



**MAIMAN**  
**ELECTRONICS**

# **Laser Diode Driver Library**

Maiman Electronics

e-mail: [info@maimanelectronics.com](mailto:info@maimanelectronics.com)

website: [www.maimanelectronics.com](http://www.maimanelectronics.com)

**Version:** 1.0.1 - 2025

**Python Version:** 3.9 or higher

# Maiman Electronics

<b>LASER DIODE DRIVER LIBRARY DOCUMENTATION</b>	<b>2</b>
OVERVIEW	2
KEY FEATURES	2
USE CASES	3
LIBRARY COMPONENTS	3
REQUIREMENTS	3
<b>DEVICE CLASS</b>	<b>4</b>
CONSTRUCTOR	4
PUBLIC METHODS	4
<b>DEVICEFACTORY CLASS</b>	<b>7</b>
CLASS ATTRIBUTES	7
STATIC METHODS	8
<b>SERIALCOMMUNICATION CLASS</b>	<b>8</b>
CONSTRUCTOR	8
PUBLIC METHODS	9
PRIVATE METHODS	10
<b>DEVICEMODEL CLASS</b>	<b>12</b>
CONSTRUCTOR	12
PUBLIC METHODS	12
PRIVATE METHODS	13
CONFIGURATION SECTIONS	16
CONSTRUCTOR	17
PUBLIC METHODS	17
PRIVATE METHODS	18
PROPERTIES	18
YAML FILE STRUCTURE	19
<b>COMMAND DATACLASS</b>	<b>22</b>
FIELDS	22
ADDITIONAL FIELDS FOR SPECIFIC COMMAND TYPES	24
DEVICESTATES ENUM	25
<b>LOGGER CLASS</b>	<b>25</b>
CONSTRUCTOR	25
PUBLIC METHODS	26
PRIVATE METHODS	27
LOGEVENT DATACLASS	27
EXCEPTION: DEVICEERROR	27
<b>EXAMPLE OF USAGE</b>	<b>28</b>
RUNNING THE TEST SCRIPT	28
LIST OF METHODS USED IN THE SCRIPT:	29
CREATING CUSTOM METHODS	31

# Laser Diode Driver Library Documentation

## Overview

The **Laser Diode Driver Library** is a Python-based library designed to facilitate the communication, configuration, and control of laser diode drivers through serial communication. This library provides a comprehensive and modular interface for interacting with various device parameters, such as current, frequency, voltage, and duration, while also supporting device state management and error handling.

With built-in support for logging, state management, and device monitoring, the library ensures that users can efficiently manage laser diode drivers with minimal overhead. Whether you're working on a simple configuration task or need more complex device operations, this library provides the necessary tools for streamlined control.

## Key Features

- **Serial Communication:** Easily interface with laser diode drivers over serial communication (UART) using the `SerialCommunication` class.
- **Device Configuration:** Load and apply device settings from YAML configuration files using the `DeviceConfig` class, making it simple to configure and manage devices across different projects.
- **Parameter Management:** Set and retrieve essential device parameters such as current, frequency, voltage, and duration via the `Device` class.
- **Device State Control:** Control and monitor the state of the laser diode driver, allowing you to enable or disable operations, manage current sources, and control interlocks.
- **Logger Integration:** Log device operations, warnings, and errors with various levels of verbosity (DEBUG, INFO, WARNING, ERROR) using the `Logger` class. This makes it easy to track device behavior and troubleshoot any issues that arise during operation.
- **Factory Pattern for Device Management:** The library uses a `DeviceFactory` class, which provides a singleton instance of the device. This pattern ensures that the same instance is reused throughout the application, simplifying device management.
- **Custom Commands and Operations:** Implement custom device operations by leveraging the flexible command structure within the library. Users can send commands directly to the device or configure the device's behavior through high-level abstractions.
- **Error Handling:** The library provides built-in error handling and custom exceptions like `DeviceError` to ensure smooth device operations and provide clear diagnostics when issues arise.
- **Extensibility:** The modular design allows developers to extend or customize the library for their specific needs. Whether adding support for new commands or handling new devices, the architecture is designed to be flexible and scalable.

## Use Cases

- **Device Calibration:** Users can set precise parameters such as current and voltage for calibrating laser diode drivers.
- **Laser Diode Control:** Easily control the operation states, including enabling or disabling the laser diode, and retrieve real-time device statuses.
- **Automated Testing:** Integrate the library into automated systems to test and verify laser diode drivers for specific conditions or configurations.
- **Data Logging:** Use the logging features to record operations and errors for diagnostics and post-operation analysis.

## Library Components

- **DeviceFactory:** Centralized management of the device instance, ensuring singleton behavior and simplifying initialization.
- **Device:** High-level interface for interacting with device parameters and performing operations.
- **DeviceModel:** Encapsulation of device parameters and command codes, providing a structured approach to device configuration.
- **SerialCommunication:** Handles the low-level communication with the laser diode driver over serial ports.
- **Command:** Dataclass for storing command details such as codes, values, and descriptions.
- **Logger:** Provides detailed logging of device operations, errors, and events, supporting various log levels.
- **DeviceConfig:** Loads and manages configuration settings from YAML files, enabling easy device configuration and updates.

## Requirements

The *requirements.txt* file lists the external dependencies required to run and develop the Laser Diode Driver library.

### *Core Dependencies:*

- **pyserial>=3.5:** For serial communication with the laser diode driver hardware.
- **PyYAML>=6.0.1:** For parsing YAML configuration files.

### *Development and Testing Dependencies:*

- **pytest>=8.3.3:** For running automated tests.

To install the dependencies, run:

```
pip install -r requirements.txt
```

This ensures all necessary packages are installed for both runtime and development environments.

---

## Device Class

The Device class represents a Singleton that manages communication with a physical device. It provides methods to retrieve and set parameters, manage device states, and handle logging of events. It interfaces with the device via SerialCommunication, retrieves configurations from DeviceConfig, and uses DeviceModel to manage device commands and parameters.

### Constructor

```
__new__ (cls, *args, **kwargs)
```

Ensures that only one instance of the Device class is created (Singleton pattern).

```
__init__ (self, device_model: DeviceModel, serial_comm: SerialCommunication, device_config: DeviceConfig, logger: Optional [Logger] = None)
```

Initializes the Device instance, setting up serial communication, configuration, and logging.

- **Parameters:**
  - device\_model (DeviceModel): Manages device parameters and commands.
  - serial\_comm (SerialCommunication): Handles serial communication with the device.
  - device\_config (DeviceConfig): Loads and stores device configurations.
  - logger (Optional [Logger]): Optional logger for recording events and errors.

---

## Public Methods

```
log_event (self, message: str, level: str = LOG_LEVEL_INFO) -> None
```

Logs events with proper encoding to handle UTF-8 and ASCII fallback in case of encoding issues.

- **Parameters:**
  - message (str): The message to log.
  - level (str): The log level (INFO, DEBUG, ERROR, WARNING).

```
__getattr__ (self, name: str)
```

Handles dynamic attribute access for parameters and states. This method is invoked automatically when an attribute is accessed that does not already exist on the instance. It

provides support for methods like `get_<parameter>`, `set_<parameter>`, `getState_<state>`, and `setState_<state>` to retrieve or set parameter values or device states dynamically.

- **Parameters:**
  - `name (str)`: The name of the dynamically accessed attribute.
- **Raises:**
  - `AttributeError`: If the dynamically accessed attribute does not follow the expected naming conventions.

*`retrieve_device_id(self) -> int`*

Retrieves the device ID from the hardware and logs the result. Handles any exceptions during communication.

- **Returns:**
  - The device ID as a 4-character hexadecimal string.

*`initialize_device(self) -> None`*

Initializes the device by retrieving its ID, loading its configuration, and setting up commands from the model.

*`get_parameter (self, name: str, param_type: str) -> Optional [Union[float, int]]`*

Retrieves a parameter value from the device, handling signed conversions and applying the correct divider for scaling.

- **Parameters:**
  - `name (str)`: The name of the parameter.
  - `param_type (str)`: The type of the parameter (e.g., "code").
- **Returns:**
  - The retrieved parameter value, scaled appropriately.

*`set_parameter (self, name: str, param_type: str, value: float) -> None`*

Sets a parameter value on the device by sending the appropriate command.

- **Parameters:**
  - `name (str)`: The name of the parameter to set.
  - `param_type (str)`: The type of the parameter (e.g., "code").
  - `value (float)`: The value to set.

*`get_state (self, state_name: str) -> Optional [Dict [str, str]]`*

Retrieves and decodes the state of the device for a specified state command.

- **Parameters:**
  - `state_name` (str): The name of the state to retrieve.
- **Returns:**
  - A dictionary of decoded state values.

*set\_state (self, state\_name: str, state\_values: Dict [str, str]) -> None*

Sets the state of the device by sending commands to update specific bits.

- **Parameters:**
  - `state_name` (str): The name of the state to set.
  - `state_values` (Dict [str, str]): A dictionary of state values to set.

*decode\_state (self, bitmask: int, bits: Dict) -> Dict [str, str]*

Decodes the device state from a bitmask by comparing the bits against predefined states.

- **Parameters:**
  - `bitmask` (int): The bitmask value representing the state.
  - `bits` (Dict): The bit layout and possible states.
- **Returns:**
  - A dictionary mapping each bit to its decoded state.

*check\_device\_status(self) -> Dict [str, str]*

Checks the device's status by retrieving raw status bits and decoding them into human-readable states.

- **Returns:**
  - A dictionary containing the raw status and specific state flags (e.g., whether the operation has started).

*get\_raw\_status (self, state\_name: str) -> int*

Retrieves the raw status bitmask for a given state command.

- **Parameters:**
  - `state_name` (str): The name of the state to retrieve.
- **Returns:**
  - The raw status as an integer.

*is\_bit\_set (self, state\_name: str, bit: str) -> bool*

Checks if a specific bit is set in a state bitmask.

- **Parameters:**
  - state\_name (str): The name of the state.
  - bit (str): The bit to check.
- **Returns:**
  - True if the bit is set, otherwise False.

*is\_operation\_state\_started(self) -> bool*

Checks if the "operation state started" bit is set.

*is\_current\_set\_internal(self) -> bool*

Checks if the "current set internal" bit is set.

*is\_enable\_internal(self) -> bool*

Checks if the "enable internal" bit is set.

*is\_external\_ntc\_interlock\_denied(self) -> bool*

Checks if the "external NTC interlock denied" bit is set.

*is\_interlock\_denied(self) -> bool*

Checks if the "interlock denied" bit is set.

---

## DeviceFactory Class

The DeviceFactory class is a Singleton factory responsible for creating and managing a single instance of the Device class. This factory pattern ensures that only one instance of the Device object is created throughout the application, providing a centralized way to retrieve the Device instance.

### Class Attributes

*\_instance: Optional [Device]*

A class-level attribute that holds the singleton instance of the Device class. It is initialized to None and will store the first instance of the Device created by the factory.

---



## Static Methods

*get\_device (device\_model: DeviceModel, serial\_comm: SerialCommunication, device\_config: DeviceConfig, logger: Optional [Logger] = None) -> Device*

Retrieves the singleton instance of the Device class. If the Device instance does not exist, it creates a new one using the provided DeviceModel, SerialCommunication, DeviceConfig, and optional Logger instances. Once created, the instance is reused for subsequent calls.

- **Parameters:**
  - device\_model (DeviceModel): The model used to manage device parameters and commands.
  - serial\_comm (SerialCommunication): The serial communication handler for communicating with the device.
  - device\_config (DeviceConfig): The configuration handler for loading device settings.
  - logger (Optional [Logger]): An optional logger instance for recording device events and errors.
- **Returns:**
  - The singleton Device instance.
- **Behavior:**
  - If the \_instance attribute is None, a new Device instance is created using the provided arguments.
  - If the \_instance attribute already contains a Device instance, that instance is returned.

---

## SerialCommunication Class

The SerialCommunication class handles communication over a serial port. It provides methods to connect, send commands, and receive responses from a device. The class also supports serial port configuration, sending both string and hexadecimal commands, and reading responses.

### Constructor

*\_\_init\_\_ (self, port=None)*

Initializes the SerialCommunication instance, setting the default baud rate, timeout, and optional port.

- **Parameters:**

- port (str, optional): The serial port to connect to (e.g., "COM3").

---

## Public Methods

### *connect(self)*

Establishes a connection to the specified serial port. Raises an error if the port is not set or if the connection fails.

- **Raises:**
  - ValueError: If the serial port is not specified.
  - ConnectionError: If the connection fails.

### *disconnect(self)*

Closes the serial port connection if it is open.

### *send\_command (self, command: str, value: int)*

Sends a formatted command with a value to the device. Calls the internal method `__send__()` to handle the actual transmission.

- **Parameters:**
  - command (str): The command to send.
  - value (int): The value to accompany the command.

### *send\_hex\_command (self, command: str, value: int)*

Sends a hexadecimal command to the device. This method formats the command string and sends it to the device.

- **Parameters:**
  - command (str): The command to send.
  - value (int): The value to send as part of the command.

### *receive\_response (self, command: str)*

Receives a response from the device for a specific command. Calls the internal method `__receive__()` to handle the response.

- **Parameters:**
  - command (str): The command to receive a response for.
- **Returns:**
  - The parsed response as an integer.

*set\_timeout(self, timeout: int)*

Sets the timeout value for the serial communication.

- **Parameters:**
  - timeout (int): The timeout value in seconds.

*check\_connection\_status(self) -> bool*

Checks if the serial connection is active by sending a simple command and waiting for a response.

- **Returns:**
  - True if the connection is active and the device responds.
  - False if the connection is not active or the device fails to respond.
- **Raises:**
  - IOError: If an error occurs while checking the connection status.

*is\_connected(self) -> bool*

Checks whether the serial connection is open.

- **Returns:**
  - True if the connection is open, otherwise False.

*\_\_repr\_\_(self) -> str*

Returns a string representation of the SerialCommunication object, including the port, baud rate, timeout, and connection status.

---

## Private Methods

*\_\_send\_\_(self, command: str, value: int)*

Sends a command with a value to the connected device. The value is formatted as a hexadecimal string and sent to the device.

- **Parameters:**
  - command (str): The command to send.
  - value (int): The value to accompany the command.
- **Raises:**
  - ConnectionError: If there is no active serial connection.

*\_\_receive\_\_(self, command: str) -> int*

Sends a request for a response and waits for the device to respond. The response is then parsed and returned as an integer.

- **Parameters:**
  - `command (str)`: The command to receive a response for.
- **Returns:**
  - The parsed response as an integer.
- **Raises:**
  - `ConnectionError`: If there is no active serial connection.
  - `ValueError`: If the response is unexpected or invalid.

*`__write (self, data: str, delay: float)`*

Writes a command to the device and waits for a specified delay. This is a helper method to manage command transmission timing.

- **Parameters:**
  - `data (str)`: The data to send.
  - `delay (float)`: The delay (in seconds) to wait after sending the command.

*`read_response (self, timeout: int = None) -> str`*

Reads the response from the device. The method waits until the response terminates with a carriage return (`\r`) or until the timeout is reached.

- **Parameters:**
  - `timeout (int, optional)`: The timeout value for reading the response (default: class-level timeout).
- **Returns:**
  - The response string, or `None` if no response is received within the timeout.
- **Raises:**
  - `IOError`: If an error occurs while reading the response.

*`parse_response (self, response: str) -> int`*

Parses the device's response and converts it to an integer. If the response contains an error code, the method raises a `ValueError`.

- **Parameters:**
  - `response (str)`: The response string received from the device.
- **Returns:**
  - The parsed response as an integer.
- **Raises:**
  - `ValueError`: If the response contains an error or is otherwise invalid.

---

## DeviceModel Class

The DeviceModel class manages device parameters and state commands. It loads configuration data, converts it into Command instances, and provides methods to set and get parameter values. It also handles signed and unsigned values and supports error code management.

### Constructor

*`__init__(self)`*

Initializes the DeviceModel with empty dictionaries for parameters, state commands, and error codes.

- **Attributes:**
  - parameters (Dict [str, Command]): A dictionary of device parameters.
  - state\_commands (Dict [str, Command]): A dictionary of device state commands.
  - error\_codes (Dict [str, str]): A dictionary of error codes and their descriptions.

---

### Public Methods

*`load_parameters (self, config: Dict) -> List [Command]`*

Loads parameters from the given configuration dictionary and returns a list of Command instances.

- **Parameters:**
  - config (Dict): A dictionary representing the configuration data.
- **Returns:**
  - A list of Command instances created from the configuration.

*`has_parameter (self, name: str) -> bool`*

Checks if the given parameter exists in either the parameters or state\_commands dictionary.

- **Parameters:**
  - name (str): The name of the parameter.
- **Returns:**
  - True if the parameter exists, otherwise False.

*set\_value (self, name: str, paramtype: str, value: float)*

Sets the value of a specific parameter by name and parameter type, performing necessary calculations and handling signed values if applicable.

- **Parameters:**
  - name (str): The name of the parameter.
  - paramtype (str): The type of the parameter (e.g., "code").
  - value (float): The value to set for the parameter.
- **Returns:**
  - A tuple of (command\_code, final\_value) where:
    - command\_code (str): The code of the command to send.
    - final\_value (int): The final value after calculations.
- **Raises:**
  - ValueError: If the parameter is not available.

*get\_value (self, name: str, paramtype: str) -> Optional[str]*

Retrieves the command code for a specific parameter by name and type.

- **Parameters:**
  - name (str): The name of the parameter.
  - paramtype (str): The type of the parameter (e.g., "code").
- **Returns:**
  - The command code if available, otherwise None.

---

## Private Methods

*\_\_load\_parameters (self, config: Dict) -> List [Command]*

Loads parameters from the configuration into Command instances, clearing previous data.

- **Parameters:**
  - config (Dict): The configuration data as a dictionary.
- **Returns:**
  - A list of Command instances.

*\_load\_command\_parameters (self, items: Dict, dataclass\_objects: List [Command])*

Helper method that loads command parameters into Command instances and stores them in self.parameters.

- **Parameters:**
  - items (Dict): The dictionary of parameters.

- `dataclass_objects` (List [Command]): A list of Command instances to append to.

*`_load_error_codes (self, items: List [Dict [str, str]])`*

Helper method that loads error codes into `self.error_codes`.

- **Parameters:**

- `items` (List [Dict [str, str]]): The list of error codes.

*`_load_state_commands (self, items: Dict, dataclass_objects: List [Command])`*

Helper method that loads state commands into Command instances and stores them in `self.state_commands`.

- **Parameters:**

- `items` (Dict): The dictionary of state commands.
- `dataclass_objects` (List [Command]): A list of Command instances to append to.

*`__set_value (self, name: str, paramtype: str, value: float)`*

Sets a parameter's value, performing scaling with dividers and handling signed/unsigned conversions if applicable.

- **Parameters:**

- `name` (str): The name of the parameter.
- `paramtype` (str): The type of the parameter (e.g., "code").
- `value` (float): The value to set for the parameter.

- **Returns:**

- A tuple of (`command_code`, `final_value`).

*`__get_value (self, name: str, paramtype: str) -> Optional[str]`*

Retrieves the command code for a given parameter and type.

- **Parameters:**

- `name` (str): The name of the parameter.
- `paramtype` (str): The type of the parameter (e.g., "code").

- **Returns:**

- The command code if available, otherwise `None`.

*`_find_command_code (self, name: str, paramtype: str) -> Optional[str]`*

Finds the command code for a specific parameter by name and type.

- **Parameters:**

- name (str): The name of the parameter.
- paramtype (str): The type of the parameter (e.g., "code").
- **Returns:**
  - The command code if available, otherwise None.

*\_get\_divider (self, name: str, paramtype: str) -> Optional[int]*

Retrieves the divider for a specific parameter, or defaults to 1.0 if not specified.

- **Parameters:**
  - name (str): The name of the parameter.
  - paramtype (str): The type of the parameter (e.g., "measured").
- **Returns:**
  - The divider for the parameter.

*\_parse\_hex\_value (self, hex\_value: Optional[str]) -> Optional[float]*

Parses a hexadecimal value and converts it to a float.

- **Parameters:**
  - hex\_value (Optional[str]): The hexadecimal string to convert.
- **Returns:**
  - The parsed value as a float, or None if no value is provided.

*\_convert\_to\_signed\_hex (self, value: float) -> int*

Converts a decimal value to a 16-bit two's complement signed integer.

- **Parameters:**
  - value (float): The decimal value to convert.
- **Returns:**
  - The value as a signed integer.
- **Raises:**
  - ValueError: If the value is out of range for a 16-bit signed integer.

*\_convert\_signed\_value (self, value: int) -> int*

Converts a 16-bit two's complement signed integer to a regular signed integer.

- **Parameters:**
  - value (int): The value to convert.
- **Returns:**
  - The converted value as an integer.

*\_get\_isSigned (self, name: str) -> Optional[bool]*



Checks whether a parameter is signed.

- **Parameters:**
  - `name (str)`: The name of the parameter.
- **Returns:**
  - True if the parameter is signed, otherwise False.

*`get_units (self, name: str) -> Optional[str]`*

Retrieves the units for a specific parameter.

- **Parameters:**
  - `name (str)`: The name of the parameter.
- **Returns:**
  - The units of the parameter if available, otherwise None.

*`get_error_description (self, error_code: str) -> str`*

Retrieves the description for a given error code.

- **Parameters:**
  - `error_code (str)`: The error code.
- **Returns:**
  - The description of the error, or "Unknown error code" if the code is not found.

---

## Configuration Sections

The following sections are processed from the configuration dictionary:

- **SECTION\_ERROR\_CODES**: Represents error codes for the device.
- **SECTION\_PARAMETERS**: Represents general parameters for the device.
- **SECTION\_STATE\_COMMANDS**: Represents state commands for the device.
- **SECTION\_TEC\_COMMANDS**: Represents TEC (Thermo-Electric Cooler) related commands.
- **SECTION\_TEC\_STATE\_COMMANDS**: Represents state commands for TEC operations.
- **SECTION\_DEVICE\_INFO**: Contains device information such as version or serial number.
- **SECTION\_SYSTEM\_COMMANDS**: Represents system-level commands for the device.

---

## DeviceConfig Class

The `DeviceConfig` class manages the loading, validation, and retrieval of device configurations from a YAML file. It provides properties to access specific sections of the configuration, such as

error codes, parameters, and commands, and includes methods to validate the structure and integrity of the configuration.

---

## Constructor

*\_\_init\_\_(self)*

Initializes the DeviceConfig object with the default file path for the configuration file.

- **Attributes:**
    - file\_path (str): The path to the YAML configuration file.
    - config (Optional[dict]): The entire configuration loaded from the YAML file.
    - device\_config (Optional[dict]): The configuration specific to a particular device, loaded based on the device ID.
- 

## Public Methods

*load(self, device\_id: int) -> dict*

Loads the configuration for a specific device by ID and validates it.

- **Parameters:**
  - device\_id (int): The ID of the device whose configuration is to be loaded.
- **Returns:**
  - A dictionary representing the device's configuration.
- **Raises:**
  - FileNotFoundError: If the YAML configuration file is not found.
  - ValueError: If the device ID is not found in the configuration or if the configuration is invalid.

*\_\_repr\_\_(self) -> str*

Generates a string representation of the DeviceConfig object, showing the file path and a snippet of the configuration.

- **Returns:**
    - A string representation of the DeviceConfig instance.
-

## Private Methods

*\_load\_yaml(self) -> None*

Loads the YAML configuration file into the `config` attribute.

- **Raises:**
  - `FileNotFoundError`: If the YAML configuration file is not found.
  - `ValueError`: If there is an error in parsing the YAML file.

*\_validate\_device\_config (self, device\_config: dict) -> None*

Validates the structure of the device configuration, ensuring it contains required sections and that parameters have valid codes.

- **Parameters:**
  - `device_config` (dict): The configuration dictionary for a specific device.
- **Raises:**
  - `ValueError`: If required sections or parameter codes are missing.

*\_get\_section (self, section\_name: str) -> dict*

Helper method to retrieve a specific section of the device configuration.

- **Parameters:**
  - `section_name` (str): The name of the section to retrieve (e.g., "error\_codes", "parameters").
- **Returns:**
  - A dictionary representing the section of the configuration.
- **Raises:**
  - `ValueError`: If the device configuration has not been loaded.

---

## Properties

*error\_codes(self) -> list*

Retrieves the error codes section from the device configuration.

- **Returns:**
  - A list of error codes from the device configuration.

*parameters(self) -> dict*

Retrieves the parameters section from the device configuration.

- **Returns:**
  - A dictionary of parameters from the device configuration.

*state\_commands(self) -> dict*

Retrieves the state commands section from the device configuration.

- **Returns:**
  - A dictionary of state commands from the device configuration.

*tec\_commands(self) -> dict*

Retrieves the TEC (Thermo-Electric Cooler) commands section from the device configuration.

- **Returns:**
  - A dictionary of TEC commands from the device configuration.

*tec\_state\_commands(self) -> dict*

Retrieves the TEC state commands section from the device configuration.

- **Returns:**
  - A dictionary of TEC state commands from the device configuration.

*device\_info(self) -> dict*

Retrieves the device information section from the device configuration.

- **Returns:**
  - A dictionary of device information from the device configuration.

*system\_commands(self) -> dict*

Retrieves the system commands section from the device configuration.

- **Returns:**
  - A dictionary of system commands from the device configuration.

---

## YAML File Structure

A typical YAML file is structured into sections based on the device's features and functionality.

## Devices

The root of the YAML file contains a `devices` section, which lists devices by their unique ID. Each device entry includes:

- `name`: A human-readable name for the device.
- `id`: The device's unique identifier.
- `error_codes`, `parameters`, `state_commands`, `tec_commands`, and other sections for device settings.

## Error Codes

The `error_codes` section lists various error conditions the device might encounter. Each error includes:

- `name`: A human-readable name for the error.
- `code`: The code returned by the device when the error occurs.
- `description`: A brief explanation of the error.

*error\_codes:*

```
- name: "internal buffer overflow"
  code: "E0000"
  description: "Internal buffer overflow or command format invalid."
```

## Parameters

The `parameters` section defines device parameters that can be controlled or measured. Each parameter includes:

- `name`: The name of the parameter.
- `code`: The code used to set or read the parameter.
- `min` and `max`: The minimum and maximum allowable values (often represented by codes).
- `divider`: A scaling factor used to convert the device's internal value to a human-readable form.
- `units`: The units of the parameter (e.g., "Hz", "mA").
- `measured`: The code for retrieving the measured value of the parameter.

*parameters:*

```
current:
  name: "current"
  code: "0300"
  min: "0301"
  max: "0302"
  divider: 10
```

```
units: "mA"  
measured: "0307"
```

### State Commands

The `state_commands` section manages device state transitions, such as enabling or starting operations. Each state command includes:

- name: The name of the state.
- code: The code used to read or modify the state.
- bits: A set of bit definitions within the state, each bit representing a sub-state of the device. Each bit includes:
  - name: A human-readable name for the bit.
  - on\_command and off\_command: Commands to turn the state on or off.
  - mask: A bitmask used to identify the state.
  - states: The possible values and their corresponding meaning.

```
state_commands:  
state_of_device:  
  name: "ofDevice"  
  code: "0700"  
  bits:  
    "1":  
      name: "operation state"  
      on_command: "0008"  
      off_command: "0010"  
      mask: "0002"  
      states:  
        "0": "Stopped"  
        "1": "Started"
```

### TEC Commands

The `tec_commands` section defines the parameters and state management for the Thermo-Electric Cooler (TEC) component of the device. This includes temperature, TEC current, and voltage settings. Each TEC command includes:

- name: The name of the TEC command.
- measured: The code for retrieving the measured value.
- limit: The code for setting limits.
- divider: A scaling factor used to convert the internal value to a human-readable form.
- isSigned: A boolean indicating whether the value is a signed integer.

```
tec_commands:  
tec_current:  
  name: "tecCurrent"
```

```
measured: "0A16"  
limit: "0A17"  
divider: 10  
units: "A"  
isSigned: true
```

## System Commands

The `system_commands` section contains commands related to system-level operations, such as saving device parameters or resetting the device.

```
system:  
  save_parameters:  
    name: "saveParameters"  
    code: "0900"
```

---

## Command Dataclass

The Command dataclass is a flexible representation of a command or parameter used by a device in the DeviceControl system. It stores essential information about commands, including codes, limits, descriptions, and optional details such as measurement scaling and bitmasks for complex state commands.

### Fields

*name: str*

The human-readable name of the command or parameter.

- **Default:** ""

*code: str*

The command code, usually a hexadecimal string, which is sent to or received from the device.

- **Default:** ""

*min: str*

The minimum value code for the parameter (if applicable).

- **Default:** ""

*max: str*

The maximum value code for the parameter (if applicable).

- **Default:** ""

*value: float*

The current value of the command, which is either set or read from the device.

- **Default:** 0.0

*description: str*

A brief description of the command or its function.

- **Default:** ""

*divider: int*

A scaling factor applied to the command's value to convert it to or from a more human-readable form. For example, a value of 1000 with a divider of 10 would represent 100.0.

- **Default:** 1

*isSigned: bool*

Indicates whether the command's value is a signed integer (useful for commands representing temperatures or other measurements that can be negative).

- **Default:** False

*units: str*

The units associated with the command (e.g., "mA", "Hz", "V").

- **Default:** ""

*properties: List[str]*

A list of additional properties associated with the command. This can store any additional details such as flags or characteristics.

- **Default:** [] (empty list)



## Additional Fields for Specific Command Types

These fields are optional and only relevant for specific types of commands:

*min\_value: Optional[float]*

The minimum allowable value for the command, expressed as a floating-point number. Used when the command has a specified numeric range.

*max\_value: Optional[float]*

The maximum allowable value for the command, expressed as a floating-point number.

*protection\_threshold: Optional[float]*

The protection threshold value for parameters that require protection settings (e.g., current or voltage protection).

*measured: Optional[str]*

The code representing the measured value of the parameter. This is used when the command involves reading a live or measured value from the device.

*divider\_measured: Optional[int]*

A scaling factor applied to measured values. This is similar to divider, but specifically for measured data.

*bits: Optional[Dict[str, Dict[str, str]]]*

A dictionary of bit-level details for state commands. Each bit has its own name, mask, and possible states (e.g., "0": "Disabled", "1": "Enabled").

*lower\_limit: Optional[float]*

The lower limit for the command's value.

*upper\_limit: Optional[float]*

The upper limit for the command's value.

*limit: Optional[str]*

A string representing a command limit (e.g., "max current limit").

*max\_limit: Optional[str]*

The maximum limit code for commands that involve an adjustable limit.

*min\_limit: Optional[str]*

The minimum limit code for commands that involve an adjustable limit.

## DeviceStates Enum

The DeviceStates enum represents the possible states of the device. Each state is mapped to a specific string value that corresponds to a command or device response.

## Enum Values

- **OPERATION\_STATE\_STARTED**: Represents the state where the device has started its operation ("1").
  - **CURRENT\_SET\_INTERNAL**: Represents the internal current setting state ("2").
  - **ENABLE\_INTERNAL**: Represents the internal enable state ("4").
  - **EXTERNAL\_NTC\_INTERLOCK\_DENIED**: Represents the state where an external NTC interlock is denied ("6").
  - **INTERLOCK\_DENIED**: Represents the state where an interlock is denied ("7").
- 

## Logger Class

The Logger class provides an asynchronous logging mechanism for recording events and errors. It supports log rotation, UTF-8 encoding, and configurable logging levels. The class processes log asynchronously using a background thread and a queue.

## Constructor

*\_\_init\_\_ (self, log\_file: str, max\_file\_size: int = 1024\*100, backup\_count: int = 5, enable\_logging: bool = True)*

Initializes the Logger class and sets up the logging configuration.

- **Parameters:**
  - log\_file (str): The file path where logs will be stored.
  - max\_file\_size (int): The maximum size of the log file before rotating (default: 100 KB).
  - backup\_count (int): The number of backup log files to maintain (default: 5).
  - enable\_logging (bool): Whether logging is enabled (default: True).
- **Attributes:**
  - log\_file (str): The path to the log file.

- `enable_logging` (bool): Flag to control whether logging is active.
- `max_file_size` (int): The maximum size of a single log file in bytes.
- `backup_count` (int): The number of backup log files to retain.
- `log_queue` (Queue): A queue used for asynchronous log event processing.
- `logger` (logging.Logger): The logger instance that manages file handling and formatting.
- **Log Rotation:**
  - The logger uses a `RotatingFileHandler` to automatically rotate log files when they reach the specified `max_file_size`. Backup log files are maintained based on `backup_count`.

---

## Public Methods

*`eventRecording(self, event: LogEvent)`*

Asynchronously records an event by placing it in the log queue for processing.

- **Parameters:**
  - `event` (LogEvent): The log event to be recorded.

*`errorRecording(self, message: str, error_code: Optional[int] = None)`*

Asynchronously records an error event with an optional error code.

- **Parameters:**
  - `message` (str): The error message to log.
  - `error_code` (Optional[int]): An optional error code associated with the error.

*`stop(self)`*

Stops the logging thread by sending a stop signal to the log queue.

*`start(self)`*

Restarts the logging thread if it was previously stopped.

*`path(self, new_path: str)`*

Updates the log file path and reinitializes the file handler with the new path.

- **Parameters:**
  - `new_path` (str): The new file path for the log file.

## Private Methods

### *\_process\_log\_queue(self)*

Processes log events from the queue asynchronously. This method runs in a background thread and handles log events as they arrive.

### *\_add\_log\_separator(self)*

Adds a separator and timestamp to the log file at the start of each run. This makes it easier to distinguish between different application sessions.

### *\_ensure\_utf8(self, message: str) -> str*

Ensures that a log message is properly encoded in UTF-8. If encoding fails, it falls back to ASCII.

- **Parameters:**
  - message (str): The log message to encode.
- **Returns:**
  - The encoded message as a UTF-8 string, or ASCII if encoding fails.

---

## LogEvent Dataclass

The LogEvent dataclass represents a single logging event, containing the log level, message, and optional event data.

- **Fields:**
  - level (str): The logging level (e.g., "INFO", "ERROR", "DEBUG").
  - message (str): The message to log.
  - event\_data (Optional[dict]): Optional additional data for the log event.

## Exception: DeviceError

The DeviceError class is a custom exception that represents a device-related error. It stores the error code and description.

### Constructor

#### *\_\_init\_\_(self, code: str, description: str)*

Initializes the DeviceError exception.

- **Parameters:**

- code (str): The error code.
- description (str): A description of the error.
- **Attributes:**
  - code (str): The error code associated with the exception.
  - description (str): A human-readable description of the error.

## Example of Usage

This section provides a practical example of how to use the Laser Diode Driver library. The script below demonstrates how to initialize a device, configure it, and retrieve various parameters such as frequency, duration, current, and voltage.

## Running the Test Script

Ensure that the required dependencies are installed:

```
pip install -r requirements.txt
```

### *Example 1: Initializing the Device*

```
# Initialize necessary objects for the Device class
logger = Logger (r'C: \path_to_log\device_operations.log')
device_config = DeviceConfig ()
serial_comm = SerialCommunication("COM4")
device_model = DeviceModel ()

# Initialize the Device instance
device = Device (device_model, serial_comm, device_config, logger)
device.initialize_device ()
```

**Explanation:** This snippet demonstrates how to initialize the device with Logger, DeviceConfig, SerialCommunication, and DeviceModel objects. The device is then initialized using the initialize\_device () method, which sets up the configuration and device parameters.

---

### *Example 2: Setting and Getting Device Parameters*

```
# Set and get frequency
device.set_frequency_code (180.6)
frequency = device.get_frequency_code ()
print (f"Frequency Code: {frequency}")

# Set and get current
device.set_current_code (1234.897)
current = device.get_current_code ()
print (f"Current Code: {current}")
```

**Explanation:** This section shows how to set and retrieve specific parameters such as frequency and current using dynamically generated methods (set\_frequency\_code, get\_frequency\_code).

---

### *Example 3: Managing Device States*

```
# Set and retrieve device state
device_state = device.getState_ofDevice ()
print (f"Device State: {device_state}")

# Set new state values
state_values = {
    "Operation state": "Stopped",
    "Current set": "External",
    "enable": "Internal",
    "External NTC interlock": "Denied",
    "interlock": "Denied"
}
device.setState_ofDevice(state_values)
```

**Explanation:** This example illustrates how to retrieve the device's current state and set new state values for various operations (e.g., operation state, current set).

---

### *Example 4: Retrieving Device Status*

```
# Check raw device status
raw_status_device = device.get_raw_status("ofDevice")
print (f"Device Raw Status: {raw_status_device:04X}")

# Check device status summary
status = device.check_device_status ()
print (f"Device Status: {status}")
```

**Explanation:** This snippet demonstrates how to check the raw device status using get\_raw\_status () and how to check the device's overall status using check\_device\_status ().

---

## List of methods used in the Script

1. **Device Initialization and Setup:**
  - initialize\_device ()
2. **Setting and Getting Parameters:**
  - set\_frequency\_code (value: float)
  - get\_frequency\_code () -> float
  - set\_current\_code (value: float)
  - get\_current\_code () -> float

- `set_duration_code (value: float)`
- `get_duration_code () -> float`
- `get_voltage_measured () -> float`
- `set_ntcTemperature_min (value: float)`
- `get_ntcTemperature_min () -> float`
- `set_ntcTemperature_max (value: float)`
- `get_ntcTemperature_max () -> float`
- `get_ntcTemperature_measured () -> float`
- `set_currentCalibration_code(value: float)`
- `get_currentCalibration_code() -> float`
- `get_current_min() -> float`
- `get_current_max() -> float`
- `get_current_measured() -> float`
- `get_current_protection_threshold() -> float`
- `get_serialNumber_code() -> int`
- `get_deviceId_code() -> int`
- `get_saveParameters_code() -> str`
- `get_resetParameters_code() -> str`
- `set_temperature_code(value: float)`
- `get_temperature_code() -> float`
- `get_temperature_min() -> float`
- `set_temperature_max(value: float)`
- `get_temperature_max() -> float`
- `get_temperature_max_limit() -> float`
- `get_temperature_min_limit() -> float`
- `get_temperature_measured() -> float`
- `get_tecCurrent_measured() -> float`
- `set_tecCurrent_limit(value: float)`
- `get_tecCurrent_limit() -> float`
- `get_tecVoltage_measured() -> float`
- `set_tecCurrentCalibration_code(value: float)`
- `get_tecCurrentCalibration_code() -> float`
- `set_internalLdNtcSensor_code(value: float)`
- `get_internalLdNtcSensor_code() -> float`
- `set_pCoefficient_code(value: float)`
- `get_pCoefficient_code() -> float`
- `set_iCoefficient_code(value: float)`
- `get_iCoefficient_code() -> float`
- `set_dCoefficient_code(value: float)`
- `get_dCoefficient_code() -> float`

### 3. Managing Device State:

- `getState_ofDevice() -> Dict[str, str]`
- `setState_ofDevice(state_values: Dict[str, str])`
- `getState_lockStatus() -> Dict[str, str]`
- `getState_tecState() -> Dict[str, str]`

### 4. Device Status and Raw Status:

- `get_raw_status(state_name: str) -> int`
- `check_device_status() -> Dict[str, str]`

### 5. Utility Methods:

- `disconnect()`

## Expected Output (example)

When running the script, you can expect output similar to the following:

```
Frequency Code: 180.6  
Frequency Min: 10.0  
Frequency Max: 200.0  
Duration Code: 6.98  
Duration Min: 1.0  
Duration Max: 10.0  
Current Code: 1234.897  
Current Min: 100.0  
Current Max: 1500.0  
Voltage Measured: 24.0 V  
Device State: {'operation state': 'Stopped', 'current set': 'Internal', 'enable': 'Internal'}  
Device Raw Status: 0700
```

## Creating Custom Methods

The Device class supports dynamic method creation for getting and setting parameters. To retrieve or set new parameters, methods can be created in the following format:

- To **set a parameter**: `set_<parameter>_code(<value>)`
- To **get a parameter**: `get_<parameter>_code()`
- Example: `set_frequency_code(100)` OR `get_frequency_code()`

This flexibility allows users to easily extend the functionality of the library without modifying the internal code.