` | `100` | Font size (in pixels) of the no-signal text. |
| `noVideoSignal.fontColor` | `255,255,255,255` | Colour of the no-signal text. Format: `R,G,B,A` — each component in the range `0`-`255`. Example: `255,255,255,255` = solid white. |

| Option | Default | Description |
|---|---|---|
| `noVideoSignal.background` | `128,128,128,255` | Background colour of the no-signal screen. Format: `R,G,B,A` — each component in the range `0`-`255`. Example: `128,128,128,255` = solid grey. |
| `filters.directShow` | `true` | Enable or disable the DirectShow virtual device filters (Windows only). |
| `filters.v4l2` | `false` | Enable or disable the V4L2 virtual device filters (Linux only). |
| `devMode` | `false` | Development mode: instantiates boards and inputs even if the hardware is not detected. Useful for testing without physical hardware. |

If you choose to use it, place the file in the same directory as the **lt300agent** executable. A default configuration file is included in the **lt300agent** installation directory. 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. |
|---|---|

## 3.2. lt300agent Controls

The **lt300agent** service can be controlled using the following commands.

*Start (requires administrative privileges)*

```
$ lt300agent start
```

*Stop (requires administrative privileges)*

```
$ lt300agent stop
```

*Check status*

```
$ lt300agent status
```

*Display agent and firmware versions*

```
$ lt300agent version
```

## 3.3. Board Firmware Update

If the boards installed in the host require a firmware update, the **lt300agent** 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.

It is also possible to manually update the board firmware.

*Update the boards firmware manually*

```
$ lt300agent update
```

| | |
|---|---|
| **NOTE** | To update the board firmware manually, the **lt300agent** must be stopped. |

# Chapter 4. Tools

Two tools are installed along with the **lt300agent** service:

- **ecurl** - a command-line tool to send REST API requests to the **lt300agent**.
- **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.

## 4.1. ecurl (CLI)

The **ecurl** program is a developer tool to help you make requests on the **LT API** directly from your terminal. 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 **lt300agent** must be running otherwise **ecurl** will not work. |
|---|---|

### 4.1.1. GET command

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

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

▼ **Examples**

*Retrieve information about the lt300 family agent*

```
$ ecurl get lt300:/
```

*Retrieve informaton from lt300 board located at index 0*

```
$ ecurl get lt300:/0
```

*Make a JPEG capture*

```
$ ecurl get lt300:/0/sdi-in/0/file -d type=image/jpeg
```

### 4.1.2. 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 lt300:/canvas/0/init -d source=$video.mp4
```

*Load a custom HDMI-IN Edid*

```
$ ecurl post lt300:/0/hdmi-in/0/edid -d data=@custom.edid
```

### 4.1.3. 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 lt300:/0/hdmi-in/0
```

*Live listening of hdmi-in audio*

```
$ ecurl play lt300:/0/hdmi-in/0 -d media=audio/pcm
```

### 4.1.4. 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*

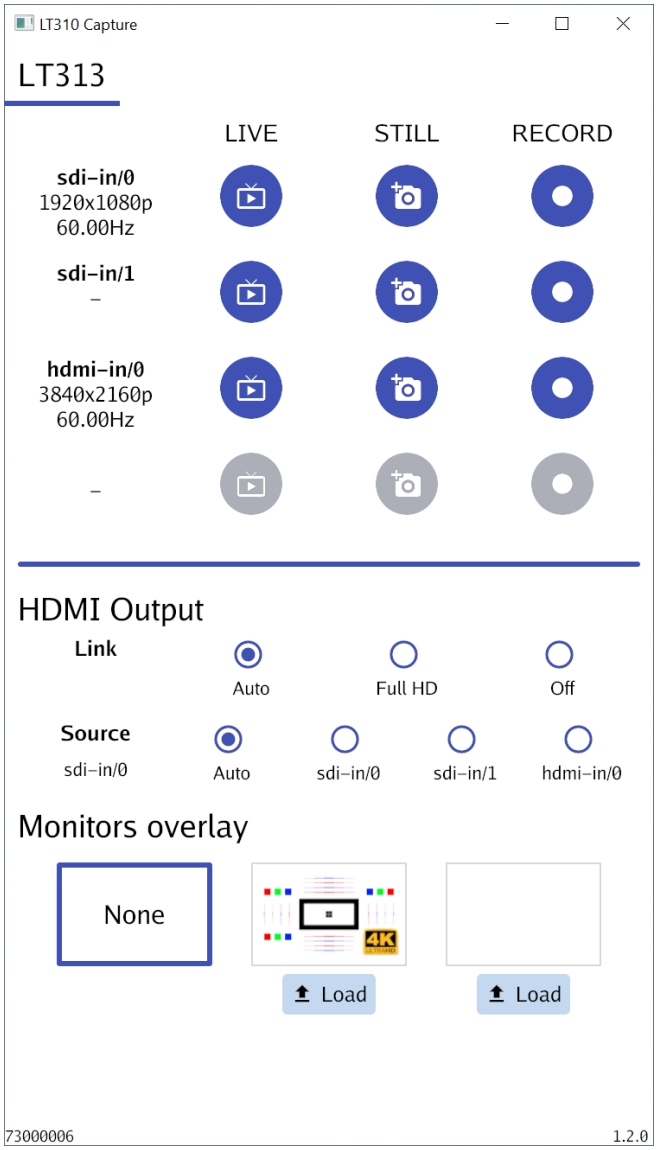
```
$ ecurl rec lt300:/0/hdmi-in/0/file -d media=video/mp4
```

*Record camera video using NVENC hardware encoder and hevc codec*

```
$ ecurl rec lt300:/0/camera/0 -d media=video/mp4 -d extra.hw=nvenc -d extra.videoCodec=hevc
```

## 4.2. ecap (GUI)

The **ecap** program is a graphical user interface designed to test and operate **LT300 boards**. It is available for both Linux and Windows platforms and is built with the Enciris SDK.




On any board input, you can:

- **play** a live preview (video and audio)
- take a **snapshot** (jpeg image)
- start/stop a **recording** (video and audio, MP4 format)

Multiple operations can run simultaneously across one or several inputs.

On the **HDMI output**, you can:

- Select which input is routed to the output
- Disable the output link to stop the video signal or limit it to Full HD resolution
- Optionally apply a **canvas overlay**

**NOTE** The **lt300agent** must be running; otherwise, **ecap** will not work.

# Chapter 5. API Description

This section is the complete reference for all endpoints exposed by the **lt300agent**. It covers the URL structure, request and response formats, and the description of every resource — from board inputs and outputs to workers and the canvas.

## 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 `lt300:`
- A path component made up of path segments separated by slashes `/`

For clarity in this documentation, the **scheme `lt300:` 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": "lt300:/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": "lt300:/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`. |

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

# 5.4. Agent

The **agent** endpoint allows to retrieve the **lt300agent** 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 **lt300agent** software.

GET **lt300:/**

```
{
    "version": "1.5.0"
}
```

| Attribute | Type | Description |
|---|---|---|
| version | string | The current agent version. |

# 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`. |
| 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 associated audio and video streams when available.
Captured data can be retrieved either as raw streams or as files, in various formats and encodings. Native formats are `yuyv` or `nv12` for video and `pcm` for audio. Depending on the requested format, the host CPU and/or GPU may be used for processing and conversion.

The endpoint also allows you to define **capture limits** for the HDMI Input, such as maximum resolution and framerate. These limits are applied globally to the capture pipeline and therefore affect all consumers of the input stream (data workers, file workers, playback, etc.).

| Endpoint | Method | Description |
|---|---|---|
| **/:board/hdmi-in/:id** | GET | Retrieve the current HDMI Input information. |
| **/:board/hdmi-in/:id** | POST | Set capture limits for the HDMI Input. |
| **/: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 represents the input interface and serves two main purposes:

**1. Signal monitoring** - Shows the presence and properties of audio and video signals.

**2. Capture configuration** - Defines maximum capture limits for resolution and framerate. When exceeded, the host streams are automatically reduced: resolution is downscaled by 2x and/or framerate is decimated by 2x. These operations are performed directly by the hardware and do not affect the internal video path feeding the HDMI output.
Defaults match the board maximum capabilities (4K at 60 Hz).

GET, POST **/: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"
    },
    "maxSize": "4k",
    "maxFramerate": 60,
    "maxPixelrate": "4k60"
}
```


| Attribute | Type | Description |
|---|---|---|
| audio | object | Audio object for the HDMI Input (read only). |
| video | object | Video object for the HDMI Input (read only). |
| maxSize | string | Maximum capture resolution preset. Accepted values: `fhd` (2048x1080) or `4k` (4096x2160, default). If the input exceeds this limit, the host stream is downscaled by 2x in both dimensions using hardware. |
| maxFramerate | int | Maximum capture framerate. Accepted values: `30` or `60` (default). If the input exceeds this limit, the host stream is decimated by 2x using hardware. |

| Attribute | Type | Description |
|---|---|---|
| maxPixelrate | string | Maximum pixel-rate preset. Accepted values: `"fhd30"`, `"fhd60"`, `"4k30"`, or `"4k60"` (default). If the input pixel rate exceeds this limit after applying `maxSize` and `maxFramerate`, the host stream is further downscaled and/or decimated to stay within the pixel budget. Limits are composed: `maxSize` and `maxFramerate` are applied first, then `maxPixelrate` is checked on the resulting stream. For example, `maxPixelrate=4k30` with `maxSize=fhd` on a 4K60 input results in FHD@60 output, since the pixel rate after downscaling is below the 4k30 threshold. If you set `maxPixelrate=fhd30` with `maxSize=fhd`, the output will be FHD@30, as both downscaling and decimation are triggered. |

▼ **GET Examples**

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

▼ *ecurl*

```
$ ecurl get lt300:/0/hdmi-in/0
```

▼ *GO*

```
var response lt.Input
err := lt.Get("lt300:/0/hdmi-in/0", &response)
```

▼ *C++*

```
lt::Input response;
lt::error err = lt::Get("lt300:/0/hdmi-in/0", response);
```

▼ *C#*

```
using var client = new lt.Client();
lt.Error err = client.Get("lt300:/0/hdmi-in/0", out lt.Input response);
```

▼ *Python*

```
response, err = Get("lt300:/0/hdmi-in/0")
```

▼ **POST Examples**

These code examples demonstrate how to set the maximum capture resolution to Full HD on hdmi-in 0.

▼ *ecurl*

```
$ ecurl post lt300:/0/hdmi-in/0 -d maxSize=fhd
```

▼ *GO*

```
body := lt.JSON{
    "maxSize": "fhd",
```

```
}
var response lt.Input
err := lt.Post("lt300:/0/hdmi-in/0", body, &response)
```

▼ *C++*

```
lt::json body = {
    {"maxSize", "fhd"}
};
lt::Input response;
lt::error err = lt::Post("lt300:/0/hdmi-in/0", body, response);
```

▼ *C#*

```
using var client = new lt.Client();
var body = new lt.JSON
{
    { "maxSize", "fhd" }
};
lt.Error err = client.Post("lt300:/0/hdmi-in/0", body, out lt.Input response);
```

▼ *Python*

```
body = {
    "maxSize": "fhd"
}
response, err = Post("lt300:/0/hdmi-in/0", body)
```

### 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 Examples**

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

▼ *ecurl*

```
$ ecurl post lt300::0/hdmi-in/0/edid -d data=@customEdid.bin
```

▼ *GO*

```
body := lt.JSON{
    "data": "00FFFFFFFFFF00...",
```

```
}
var response lt.Edid
err := lt.Post("lt300::0/hdmi-in/0/edid", body, &response)
```

▼ *C++*

```
lt::json body = {
    {"data", "00FFFFFFFFFF00..."}
};
lt::Edid response;
lt::error err = lt::Post("lt300::0/hdmi-in/0/edid", body, response);
```

▼ *C#*

```
using var client = new lt.Client();
var body = new lt.JSON
{
    { "data", "00FFFFFFFFFF00..." }
};
lt.Error err = client.Post("lt300::0/hdmi-in/0/edid", body, out lt.Edid response);
```

▼ *Python*

```
body = {
    "data": "00FFFFFFFFFF00..."
}
response, err = Post("lt300::0/hdmi-in/0/edid", body)
```

# 5.7. SDI Input

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

The endpoint also allows you to define **capture limits** for the SDI Input, such as maximum resolution and framerate. These limits are applied globally to the capture pipeline and therefore affect all consumers of the input stream (data workers, file workers, playback, etc.).

| Endpoint | Method | Description |
|---|---|---|
| **/:board/sdi-in/:id** | GET | Retrieve the current SDI Input information. |
| **/:board/sdi-in/:id** | POST | Set capture limits for the SDI Input. |
| **/: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 represents the input interface and serves two main purposes:

**1. Signal monitoring** - Shows the presence and properties of audio and video signals.

**2. Capture configuration** - Defines maximum capture limits for resolution and framerate. When exceeded, the host streams are automatically reduced: resolution is downscaled by 2x and/or framerate is decimated by 2x. These operations are performed directly by the hardware and do not affect the internal video path feeding the HDMI output.
Defaults match the board maximum capabilities (4K at 60 Hz).

GET, POST **/: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"
    },
    "maxSize": "4k",
    "maxFramerate": 60,
    "maxPixelrate": "4k60"
}
```


| Attribute | Type | Description |
|---|---|---|
| audio | object | Audio object for the SDI Input (read only). |
| video | object | Video object for the SDI Input (read only). |
| maxSize | string | Maximum capture resolution preset. Accepted values: `fhd` (2048x1080) or `4k` (4096x2160, default). If the input exceeds this limit, the host stream is downscaled by 2x in both dimensions using hardware. |
| maxFramerate | int | Maximum capture framerate. Accepted values: `30` or `60` (default). If the input exceeds this limit, the host stream is decimated by 2x using hardware. |

| Attribute | Type | Description |
|---|---|---|
| maxPixelrate | string | Maximum pixel-rate preset. Accepted values: `"fhd30"`, `"fhd60"`, `"4k30"`, or `"4k60"` (default). If the input pixel rate exceeds this limit after applying `maxSize` and `maxFramerate`, the host stream is further downscaled and/or decimated to stay within the pixel budget. Limits are composed: `maxSize` and `maxFramerate` are applied first, then `maxPixelrate` is checked on the resulting stream. For example, `maxPixelrate=4k30` with `maxSize=fhd` on a 4K60 input results in FHD@60 output, since the pixel rate after downscaling is below the 4k30 threshold. If you set `maxPixelrate=fhd30` with `maxSize=fhd`, the output will be FHD@30, as both downscaling and decimation are triggered. |

▼ **GET Examples**

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

▼ *ecurl*

```
$ ecurl get lt300:/0/sdi-in/0
```

▼ *GO*

```
var response lt.Input
err := lt.Get("lt300:/0/sdi-in/0", &response)
```

▼ *C++*

```
lt::Input response;
lt::error err = lt::Get("lt300:/0/sdi-in/0", response);
```

▼ *C#*

```
using var client = new lt.Client();
lt.Error err = client.Get("lt300:/0/sdi-in/0", out lt.Input response);
```

▼ *Python*

```
response, err = Get("lt300:/0/sdi-in/0")
```

▼ **POST Examples**

These code examples demonstrate how to set the maximum capture resolution to Full HD on sdi-in 0.

▼ *ecurl*

```
$ ecurl post lt300:/0/sdi-in/0 -d maxSize=fhd
```

▼ *GO*

```
body := lt.JSON{
    "maxSize": "fhd",
```

```
}
var response lt.Input
err := lt.Post("lt300:/0/sdi-in/0", body, &response)
```

▼ *C++*

```
lt::json body = {
    {"maxSize", "fhd"}
};
lt::Input response;
lt::error err = lt::Post("lt300:/0/sdi-in/0", body, response);
```

▼ *C#*

```
using var client = new lt.Client();
var body = new lt.JSON
{
    { "maxSize", "fhd" }
};
lt.Error err = client.Post("lt300:/0/sdi-in/0", body, out lt.Input response);
```

▼ *Python*

```
body = {
    "maxSize": "fhd"
}
response, err = Post("lt300:/0/sdi-in/0", body)
```

# 5.8. HDMI Output

The **hdmi-out** endpoint allows you to configure the physical HDMI output ports on a **LT board**. It exposes source selection, overlay, color format and link carrier settings for each output, along with read-only audio and video signal status.

| 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`. |
| 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 Examples**

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

▼ *ecurl*

```
$ ecurl get lt300:/0/hdmi-out/0
```

▼ *GO*

```
var response lt.Output
err := lt.Get("lt300:/0/hdmi-out/0", &response)
```

▼ *C++*

```
lt::Output response;
lt::error err = lt::Get("lt300:/0/hdmi-out/0", response);
```

▼ *C#*

```
using var client = new lt.Client();
lt.Error err = client.Get("lt300:/0/hdmi-out/0", out lt.Output response);
```

▼ *Python*

```
response, err = Get("lt300:/0/hdmi-out/0")
```

▼ **POST Examples**

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

▼ *ecurl*

```
$ ecurl post lt300:/0/hdmi-out/0 -d overlay=canvas/0
```

▼ *GO*

```
body := lt.JSON{
    "overlay": "canvas/0",
}
var response lt.Output
err := lt.Post("lt300:/0/hdmi-out/0", body, &response)
```

▼ *C++*

```
lt::json body = {
    {"overlay", "canvas/0"}
};
lt::Output response;
lt::error err = lt::Post("lt300:/0/hdmi-out/0", body, response);
```

▼ *C#*

```
using var client = new lt.Client();
var body = new lt.JSON
{
    { "overlay", "canvas/0" }
};
lt.Error err = client.Post("lt300:/0/hdmi-out/0", body, out lt.Output response);
```

▼ *Python*

```
body = {
    "overlay": "canvas/0"
}
response, err = Post("lt300:/0/hdmi-out/0", body)
```

## 5.9. Canvas

The **canvas** endpoint is both a virtual 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 output.

*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. |
| **/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.

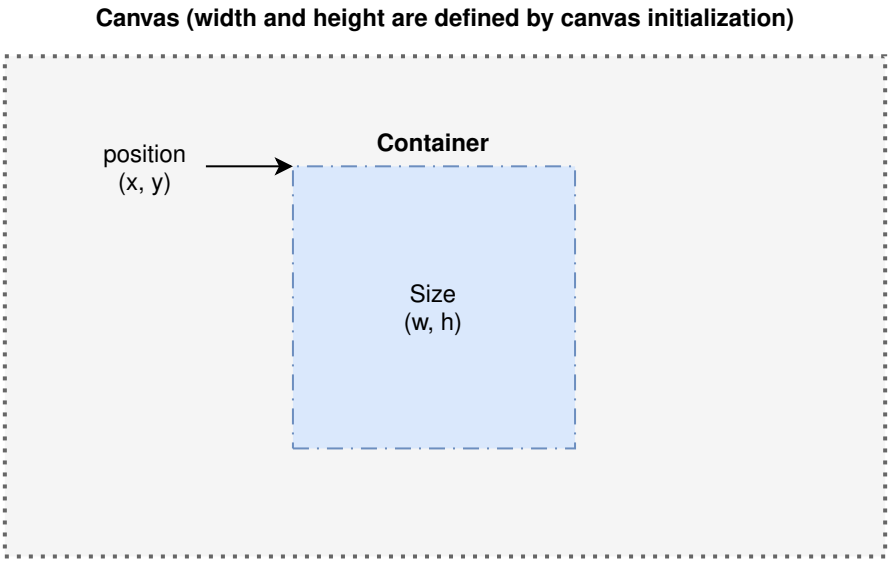
GET **/canvas/:id**


```
{
    "video": {
        "description": "",
        "format": "rgba",
        "framerate": 30,
        "interlaced": false,
        "signal": "locked",
        "size": [3840, 2160]
    }
}
```


| Attribute | Type | Description |
|---|---|---|
| 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. |

| Parameter | Type | Default | Description |
|---|---|---|---|
| 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.** |
| 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`. Can also be set to an absolute path to a `.ttf` font file. |
| 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": 2,
    "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 2. |
| 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 2. |
| 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": 2,
    "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 `2`. |
| 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}`. |
| 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. |

| Attribute | Type | Description |
|---|---|---|
| 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]`. |

| Attribute | Type | Description |
|---|---|---|
| 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* **lt300:/canvas/0/rectangle**

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

*POST* **lt300:/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 lt300:/canvas/0/rectangle -d position=50,900 -d size=1200,100 -d fill=0,0,0,180 -d
rounded=10

$ ecurl post lt300:/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("lt300:/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("lt300:/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("lt300:/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("lt300:/canvas/0/text", body, nullptr);
```

▼ *C#*

```
// Add semi-transparent background bar
var body = new lt.JSON
{
    { "position", new int[] { 50, 900 } },
    { "size", new int[] { 1200, 100 } },
    { "fill", new int[] { 0, 0, 0, 180 } },
    { "rounded", 10 }
};
err = lt.Post("lt300:/canvas/0/rectangle", body, null);

// Add text over the background
body = new lt.JSON
{
    { "position", new int[] { 50, 900 } },
    { "size", new int[] { 1200, 100 } },
    { "text", "Hello, World!" },
    { "fontSize", 48 },
    { "color", new int[] { 255, 255, 255, 255 } },
    { "align", "center" }
};
err = lt.Post("lt300:/canvas/0/text", body, null);
```

▼ *Python*

```
# Add semi-transparent background bar
body = {
    "position": [50, 900],
    "size": [1200, 100],
    "fill": [0, 0, 0, 180],
    "rounded": 10
}
resp, err = Post("lt300:/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("lt300:/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* **lt300:/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 lt300:/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("lt300:/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("lt300:/canvas/0/ops", body, nullptr);
```

▼ *C#*

```
var body = new lt.JSON
{
    { "ops", new lt.JSON[]
        {
            new lt.JSON
            {
                { "op", "init" },
                { "size", new int[] { 3840, 2160 } },
                { "color", new int[] { 0, 0, 0, 255 } },
                { "framerate", 30.0 }
            },
            new lt.JSON
            {
                { "op", "video" },
                { "source", "0/sdi-in/0" },
                { "position", new int[] { 0, 600 } },
                { "size", new int[] { 1920, 1080 } }
            },
            new lt.JSON
            {
                { "op", "text" },
                { "text", "SDI 0" },
                { "fontSize", 48 },
                { "color", new int[] { 255, 255, 255, 255 } },
                { "align", "center" },
                { "position", new int[] { 0, 1680 } },
                { "size", new int[] { 1920, 100 } }
            },
            new lt.JSON
            {
                { "op", "video" },
                { "source", "0/sdi-in/1" },
                { "position", new int[] { 1920, 600 } },
                { "size", new int[] { 1920, 1080 } }
            },
            new lt.JSON
            {
                { "op", "text" },
                { "text", "SDI 1" },
                { "fontSize", 48 },
                { "color", new int[] { 255, 255, 255, 255 } },
                { "align", "center" },
                { "position", new int[] { 1920, 1680 } },
                { "size", new int[] { 1920, 100 } }
            }
        }
    }
};
err = lt.Post("lt300:/canvas/0/ops", body, null);
```

▼ *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("lt300:/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:

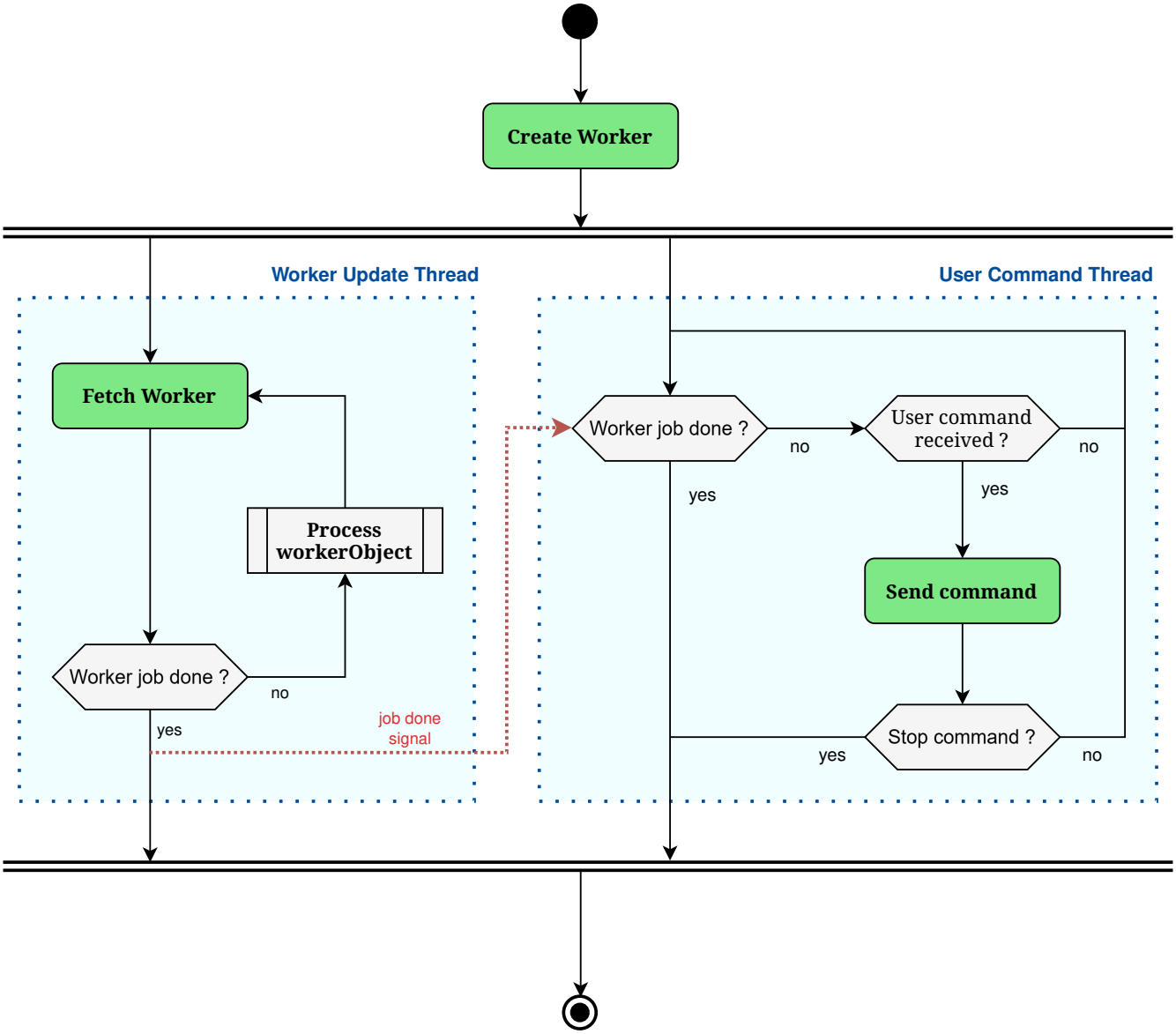
- `data` workers expose media data directly to the client application for real-time analysis or custom processing. The worker object provides packets containing shared memory buffers with raw video or audio data, ensuring minimal latency and zero-copy transfer. Data workers are useful for applications 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 object model,
- 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` or `:board/sdi-in/:id`.

*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. Supported values: `44100`, `48000`. 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|string)`

The duration of the recording. Accepts an integer value in seconds, or a string with the suffix `s`, `m`, or `h` such as `30`, `"30s"`, `"5m"`, or `"2h"`. Default: `0` (infinite).

**splitSize** `(int|string)`

The maximum size of each split file. Accepts an integer value in megabytes, or a string with the suffix `m` or `g` such as `500`, `"500m"`, or `"2g"`. Default: `0` (unlimited, no splitting).

**splitDuration** `(int|string)`

The duration of each split file. Accepts an integer value in seconds, or a string with the suffix `s`, `m`, or `h` such as `30`, `"30s"`, `"5m"`, or `"2h"`. 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": "lt300:/:url/data",
    "body": {
        "media": "audio/pcm",
        "channels": 2,
        "samplerate": 48000,
        "depth": 16
    }
}
```

*response*

```
{
    "location": "lt300:/client/jobs/...",
    "error": "redirect"
}
```

### POST /:url/file

*request*

```
{
    "method": "POST",
    "url": "lt300:/:url/file",
    "body": {
        "media": "audio/pcm",
        "channels": 2,
        "samplerate": 48000,
        "depth": 16,
        "location": "path_to_directory",
        "duration": "1h",
        "splitSize": "2g",
        "splitDuration": "15m"
    }
}
```

*response*

```
{
    "location": "lt300:/client/jobs/...",
    "error": "redirect"
}
```

▼ **Examples**

▼ *GO*

```go
// Create Data worker
err := lt.Post("lt300:/:url/data", lt.AudioDataWorker{Media: "audio/pcm"}, nil)

// Create File worker
// err := lt.Post("lt300:/:url/file", lt.AudioFileWorker{Media: "audio/pcm"}, 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("lt300:/:url/data", lt::AudioDataWorker{"audio/pcm"}, nullptr);

// Create File worker
// lt::error err = lt::Post("lt300:/:url/file", lt::AudioFileWorker{"audio/pcm"}, nullptr);

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

▼ *C#*

```csharp
using var client = new lt.Client();

// Create Data worker
lt.Error err = client.Post("lt300:/:url/data", new lt.AudioDataWorker {{ Media = "audio/pcm" } },
out _);

// Create File worker
```

```
// lt.Error err = client.Post("lt300:/:url/file", new lt.AudioFileWorker {{ Media = "audio/pcm" }
}, out _);

if (!lt.ErrorIs(err, lt.ErrRedirect)) {
    throw new Exception($"Worker creation failed: {err}");
}
string workerURL = lt.Errors.RedirectLocation(err);
```

▼ *Python*

```
# Create Data worker
resp, err = Post("lt300:/:url/data", {'media': "audio/pcm"})

# Create File worker
# resp, err = Post("lt300:/:url/file", {'media': "audio/pcm"})

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. Leave empty to use the default size.

*Additional file worker parameters*

**location** `string`

The file location to save the recorded image. 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": "lt300:/:url/data",
    "body": {
        "media": "image/jpeg",
        "size": [1920, 1080]
    }
}
```

*response*

```
{
    "location": "lt300:/client/jobs/...",
    "error": "redirect"
}
```

### POST /:url/file

*request*

```
{
    "method": "POST",
    "url": "lt300:/:url/file",
    "body": {
        "media": "image/jpeg",
        "size": [1920, 1080],
        "location": "path_to_directory"
    }
}
```

*response*

```
{
    "location": "lt300:/client/jobs/...",
    "error": "redirect"
}
```

▼ **Examples**

▼ *GO*

```go
// Create Data worker
err := lt.Post("lt300:/:url/data", lt.ImageDataWorker{Media: "image/jpeg"}, nil)

// Create File worker
// err := lt.Post("lt300:/: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("lt300:/:url/data", lt::ImageDataWorker{"image/jpeg"}, nullptr);

// Create File worker
// lt::error err = lt::Post("lt300:/:url/file", lt::ImageFileWorker{"image/jpeg"}, nullptr);

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

▼ *C#*

```cs
using var client = new lt.Client();

// Create Data worker
lt.Error err = client.Post("lt300:/:url/data", new lt.ImageDataWorker {{ Media = "image/jpeg" }
}, out _);

// Create File worker
// lt.Error err = client.Post("lt300:/:url/file", new lt.ImageFileWorker {{ Media = "image/jpeg"
} }, out _);

if (!lt.ErrorIs(err, lt.ErrRedirect)) {
    throw new Exception($"Worker creation failed: {err}");
```

```
}
string workerURL = lt.Errors.RedirectLocation(err);
```

▼ *Python*

```
# Create Data worker
resp, err = Post("lt300:/:url/data", {'media': "image/jpeg"})

# Create File worker
# resp, err = Post("lt300:/: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. Leave empty to use the default size.

**framerate `float`**

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

**samplerate `int`**

Audio sample rate in Hz for MP4 recording. Supported values: `44100`, `48000`. Default: `48000`.

*Additional file worker parameters*

**location `string`**

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

**duration `(int|string)`**

The duration of the recording. Accepts an integer value in seconds, or a string with the suffix `s`, `m`, or `h` such as `30`, `"30s"`, `"5m"`, or `"2h"`. Default `0` (infinite).

**splitSize `(int|string)`**

The maximum size of each split file. Accepts an integer value in megabytes, or a string with the suffix `m` or `g` such as `500`, `"500m"`, or `"2g"`. Default: `0` (unlimited, no splitting).

**splitDuration `(int|string)`**

The duration of each split file. Accepts an integer value in seconds, or a string with thesuffix `s`, `m`, or `h` such as `30`, `"30s"`, `"5m"`, or `"2h"`. 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
- `videoCodec` - Video codec to use: `"h264"` or `"hevc"`
- `audioCodec` - Audio codec to use: `"aac"` or `"mp3"` (used in MP4 container)
- `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**

### POST /:url/data

*request*

```
{
    "method": "POST",
    "url": "lt300:/:url/data",
    "body": {
        "media": "video/nv12",
        "size": [1920, 1080],
        "framerate": 30
    }
}
```

*response*

```
{
    "location": "lt300:/client/jobs/...",
    "error": "redirect"
}
```

**File Worker Creation**

### POST /:url/file

*request*

```
{
    "method": "POST",
    "url": "lt300:/:url/file",
    "body": {
        "media": "video/mp4",
        "size": [1920, 1080],
        "framerate": 30,
        "samplerate": 0,
        "location": "path_to_directory",
        "duration": "1h",
        "splitSize": "2g",
        "splitDuration": "15m",
        "extra": {
            "hw": "",
            "bitrate": 0,
            "quality": 0,
            "gop": 0,
            "videoCodec": "",
            "audioCodec": "",
            "preset": ""
        }
    }
}
```

▼ **Examples**

▼ *GO*

```go
// Create Data worker
err := lt.Post("lt300:/:url/data", lt.VideoDataWorker{Media: "video/nv12"}, nil)

// Create File worker
// err := lt.Post("lt300:/:url/file", 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("lt300:/:url/data", lt::VideoDataWorker{"video/nv12"}, nullptr);

// Create File worker
// lt::error err = lt::Post("lt300:/:url/file", lt::VideoFileWorker{"video/nv12"}, nullptr);

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

▼ *C#*

```csharp
using var client = new lt.Client();

// Create Data worker
lt.Error err = client.Post("lt300:/:url/data", new lt.VideoDataWorker {{ Media = "video/nv12" }
}, out _);

// Create File worker
// lt.Error err = client.Post("lt300:/:url/file", new lt.VideoFileWorker {{ Media = "video/nv12"
} }, out _);

if (!lt.ErrorIs(err, lt.ErrRedirect)) {
    throw new Exception($"Worker creation failed: {err}");
}
string workerURL = lt.Errors.RedirectLocation(err);
```

▼ *Python*

```python
# Create Data worker
resp, err = Post("lt300:/:url/data", {'media': "video/nv12"})

# Create File worker
# resp, err = Post("lt300:/: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.

**length** `int`

Quantity of byte processed since the segment started.

**status** `string`

Worker current status (`running`, `paused`, `splitting` or `completed`). The `splitting` status indicates that a file split is in progress (file workers only). The `completed` status indicates that the worker has finished its task and will be removed.

**packets** `map[int]packet`

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

**Worker object**

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

### 5.10.3. Packet Object

Packets are the fundamental units delivered by a worker. They contain metadata and media data (PCM audio samples, raw video frames, encoded bitstreams, etc.).

Under the hood, the server delivers packets in two ways depending on the payload size:

- **Inline data**: for small payloads, the media data is embedded directly in the JSON response as a base64-encoded `data` field.
- **Shared memory**: for large payloads (e.g., raw video frames), the server allocates a shared memory buffer and returns a reference (`ref`, `handle`) instead of the data itself. This enables zero-copy transfer between the server and the client.

The SDK transparently handles both cases: it decodes inline data or maps the shared memory buffer, and exposes the result through a single unified `Packet` object. From the developer's perspective, every packet has a `data` field containing the media bytes, regardless of the underlying transport mechanism.

| | |
|---|---|
| **IMPORTANT** | Each packet must be released after use by calling `Close()`. This frees the associated resources, and — in the case of shared memory packets — notifies the server that the buffer can be reused. |

**track** `int`

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

**media** `string`

The packet media 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.

**data** `[]byte`

The media data bytes. Populated transparently by the SDK from either inline data or shared memory.

**ref** `string`

Empty for inline data packets. For shared memory packets, contains the server-side reference used for cleanup. The presence of this field indicates that the data was loaded from shared memory, but the developer does not need to handle it — `Close()` takes care of releasing the reference.

**Packet object**


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


**Packet object (shared memory)**


```
{
    "track": 0,
    "media": "video/nv12",
    "signal": "locked",
    "timestamp": 1695815814430318,
    "meta": {
        "size": [1920, 1080],
        "framerate": 30,
        "interlaced": false,
        "keyframe": true
    },
    "data": "bytes_array",
    "ref": "lt300:/client/ref/..."
}
```


#### 5.10.3.1. Metadata

The content of metadata structure present in 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`

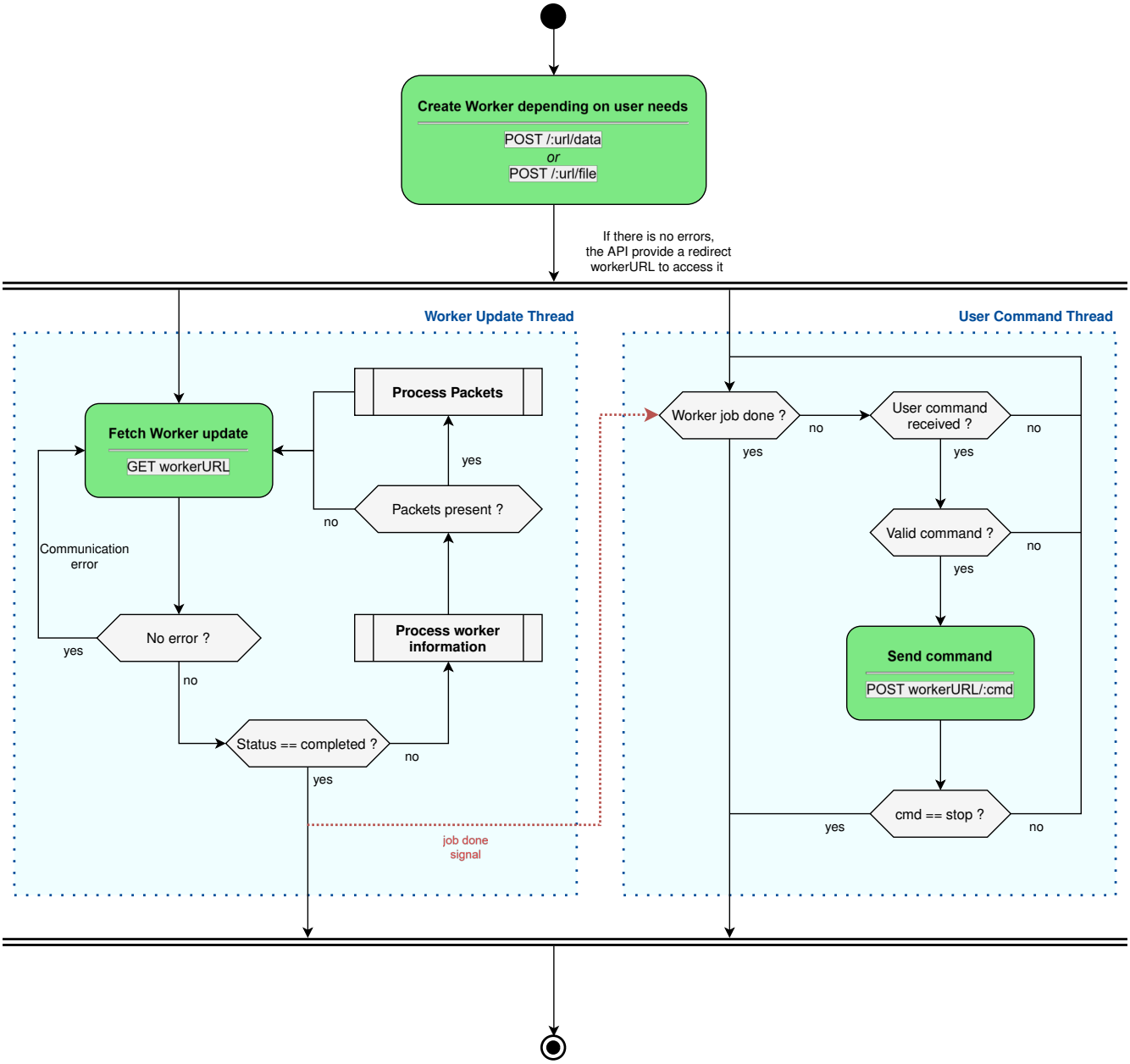
Is the frame intra coded.

**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:

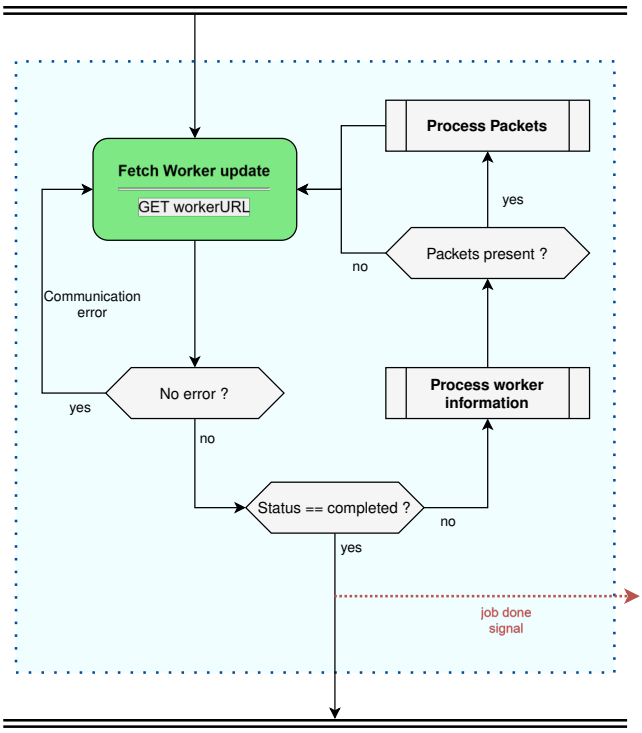
- **Worker Update Thread**: continuously fetches the worker object from the server, updating its status and packets. The loop continues until the worker status becomes `completed`, indicating that the worker has finished its task.
- **User Command Thread (optional)**: processes user commands and forwards them to the worker. Typical commands include `stop`, `pause`, and `start`. Sending such a command will eventually lead to the worker status becoming `completed`, 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 when the worker reaches the `completed` status. 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 error occurs, handle it appropriately (log, retry, etc.).

3. `Check worker status`

   If the status is `completed`, the worker has finished its task. The client should stop fetching updates.
   Otherwise, continue to process the worker information and packets.

4. `Process worker information`

   Extract status, progress, and data packets availability.

5. `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()`.

6. `Repeat`

   Return to step 1 until the worker status is `completed`.

#### 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`**

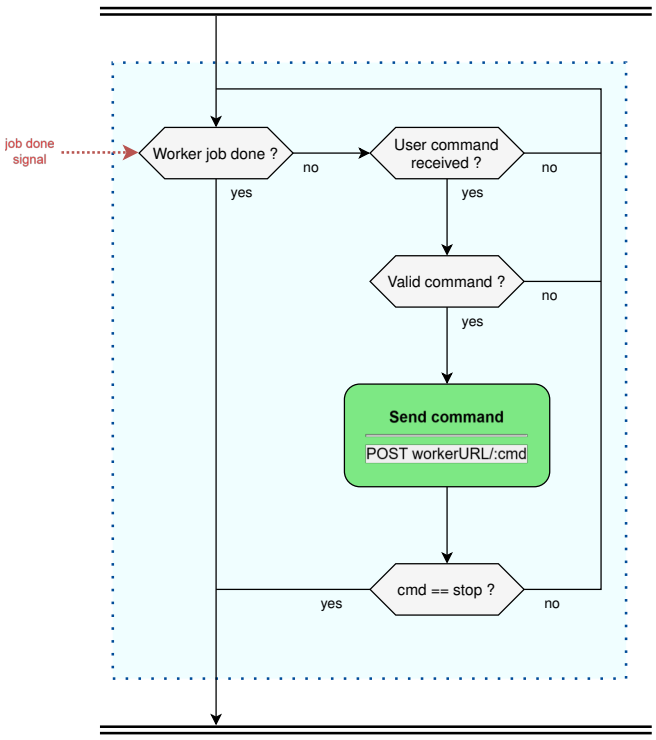
   Ensure the command is valid and POST it to `workerURL`.

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. Examples

This section provides practical examples demonstrating common usage of workers.

### 5.10.5.1. 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": "lt300:/0/hdmi-in/0/data",
    "body": {
        "media": "audio/pcm",
        "channels": 2,
        "samplerate": 48000,
        "depth": 16
    }
}
```

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

*response (redirect)*

```
redirect: lt300:/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": "lt300:/client/jobs/VCW90ecK90GwE"
}
```

Possible outcomes:

- `null`: request successful, worker object returned.
- 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 worker status is `completed`.

**Worker completion**

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 worker status becomes `completed`, which signals that the worker has terminated.

#### 5.10.5.2. 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": "lt300:/0/sdi-in/0/file",
    "body": {
        "media": "video/mp4",
        "location": "C:\\Users\\A\\Videos",
        "duration": "10m",
        "splitSize": ,
        "splitDuration": "5s",
        "size": [
            0,
            0
        ],
        "framerate": 0,
        "samplerate": 0,
        "extra": {
            "hw": "",
            "bitrate": 0,
            "quality": 0,
            "gop": 0,
            "videoCodec": "",
```

```
            "audioCodec": "",
            "preset": ""
        }
    }
}
```

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

*response (redirect)*

```
redirect: lt300:/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": "lt300:/client/jobs/NBCgXxNEOsc"
}
```

Possible outcomes:

- `null`: request successful, worker object returned.
- 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 to `splitting` when a split is occurring.
- 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 worker status is `completed`.

#### Worker completion

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

# Chapter 6. SDK Examples

The Enciris SDK provides complete examples demonstrating how to use the API to perform common multimedia tasks.

Each example is available for the supported programming languages (C++, C#, Go and Python), and full source code is provided. The following sections provide an overview of the available examples.

## 6.1. Audio and video player

Connects to an input source and plays back its video and audio streams in real time. A video window renders the decoded frames, while audio is sent directly to the system's output device.

*Actions demonstrated*

- Retrieve input source status
- Create a video data worker and an audio data worker concurrently
- Fetch and process packets in separate threads
- Decode and render video frames with OpenGL
- Decode and play audio samples
- Display real-time statistics (FPS, dropped packets)
- Stop both workers on window close

The full source code is available in the SDK:

```
sdk/<lang>/examples/av_player
```

## 6.2. HDMI Output

Configures the board's HDMI output to display a live input source from the same board, with a text and a countdown drawn on a canvas then applied as an overlay.

*Actions demonstrated*

- Verify that a board input source has a locked signal
- Route a board input to the board's HDMI output
- Create and initialize a canvas
- Draw text on the canvas
- Apply the canvas as an overlay on the HDMI output
- Clear an area of the canvas

The full source code is available in the SDK:

```
sdk/<lang>/examples/hdmi_out
```

## 6.3. Inputs Status

Enumerates all connected boards and prints the status of each of their inputs, including video format, resolution, frame rate and audio format.

*Actions demonstrated*

- Enumerate connected LT devices
- Iterate over all available inputs per device
- Fetch and display input signal status (resolution, frame rate, audio format)

The full source code is available in the SDK:

```
sdk/<lang>/examples/inputs
```

## 6.4. Recording

Records a video stream from an input source into MP4 files on disk. The recording can be paused, resumed and stopped interactively from the keyboard. Files are automatically split by duration.

*Actions demonstrated*

- Verify that an input source has a locked signal
- Create a video file worker with MP4 encoding and automatic file splitting
- Monitor worker status and file metadata in real time
- Pause, resume and stop the recording via keyboard commands (`Space` to pause/resume, any other key to stop)

The full source code is available in the SDK:

```
sdk/<lang>/examples/record
```

## 6.5. Snapshot

Captures a single video frame from an input source and saves it as a JPEG image file.

*Actions demonstrated*

- Verify that selected input source has a locked signal

- Create an image file worker for single-frame capture
- Wait for the worker to complete
- Retrieve image metadata (file name, resolution, file size)
- Release the packet after use

The full source code is available in the SDK:

```
sdk/<lang>/examples/still_capture
```

# Chapter 7. Contact

For any questions, feedback, or support inquiries, please reach out to us:

| | |
|---|---|
| Email | support@enciris.com |
| Phone | +33 (0)5 82 95 09 55 |
| Website | www.enciris.com |

**NOTE** For technical support, please include your board serial number and software version in your message.

# Chapter 8. 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 | POST | Set capture limits for the HDMI Input. |
| /: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 | POST | Set capture limits for the SDI Input. |
| /: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 9. Changelog

```
# 1.5.0 (24/04/2026):
- Renamed from "LT310" to "LT300"
- Add MP4 audio/video recording
- Add input capture limitation
- Add background support in canvas text
- Add hardware encoder preference for H.264 and HEVC
- Add C# client & examples to SDK
- Improve QoS
- Improve HDMI input compatibility
- Improve SDK clients & examples (C++, Go, Python)
- Improve file splitting status
- Update documentation
- Minor fixes

# 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
```