

# OnLogic M031 Manager Documentation

version 0.0.1

[nick.hanna@onlogic.com](mailto:nick.hanna@onlogic.com), [fwengineeringteam@onlogic.com](mailto:fwengineeringteam@onlogic.com)

May 29, 2025



# Contents

<b>OnLogic M031 Manager Documentation</b>	<b>1</b>
OnLogicM031Manager Overview	1
Setup Required	1
<b>Examples</b>	<b>2</b>
<b>Shell Transport Protocol</b>	<b>2</b>
<b>Automotive Timings</b>	<b>3</b>
<b>Digital Input/Output Module (DIO) Functionality</b>	<b>5</b>
Source Code Documentation	7
src Module	7
Examples	8
src.onlogic_m031_manager Module	22
src.dio_handler Module	24
src.automotive_handler Module	28
src.command_set Module	36
src.logging_util Module	38
<b>Indices and tables</b>	<b>40</b>
<b>Index</b>	<b>41</b>
<b>Python Module Index</b>	<b>45</b>



# OnLogic M031 Manager Documentation

## OnLogicM031Manager Overview

The OnLogicM031Manager provides a set of tools to interface with peripherals on Onlogic K/HX-52x series computers.

- On the HX-52x, it can send commands to the DIO add-in-card.
- On the K-52x, it can send commands both to the DIO add-in-card and the sequence microcontroller to control automotive timings.

## Setup Required

Python 3 must be installed prior to following this guide. Python 3 can be installed from python.org. Ensure Python and pip are added to the system's PATH during installation.

## Setting up OnLogicM031Manager on Windows (Native Install)

These steps serve as a guide for installing the OnLogicM031Manager directly into the local Windows Python environment.

1. Open Command Prompt or PowerShell: "cmd" or "powershell" are shorthands that can be entered in the Start Menu to pull them up.
2. Clone Project and Navigate to the Project Directory: In the parent directory where the project is to be run, run the following command in powershell to clone the repository

```
git clone git@github.com:onlogic/onlogic-m031-manager.git
```

or download the directory clicking the green button and then Download ZIP on the top right of the GitHub page.

Use the `cd` command to change to the directory where the OnLogicM031Manager files are located (same directory as where the `setup.py` file is located). Example:

```
cd path\to\onlogic-m031-manager
```

3. Install Required Packages: Run the following command to install the package. This will install it into the global Python site-packages or user-specific site-packages:

```
pip install -e .
```

5. Verify Installation: To see all installed Python packages in the global environment (this will include packages beyond this project):

```
pip freeze
```

6. Running Scripts Requiring Elevated Privileges: For operations that require direct hardware access (like interacting with serial ports), the Python scripts may need to be run from a Command Prompt or PowerShell that has been opened "As Administrator". To do this, right-click on the Command Prompt/PowerShell icon and select "Run as administrator".

## Setting up OnLogicM031Manager in a Python3 venv on Ubuntu 24.04 LTS

Linux Ubuntu has enforced a stricter package management scheme in the new 24.04 LTS distribution to avoid interfering with global package dependencies used by the OS. While this is a more stable way to administer Python on a system, it is also more complex to program in user environments. To run the OnLogicM031Manager package in Ubuntu, it's best practice to use a venv. Download the OnLogicM031Manager package by following step 2 above.

- Creating a venv:

```
$ python3 -m venv <path/to/venv>
```

One can get <path/to/venv> by navigating to desired directory in terminal and inputting `pwd`,

## Examples

```
python3 -m venv venv
```

Will likely work as well

- Activating a venv:

```
$ source <path/to/venv>/bin/activate
```

- Deactivating a venv:

```
$ deactivate
```

- When the venv is activated, running any python scripts will use the venv's interpreter and packages. But, when running a script that needs root privileges (`sudo python . . .`), the venv's Python won't be used, even if it's activated. One solution is to explicitly use the venv's interpreter when running a Python script:

```
$ sudo <path/to/venv>/bin/python somescript.py
```

- The whole path must be used to sudo in, and IO cannot be accessed without sudo privileges

After, set up required packages in venv:

- `pip install -e .`

- Verify with: `pip freeze` within local directory

## Examples

There are several examples in the `examples` directory. The examples are designed to run from the command line and follow the setup seen above. Make sure, however, that for Automotive settings, COM visibility is enabled within the BIOS.

The examples are designed to be run from the command line with:

```
sudo /path/to/project/bin/python3 dio_implementation.py
```

for the `dio_implementation.py` script in Ubuntu, for example.

## Shell Transport Protocol

A protocol is used for transferring commands. By convention, the CPU issues commands, and the MCU listens and responds to them. Each valid command message generates a response message.

Each message consists of a 4-byte header: a fixed `0x01` start byte, the CRC-8 of the message, the length of any data following the header, and the kind of message. The CRC-8 is calculated from the third byte of the message (the length byte) onward, and uses the SMBUS polynomial (`0x107`).

A primitive form of flow control is built into the protocol. After a byte is received, the receiver processes it and replies with `\r` if the byte was expected, or `\a` if not. An example command/response sequence might look like this:

CPU		MCU
(start of frame)	0x01 ->	
	<- (acknowledge)	\r
(crc-8)	0x38 ->	
	<- (acknowledge)	\r
(data length)	0x00 ->	
	<- (acknowledge)	\r
(message kind)	0x08 ->	
	<- (acknowledge)	\r
<MCU processes command>		
	<- (start of frame)	0x01
(acknowledge)	\r ->	
	<- (crc-8)	0xc4
(acknowledge)	\r ->	
	<- (data length)	0x01
(acknowledge)	\r ->	

```

                                <- (message kind)    0x08
(acknowledge)    \r    ->
                                <- (data byte)       0x01
(acknowledge)    \r    ->

```

This sequence shows the CPU sending a `kGet_LowPowerEnable` message with no additional data and the MCU responding with a `kGet_LowPowerEnable` response with one byte of additional data.

This Python Module administers this protocol in communication with both DIO and Sequence microcontrollers. It makes native Python datatypes, converts them to byte compatible communication, and administers this process with additional type and value checking.

**Note** the CPU uses two distinct communication protocols to talk with the DIO and Sequence Microcontrollers. 1. CDC-USB with the DIO Card 2. UART with the Sequence Micro

For this reason, the user must manually specify the serial port name for the sequence micro `.claim()` method in the `AutomotiveManager` class, whereas for the `DioHandler`, the `.claim()` method can be left blank and the program will autolock on the serial connection label.

The status types are defined in `src/command_set.py` and are used to mark and indicate failures during different stages of the LPMCU protocol.

The table below is a summary of the status types, but note that method class members do not all report the status types in the same way.

Status Type	Value	Description
<i>SUCCESS</i>	0	The LPMCU protocol completed successfully.
<i>SEND_CMD_FAILURE</i>	-1	Failed to send the command during the initial transmission process.
<i>RECV_UNEXPECTED_PAYLOAD_ERROR</i>	-2	The received payload did not match the expected format or structure during validation.
<i>RECV_FRAME_CRC_ERROR</i>	-3	The CRC value of the received frame did not match the expected value, indicating corruption.
<i>RECV_FRAME_ACK_ERROR</i>	-4	The acknowledgment frame validation failed, indicating an issue with the tail frame.
<i>RECV_FRAME_SOF_ERROR</i>	-5	The start-of-frame (SOF) byte <code>0x01</code> was not found in the received frame.
<i>RECV_PARTIAL_FRAME_VALIDATION_ERROR</i>	-6	Validation of a partially received frame failed, indicating incomplete or corrupted data.
<i>RECV_FRAME_VALUE_ERROR</i>	-7	The received payload contained unexpected or invalid values.
<i>FORMAT_NONE_ERROR</i>	-8	A <i>None</i> value was encountered during type formatting, indicating a missing or invalid type.

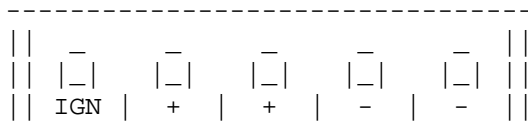
## Automotive Timings

The ignition sense feature can be used to turn OnLogic K52x units on and off with a battery, a vehicle's ignition (given proper electrical setup), or the use of a switch. Automotive timings and ignition sense can be toggled by bridging DC power to the IGN pin, see the diagram below. The proper sequence microcontroller settings must be set before using either the BIOS settings, LPMCU tool, or this Python API.

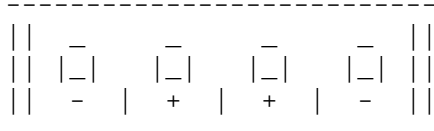
The unit will turn on when power is applied to the IGN pin, and turn off when power is removed.

Ignition sensing can be enabled and adjusted through a UART connection to the system's microcontroller. This program administers the LPMCU communication protocol to the sequence microcontroller to allow control over various timers and values detailed in the Command Summary table below.

**Terminal Block Diagram of K-52x - Ignition pin is on leftmost side**



### Terminal Block Diagram of HX-52x



Note: The HX-52x does not have an ignition pin. Therefore, automotive mode is not supported.

## Command Summary

Command	Description	Parameters	Returns
get_automotive_mode	Get the automotive mode of the device. Determine if system automotive features are enabled.	None	(0:low, 1:high)
set_automotive_mode	Set the automotive mode of the device. Enables or disables system automotive features.	(0:low, 1:high)	Status
get_low_power_enable	Get the low power enable status from the MCU. Enables entering a very low power state when the system powers off.	None	(0:low, 1:high)
set_low_power_enable	Set the low power enable status in the MCU. Enables entering a very low power state when the system powers off.	(0:low, 1:high)	Status
get_start_up_timer	Get the start-up timer value from the MCU. Controls the number of seconds the ignition input must be stable before powering on.	None	Integer: Start-up timer value in seconds.
set_start_up_timer	Set the start-up timer value in the MCU. Controls the number of seconds the ignition input must be stable before powering on.	Integer: Start-up timer value in seconds.	Status
get_soft_off_timer	Get the soft-off timer value from the MCU. Controls the number of seconds the ignition input can be low before the MCU requests a power-down event.	None	Integer: Soft-off timer value in seconds.



set_soft_off_timer	Set the soft-off timer value in the MCU. Controls the number of seconds the ignition input can be low before the MCU requests a power-down event.	Integer: Soft-off timer value in seconds.	Status
get_hard_off_timer	Get the hard-off timer value from the MCU. Controls the number of seconds the ignition input can be low before the MCU forces the system to power down.	None	Integer: Hard-off timer value in seconds.
set_hard_off_timer	Set the hard-off timer value in the MCU. Controls the number of seconds the ignition input can be low before the MCU forces the system to power down.	Integer: Hard-off timer value in seconds.	Status
get_shutdown_voltage	Get the shutdown voltage value from the MCU. The threshold voltage for triggering low-voltage shutdown events.	None	Integer: Shutdown voltage in millivolts.
set_shutdown_voltage	Set the shutdown voltage value in the MCU. The threshold voltage for triggering low-voltage shutdown events.	Integer: Shutdown voltage in millivolts.	Status
get_all_automotive_settings	Get all automotive settings from the MCU. Returns a dictionary containing all automotive configurations.	None	Dictionary: All automotive settings.
set_all_automotive_settings	Set all automotive settings in the MCU. Takes a list of values for all settings.	List: [amd, lpe, sut, sot, hot, sdv]	List: Status code of each command.

## Digital Input/Output Module (DIO) Functionality

The DIO module can be configured in two modes: **wet contact** and **dry contact**. The outputs function as open drains. The inputs are high impedance.

### 1. DO Output Configurations:

#### DO Wet Contact Mode (Suitable for Inductive Load Operation)

To function properly, the **V+ pin of the module should be connected to external power and ground**. The **high side of the load** should be connected to the **external power source**, and the **low side to the module DO pin**.

- Load current should not exceed 150 mA.
- Voltage ranges should be 5 V to 30 V.

#### Setup required for Output:

- Vin + GND + Pull up (10kOhm acceptable) + Pin connections

#### DO Dry-Contact Mode

- Voltage is provided by the system. Each DO will output **11 V - 12.6 V when active**.

### Setup required for Output:

- Shared GND + Pin Connections

## 2. Digital Input (DI) Configurations

## DI Wet Contact Mode

There is **no internal pull-up** to the DI[0:7] pins when set to WET mode.

- Externally supplied **5 V - 30 V** is recognized as logic 0.
- Externally supplied **0 V - 3 V** is recognized as logic 1.

### Setup required for Input:

- Vin + GND + Pull up (10kOhm acceptable) + Pin connections

## DI Dry Contact Mode

When the contact type is set to DRY mode, DI[0:7] are **pulled up to the internal isolated ~12V supply**.

- An **open/floating connection** is recognized as logic 0.
- A **short to GND** is recognized as logic 1.

### Setup required for Input:

- Shared GND + Pin Connections

Operations are blocking but can be threaded to accomodate other processing operations, though the DIO card can only retrieve one value at a time over UART.

## Pinout

[illegible]

## Command Summary

**NOTE:** for more information on pin values and their relationship to contact states, please refer to the MCU documentation spec sheet.

Command	Description	Parameters	Returns
<i>get_di</i>	Read digital input pin state	pin val (0-7)	(0, 1)
<i>get_do</i>	Read digital output pin state	pin val (0-7)	(0, 1)
<i>set_do</i>	Set digital output pin state	pin val (0-7)   state (0:low, 1:high)	status
<i>set_all_do</i>	Set all DO pin states	list of desired pin states	status list
<i>get_all_do</i>	Read all DO pin states		list of pin states
<i>get_all_input_states</i>	Read all DI pin states		list of pin states
<i>get_all_output_states</i>	Read all DO pin states		list of pin states
<i>get_all_io_states</i>	Read all DIO pin states		list of pin states
<i>set_all_output_states</i>	Set all DIO pin states	list of 8 pin states	list of pin statuses

### Parameter Summary For Contact Modes

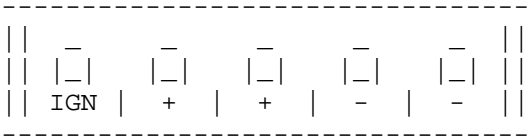
Command	Description	Parameters	Returns
<i>set_di_contact</i>	Set digital input contact type	(0:wet, 1:dry)	

<code>set_do_contact</code>	Set digital output contact type	(0:wet, 1:dry)	
<code>get_di_contact</code>	Read digital input contact type		(0:wet, 1:dry)
<code>get_do_contact</code>	Read digital output contact type		(0:wet, 1:dry)

# Source Code Documentation

## src Module

**class** `src.AutomotiveHandler` (`serial_connection_label=None`)  
Bases: `OnLogicM031Manager`  
AutomotiveHandler class for managing automotive features of the OnLogic Nuvoton MCU.  
This class provides methods to interact with the automotive features of the OnLogic Nuvoton MCU. For more information about the automotive features, please refer to the AutomotiveModeDescription section of the documentation or the docstrings below.  
K52x Ignition Pin Diagram:



**serial\_connection\_label**  
The label of the serial connection.  
**Type:** `str`

### Examples

```
Claim and release port for the Automotive class with either:

my_auto = AutomotiveHandler()

# will be "/dev/ttySX" on Linux or "COMX" on Windows
my_auto.claim("COMX")
# ... (other operations)
my_auto.release()

or

with AutomotiveHandler() as my_auto:
    # ... (operations within the context manager)
    # Port is claimed on entry and released on exit
    pass
```

**\_\_init\_\_** (`serial_connection_label=None`)  
Initialize the AutomotiveHandler class.  
**Parameters:** `serial_connection_label` (`str`) – The label of the serial connection.

**get\_all\_automotive\_settings** (`output_to_console: bool = False`) → dict  
Get all automotive settings from the sequence MCU.  
This method is a wrapper that calls all get automotive attributes and formats them in one dictionary. It provides the option to print the settings to the console for easy viewing, similar to the screen provided in the BIOS settings. This method uses the LPMCU protocol discussed in the README and documentation to get the automotive settings from the sequence MCU.

(Note numpy commenting format was used as this was the only way to get the docstring to render correctly in Sphinx)

## Parameters

**output\_to\_console** : *bool*

If True, print the settings to the console.

## Returns

**automotive\_settings** : *dict*

A dictionary containing the current automotive settings of the device. The keys and their corresponding value types and meanings are:

- 'amd' (int): Current state of Automotive Mode (0 for enabled, 1 for disabled).
- 'lpe' (int): Current state of Low Power Enable (0 for enabled, 1 for disabled).
- 'sut' (int): Current setting for the Start Up Timer (Seconds).
- 'sot' (int): Current setting for the Soft Off Timer (Seconds).
- 'hot' (int): Current setting for the Hard Off Timer (Seconds).
- 'sdv' (int): Current setting for the Shutdown Voltage (millivolts).

## Examples

```
>>> automotive_settings = my_auto.get_all_automotive_settings()
>>> print(automotive_settings)
{
    "amd": 1,
    "lpe": 1,
    "sut": 10,
    "sot": 5,
    "hot": 15,
    "sdv": 1200
}
```

**get\_automotive\_mode()** → int

Get the automotive mode of the device.

Automotive-mode enables or disables system automotive features, this method uses the LPMCU protocol discussed in the README and documentation to get the automotive mode from the Sequence MCU.

**Parameters:** None

**Returns:** The automotive mode of the device. 0 indicates that the device is not in automotive mode, 1 indicates that the device is in automotive mode. A value < 0 indicates an error in the command or response.

**Return type:** int

## Example

```
>>> auto_mode = get_automotive_mode()
>>> print(f"Automotive Mode: {auto_mode}")
Automotive Mode: 1
```

**get\_hard\_off\_timer()** → int

Get the existing hard-off timer value from the MCU.

The number of seconds that the ignition input can be low before the MCU will force the system to power down. This method uses the LPMCU protocol discussed in the README and documentation to set the hard-off timer of the sequence MCU.

**Parameters:** None

**Returns:** The hard-off timer value of the device in seconds. The hard-off timer can be configured between 1 - 1048575 seconds. A value < 0 indicates an error in the command or response.

**Return type:** int

## Example

```
>>> hard_off_timer = my_auto.get_hard_off_timer()
>>> print(f"Hard Off Timer: {hard_off_timer}")
Hard Off Timer: 15
```

**get\_low\_power\_enable ()** → int

Get the low power enable status value from the MCU.

Low Power Enable enables entering a very low power state when the system powers off. The system can only wake from the power-button and the ignition switch when in this power state. This method uses the LPMCU protocol discussed in the README and documentation to get the low power enable status from the Sequence MCU.

**Parameters:** None

**Returns:** The low power enable status of the device. 0 indicates that low power mode is disabled, 1 indicates that low power mode is enabled. A value < 0 indicates an error in the command or response.

**Return type:** int

## Example

```
>>> low_power_enable = my_auto.get_low_power_enable()
>>> print(f"Low Power Enable: {low_power_enable}")
Low Power Enable: 1
```

**get\_low\_voltage\_timer ()** → int

Get the existing low voltage timer value from the MCU.

The low voltage timer controls the number of seconds that the measured voltage can be lower than the shutdown threshold before a forced shutdown will occur. This method uses the LPMCU protocol discussed in the README and documentation to set the low voltage timer of the sequence MCU.

**Parameters:** None

**Returns:** The low voltage timer value of the device in seconds. The low voltage timer can be configured between 1 - 1048575 seconds. A value < 0 indicates an error in the command or response.

**Return type:** int

## Example

```
>>> low_voltage_timer = my_auto.get_low_voltage_timer()
>>> print(f"Low Voltage Timer: {low_voltage_timer}")
Low Voltage Timer: 20
```

**get\_shutdown\_voltage ()** → int

Get the existing shutdown voltage value from the MCU.

The shutdown voltage value dictates threshold voltage for triggering low-voltage shutdown events. This method uses the LPMCU protocol discussed in the README and documentation to get the shutdown voltage threshold of the sequence MCU.

**Parameters:** None

**Returns:** The shutdown voltage value of the device in centi-volts. The shutdown voltage can be configured between 1.000 - 48.000 A value < 0 indicates an error in the command or response.

**Return type:** int

## Example

```
>>> shutdown_voltage = my_auto.get_shutdown_voltage()
>>> print(f"Shutdown Voltage: {shutdown_voltage}")
Shutdown Voltage: 600
```

**get\_soft\_off\_timer()** → int

Get the existing soft-off timer value from the MCU.

The soft-off timer controls the number of seconds that the ignition input can be low before the mcu requests the system powers down via a virtual power button event. This method uses the LPMCU protocol discussed in the README and documentation to set the soft-off timer of the sequence MCU.

**Parameters:** None

**Returns:** The soft-off timer value of the device in seconds. The soft-off timer can be configured between 1 - 1048575 seconds A value < 0 indicates an error in the command or response.

**Return type:** int

## Example

```
>>> soft_off_timer = my_auto.get_soft_up_timer()
>>> print(f"Soft Off Timer: {soft_off_timer}")
Soft Off Timer: 5
```

**get\_start\_up\_timer()** → int

Get the start-up timer value from the MCU.

The start-up timer controls the number of seconds that the ignition input must be stable before the system will power on. This method uses the LPMCU protocol discussed in the README and documentation to get the start up timer.

**Parameters:** None

**Returns:** the start-up timer value of the device in seconds. the start-up timer can be configured between 1 - 1048575 seconds A value < 0 indicates an error in the command or response.

**Return type:** int

## Example

```
>>> start_up_timer = my_auto.get_start_up_timer()
>>> print(f"Start Up Timer: {start_up_timer}")
Start Up Timer: 10
```

**set\_all\_automotive\_settings(setting\_inputs: list)** → list

Sets all automotive settings based on a provided input list of desired states.

(Note numpy commenting format was used as this was the only way to get the docstring to render correctly in Sphinx)

## Args

**setting\_input** : *list*

A list of values corresponding to the desired states for:

- `automotive_mode_enabled (int)`
- `low_power_enabled (int)`
- `start_up_timer_seconds (int)`
- `soft_off_timer_seconds (int)`
- `hard_off_timer_seconds (int)`
- `shutdown_voltage_mv (int)`

## Returns

**operation\_results** : *list*

A list of integer status codes from each `set_*` method, detailing the outcome of each operation. The items in this list correspond to:

- Result of `set_automotive_mode()` (int): Status code for setting automotive mode.
- Result of `set_low_power_enable()` (int): Status code for enabling/disabling low power mode.
- Result of `set_start_up_timer()` (int): Status code for setting the start-up timer.
- Result of `set_soft_off_timer()` (int): Status code for setting the soft-off timer.
- Result of `set_hard_off_timer()` (int): Status code for setting the hard-off timer.
- Result of `set_shutdown_voltage()` (int): Status code for setting the shutdown voltage.

## Example

```
>>> setting_inputs = [1, 1, 10, 5, 15, 12000] # 12000mV for 12.0V
>>> results = my_auto.set_all_automotive_settings(setting_inputs)
>>> print(results)
[0, 0, 0, 0, 0, 0]
```

**set\_automotive\_mode** (amd: int) → int

Set the automotive mode of the device.

Automotive-mode enables or disables system automotive features. This method uses the LPMCU protocol discussed in the README and documentation to set whether the device is in automotive mode or not.

**Parameters:** `amd (int)` – The automotive mode to set. 0 turn automotive mode off, 1 turn automotive mode on.

**Returns:** The status of the command. 0 indicates success, < 0 indicates an error in the command or response.

**Return type:** int

**Raises:**

- **ValueError** – If the input parameter is not a valid integer or is out of range.
- **TypeError** – If the input parameter is not of type int.

## Example

```
>>> auto_mode = my_auto.set_automotive_mode(1)
>>> print(f"Automotive Mode Set to: {auto_mode}")
Automotive Mode Set to: 0
```

**set\_hard\_off\_timer**(hot: int) → int

Set the hard-off timer value in the sequence MCU.

The number of seconds that the ignition input can be low before the MCU will force the system to power down. This method uses the LPMCU protocol discussed in the README and documentation to set the hard off timer of the sequence MCU.

**Parameters:** **hot** (*int*) – The hard off timer to set. 1 - 1048575 seconds.

**Returns:** The status of the command. 0 indicates success, < 0 indicates an error in the command or response.

**Return type:** int

**Raises:**

- **ValueError** – If the input parameter is not a valid integer or is out of range.
- **TypeError** – If the input parameter is not of type int.

## Example

```
>>> status = my_auto.set_hard_off_timer(15)
>>> print(f"Success" if status == 0 else f"Error: {status}")
Success
```

**set\_low\_power\_enable**(lpe: int) → int

Set the low power enable status in the sequence MCU.

Low Power Enable enables entering a very low power state when the system powers off. The system can only wake from the power-button and the ignition switch when in this power state. This method uses the LPMCU protocol discussed in the README and documentation to set the low power enable status of the sequence MCU.

**Parameters:** **lpe** (*int*) – The low power enable status to set. 0 turn low power mode off, 1 turn low power mode on.

**Returns:** The status of the command. 0 indicates success, < 0 indicates an error in the command or response.

**Return type:** int

## Example

```
>>> status = my_auto.set_low_power_enable(1)
>>> print(f"Success" if status == 0 else f"Error: {status}")
Success
```

**set\_low\_voltage\_timer**(lvt: int) → int

Set the low voltage timer value in the sequence MCU.

The number of seconds that the measured voltage can be lower than the shutdown threshold before a forced shutdown will occur. This method uses the LPMCU protocol discussed in the README and documentation to set the low voltage timer of the sequence MCU.

**Parameters:** **lvt** (*int*) – The low voltage timer to set. 1 - 1048575 seconds.



**Returns:** The status of the command. 0 indicates success, < 0 indicates an error in the command or response.

**Return type:** int

**Raises:**

- **ValueError** – If the input parameter is not a valid integer or is out of range.
- **TypeError** – If the input parameter is not of type int.

## Example

```
>>> status = my_auto.set_low_voltage_timer(20)
>>> print(f"Success" if status == 0 else f"Error: {status}")
Success
```

**set\_shutdown\_voltage** (sdv: int) → int

Set the shutdown voltage value in the sequence MCU.

The shutdown voltage value dictates threshold voltage for triggering low-voltage shutdown events. This method uses the LPMCU protocol discussed in the README and documentation to set the shutdown voltage threshold of the sequence MCU.

**Parameters:** **sdv** (int) – The shutdown voltage to set. 1.000 - 48.000 volts.

**Returns:** The status of the command. 0 indicates success, < 0 indicates an error in the command or response.

**Return type:** int

**Raises:**

- **ValueError** – If the input parameter is not a valid integer or is out of range.
- **TypeError** – If the input parameter is not of type int.

## Example

```
>>> status = my_auto.set_shutdown_voltage(12)
>>> print(f"Success" if status == 0 else f"Error: {status}")
Success
```

**set\_soft\_off\_timer** (sot: int) → int

Set the soft-off timer value in the sequence MCU.

The soft-off timer controls the number of seconds that the ignition input can be low before the mcu requests the system powers down via a virtual power button event. This method uses the LPMCU protocol discussed in the README and documentation to set the soft off timer of the sequence MCU.

**Parameters:** **sot** (int) – The soft off timer to set. 1 - 1048575 seconds.

**Returns:** The status of the command. 0 indicates success, < 0 indicates an error in the command or response.

**Return type:** int

**Raises:**

- **ValueError** – If the input parameter is not a valid integer or is out of range.
- **TypeError** – If the input parameter is not of type int.

## Example

```
>>> status = my_auto.set_soft_off_timer(5)
>>> print(f"Success" if status == 0 else f"Error: {status}")
Success
```

**set\_start\_up\_timer** (sut: int) → int

Set the start-up timer value in the sequence MCU.

The start-up timer controls the number of seconds that the ignition input must be stable before the system will power on. This method uses the LPMCU protocol discussed in the README and documentation to set the start up timer of the sequence MCU.

**Parameters:** **sut** (int) – The start up timer to set. 1 - 1048575 seconds.

**Returns:** The status of the command. 0 indicates success, < 0 indicates an error in the command or response.

**Return type:** int

**Raises:**

- **ValueError** – If the input parameter is not a valid integer or is out of range.
- **TypeError** – If the input parameter is not of type int.

## Example

```
>>> status = my_auto.set_start_up_timer(10)
>>> print(f"Success" if status == 0 else f"Error: {status}")
Success
```

**show\_info** () → None

Show information about the target MCU depending on the child class.

This method is used to display information about the target MCU. It is an inheritable method provided to the base class. It will print a manual to the console, which is a text file with the documentation of the target MCU. It closely interacts with the `_read_files` method, which is also an inheritable method.

**Parameters:** **None**

**Returns:** None

**Raises:**

- **FileNotFoundError** – If the file is not found.
- **IOError** – If there is an error reading the file.
- **Exception** – For any other exceptions that may occur.

**class** `src.DioHandler` (serial\_connection\_label: str = None)

Bases: **OnLogicM031Manager**

Handles Digital Input/Output (DIO) operations for the DIO card.

This class provides methods to get and set the states of digital input and output pins. It can get and set input and output pins individually, or in bulk. It uses the LPMCU methods and protocols inherited from `OnLogicM031Manager`. A new feature in this line of DIO cards is the ability to set the contact type of the pins to Wet or Dry.

As a reminder, the DIO card has 8 digital input pins and 8 digital output pins.

The digital input pins are active-low, meaning that a 0 indicates that the pin is on, and a 1 indicates that the pin is off. The digital output pins are active-high, meaning that a 0 indicates that the pin is off, and a 1 indicates that the pin is on.

The DIO card also has a contact type for each pin, which can be either Wet or Dry. The contact type is set using the `set_di_contact` and `set_do_contact` methods. The contact type can be gotten by using the `get_di_contact` and `get_do_contact` methods.

The claim method is used to connect to the DIO card and the parameter can be left blank for an auto-lock onto the port. The release method disconnects from the DIO card.

## Note

More information on the DIO card can be found in the Data sheet, and also by calling the `show_info` method.

Pin Diagram:



**Raises:**

- **ValueError** – if di\_pin is not in the range of 0-7
- **TypeError** – if di\_pin is not an integer

**Example**

```
>>> with DioHandler() as dio_handler:
...     dio_handler.get_di(0)
1
```

**get\_di\_contact ()** → int

Gets the contact state of the digital input pins on the DIO card.

0 indicates that DI is in Wet Contact mode and 1 Indicates that DI is in Dry Contact Mode.

**Parameters:** None

**Returns:** 0, indicating Wet Contact, 1, indicating Dry Contact, StatusTypes.SEND\_CMD\_FAILURE if command send failed, or any other negative value indicating failure.

**Return type:** int

**Raises:**

- **TypeError** – if contact\_type is not an integer
- **ValueError** – if contact\_type is not in the range of 0-1

**Note**

Consult DIO description section of the data sheet for more details on the specification of contact types.

**Example**

```
>>> with DioHandler() as dio_handler:
...     dio_handler.get_di_contact()
0
```

**get\_do (do\_pin: int)** → int

Gets the state of active-high digital outputs on the DIO card.

**Parameters:** do\_pin (*int*) – digital output pin with range [0-7]

**Returns:** 0, indicating off, 1, indicating on, StatusTypes.SEND\_CMD\_FAILURE if command send failed, or any other negative value indicating failure.

**Return type:** int

**Raises:**

- **ValueError** – if do\_pin is not in the range of 0-7
- **TypeError** – if do\_pin is not an integer

**Example**

```
>>> with DioHandler() as dio_handler:
...     dio_handler.get_do(0)
1
```

**get\_do\_contact ()** → int

Gets the contact state of the digital output pins on the DIO card.

0 indicates that DO is in Wet Contact mode and 1 Indicates that DO is in Dry Contact Mode.

**Parameters:** None

**Returns:** 0, indicating Wet Contact, 1, indicating Dry Contact, StatusTypes.SEND\_CMD\_FAILURE if command send failed, or any other negative value indicating failure.

**Return type:** int

**Raises:**

- **TypeError** – if contact\_type is not an integer
- **ValueError** – if contact\_type is not in the range of 0-1

## Note

Consult DIO description section of the data sheet for more details on the specification of contact types.

## Example

```
>>> with DioHandler() as dio_handler:
...     dio_handler.get_do_contact()
0
```

**set\_all\_output\_states** (do\_lst: list) → list

Sets the states of all digital output pins given an input list of binary inputs.

This is a wrapper method that calls set\_do for each pin in the range of 0-7. Sets the states of all digital output pins given an input list of binary inputs. Indices 0-7 correspond to output pins 0-7 respectively.

**Parameters:** **do\_lst** (*list[int]*) – A list of 8 values (0 or 1), one for each output pin.

**Returns:** A list of status codes for each pin operation, there should be 7 in total. 0 indicates success; < 0 indicates failure.

**Return type:** list[int]

**Raises:**

- **TypeError** – If do\_lst is not a list or is None.
- **ValueError** – If do\_lst does not contain exactly 8 values or contains invalid values, i.e. if any value in do\_lst is not a binary valued integer in the range of 0-1.

**set\_di\_contact** (contact\_type: int) → int

Sets the contact state of the digital input pins on the DIO card.

0 indicates that DI is in Wet Contact mode and 1 Indicates that DI is in Dry Contact Mode.

**Parameters:** **contact\_type** (*int*) – 0 for Wet Contact, 1 for Dry Contact

**Returns:** 0, indicating success, StatusTypes.SEND\_CMD\_FAILURE if command send failed, or any other negative value indicating failure.

**Return type:** int

**Raises:**

- **TypeError** – if contact\_type is not an integer
- **ValueError** – if contact\_type is not in the range of 0-1

## Note

Consult DIO description section of the data sheet for more details on the specification of contact types.

**set\_do** (pin: int, value: int) → int

Sets the state of active-high digital outputs on the DIO card.

**Parameters:**

- **pin** (*int*) – digital output pin with range [0-7]
- **value** (*int*) – binary value to set the pin to, either 0 or 1

**Returns:** 0, indicating success, `StatusTypes.SEND_CMD_FAILURE` if command send failed, or any other negative value indicating failure.

**Return type:** `int`

**Raises:**

- **ValueError** – if pin is not in the range of 0-7
- **TypeError** – if pin or value is not an integer

**Example**

```
>>> with DioHandler() as dio_handler:
...     dio_handler.set_do(0, 1)
0
```

**set\_do\_contact** (`contact_type: int`) → `int`

Sets the contact state of the digital output pins on the DIO card.

0 indicates that DO is in Wet Contact mode and 1 Indicates that DO is in Dry Contact Mode.

**Parameters:** **contact\_type** (*int*) – 0 for Wet Contact, 1 for Dry Contact

**Returns:** 0, indicating success, `StatusTypes.SEND_CMD_FAILURE` if command send failed, or any other negative value indicating failure.

**Return type:** `int`

**Raises:**

- **TypeError** – if `contact_type` is not an integer
- **ValueError** – if `contact_type` is not in the range of 0-1

**Note**

Consult DIO description section of the data sheet for more details on the specification of contact types.

**show\_info** () → `None`

Displays DIO card info located in the docs folder, required `OnLogicM031Manager`.

**class** `src.LoggingUtil` (`logger_name`, `logger_level`, `handler_mode`)

Bases: `object`

LoggingUtil class for configuring logging in the OnLogic M031 Manager.

This class provides methods to set up a logger with a specified name, level, and handlers.

**logger\_name**

The name of the logger. It is converted to lowercase and stripped of whitespace. Note, please use 'root' for the root logger unless you are willing to add a field to the target class to allow for a custom logger utility.

**Type:** `str`

**logger\_level**

The logging level (e.g., 'info', 'debug', 'error'). It is converted to lowercase and stripped of whitespace.

**Type:** `str`

**handler\_mode**

The mode for the handler (e.g., 'console', 'file', 'both'). It is converted to lowercase and stripped of whitespace.

**Type:** `str`

**logger**

The configured logger instance.

**Type:** logging.Logger

**format**

The format for log messages, specified in the initialization of the class.

**Type:** str

## Example

For debug logging to console:

```
config_logger = LoggingUtil(
    logger_name='root', logger_level="DEBUG", handler_mode="CONSOLE"
)

config_logger.config_logger_elements()
```

For info logging to file:

```
config_logger = LoggingUtil(
    logger_name='root', logger_level="INFO", handler_mode="FILE"
)

config_logger.config_logger_elements()
```

For error logging to both console and file:

```
config_logger = LoggingUtil(
    logger_name='root', logger_level="ERROR", handler_mode="BOTH"
)

config_logger.config_logger_elements()
```

### **\_\_init\_\_** (logger\_name, logger\_level, handler\_mode)

Initialize the LoggingUtil class with the specified logger name, level, and handler mode. :param logger\_name: The name of the logger. It is converted to lowercase and stripped of whitespace.

Note, please use 'root' for the root logger unless you are willing to add a field to the target class to allow for a custom logger utility.

**Parameters:**

- **logger\_level** (*str*) – The logging level (e.g., 'info', 'debug', 'error'). It is converted to lowercase and stripped of whitespace.
- **handler\_mode** (*str*) – The mode for the handler (e.g., 'console', 'file', 'both'). It is converted to lowercase and stripped of whitespace.

### **config\_logger\_elements** () → Logger | None

Configure the logger with the specified name, level, and handlers.

This method will set up a logger based on logging configurations provided to the LoggingUtil class constructor. It supports 3 logging level configurations, DEBUG, INFO, and ERROR. And the log messages can be directed at the Console, an external file, or both.

It chooses the file handler object based off cleaned input mode selection. The format of the logger is set to match the default format defined in the class. After, it will set the logger as a null handler if there is handler objects configured. It will make sure not to propagate the logging configurations or messages to other loggers.

**Params:**

None

**Returns:** The configured logger instance.

**Return type:** logging.Logger

## Example

```
config_logger = LoggingUtil(
    logger_name='root', logger_level="DEBUG", handler_mode="CONSOLE"
) config_logger.config_logger_elements()
```

```
class src.OnLogicM031Manager (serial_connection_label: str = None)
```

Bases: **ABC**

Administers the serial connection and communication protocol with the embedded MCUs.

This class provides tools to communicate with the microcontrollers embedded in the K/HX-52x DIO-Add and Sequence MCU for Automotive Control. It contains the root context manager, serial handling methods, input validation, command construction, and frame reception and validation functionality. These are all inheritable by the child classes:

- **AutomotiveHandler**
- **DioHandler**

## Note

This class should not be directly called. Instead, it should be accessed through child classes like **AutomotiveHandler** or **DioHandler**.

When using a child class as a context manager, the `__enter__` and `__exit__` methods are called to claim and release the serial port. This means that the *serial\_connection\_label* should be specified as a parameter during the instantiation of the child class.

The class and its children also contain extensive logging for debugging and tracking purposes. Logging is designed to trace the execution of the code and log important events, not the target payloads themselves (which are returned by called functions in child classes).

<https://fastcrc.readthedocs.io/en/latest/> contains more info on the CRC8 methodology used.

## Example

Instantiating and using a child class (e.g., **AutomotiveHandler**) with context manager:

```
>>> with AutomotiveHandler(serial_connection_label="/dev/ttyS4") as my_auto:
...     # Perform operations with my_auto
...     pass
```

Or using **DioHandler**:

```
>>> with DioHandler(serial_connection_label="/dev/ttyS5") as my_dio:
...     # Perform operations with my_dio
...     pass
```

**Parameters:** **serial\_connection\_label** (*str*) – The label or device path for the serial connection (e.g., `"/dev/ttyS4"`). This is passed when a child class is instantiated.

**serial\_connection\_label**

Stores the label for the serial connection.

**Type:** `str`

**port**

The serial port object for communication, which is an instance of `serial.Serial` from the *pyserial* library once the port is opened, or `None` if not yet opened/set up.

**Type:** `serial.Serial | None`



**is\_setup**

Indicates if the serial connection has been successfully set up (True) or not (False).

**Type:** bool

**\_\_del\_\_()**

Destroy the object and end device communication gracefully.

**\_\_init\_\_(serial\_connection\_label: str = None)**

Initialize the OnLogicM031Manager class.

**Parameters:** **serial\_connection\_label** (*str*) – The label or device path for the serial connection (e.g., “/dev/ttySx” or ). This is passed when a child class is instantiated.

**\_\_str\_\_()**

String representation of class.

**claim(serial\_connection\_label=None) → None**

Claim the serial port and set up the connection.

This method initializes the serial port with the given baudrate and device descriptor. If the port is not specified, it will search for the device with the given VID and PID when used for DIO Utility. Otherwise, it will simply initialize the port with the given serial connection label.

**Parameters:** **serial\_connection\_label** (*str*) – The label or device path for the serial connection (e.g., “/dev/ttySx” or “COMx”). This is passed when a child class is instantiated.

**Returns:** None

**Raises:**

- **serial.SerialException** – If the port cannot be opened or configured.
- **ValueError** – If the port is not found or cannot be opened.

**get\_version() → str | int**

Get the firmware version of the microcontroller.

Retrieves the firmware of the microcontroller by using the GET\_FIRMWARE\_VERSION command and the LPMCU protocol discussed in the documentation and README. The version is returned as a string in the format “X.X.X”,

**Params:**

None

**Returns:** The firmware version of the microcontroller in the format “X.X.X”. If the version cannot be retrieved, it returns StatusTypes.SEND\_CMD\_FAILURE. If the payload is empty, it returns StatusTypes.FORMAT\_NONE\_ERROR.

**Return type:** str

**list\_all\_available\_ports(verbose: bool = False) → None**

List all available serial ports.

A convenient method provided to list all available serial ports on the system. The user should be able to the DIO card by its Identifier if plugged in, but unfortunately, the device descriptor is not available for the Sequence MCU for Automotive control. It is inheritable from the base class and can be called from both the Automotive and DIO classes.

**Parameters:** **verbose** (*bool*) – If True, prints detailed information about each port. If False, prints only the port names.

**Returns:** This method does not return anything. It prints the available ports to the console.

**Return type:** None

**release()**

Release the serial port, reset the buffers, and reset the connection.

**abstractmethod show\_info() → None**

Show information about the target MCU depending on the child class.

This method is used to display information about the target MCU. It is an inheritable method provided to the base class. It will print a manual to the console, which is a text file with the documentation of the target MCU. It closely interacts with the `_read_files` method, which is also an inheritable method.

**Parameters:** `None`

**Returns:** `None`

**Raises:**

- **FileNotFoundError** – If the file is not found.
- **IOError** – If there is an error reading the file.
- **Exception** – For any other exceptions that may occur.

## src.onlogic\_m031\_manager Module

Administers the serial connection and communication protocol with the embedded MCUs.

This module contains the `OnLogicM031Manager`, used to control and communicate with the Nuvoton microcontrollers embedded in OnLogic HX/K5xx products.

**class** `src.onlogic_m031_manager.OnLogicM031Manager` (`serial_connection_label: str = None`)

Bases: `ABC`

Administers the serial connection and communication protocol with the embedded MCUs.

This class provides tools to communicate with the microcontrollers embedded in the K/HX-52x DIO-Add and Sequence MCU for Automotive Control. It contains the root context manager, serial handling methods, input validation, command construction, and frame reception and validation functionality. These are all inheritable by the child classes:

- `AutomotiveHandler`
- `DioHandler`

### Note

This class should not be directly called. Instead, it should be accessed through child classes like `AutomotiveHandler` or `DioHandler`.

When using a child class as a context manager, the `__enter__` and `__exit__` methods are called to claim and release the serial port. This means that the `serial_connection_label` should be specified as a parameter during the instantiation of the child class.

The class and its children also contain extensive logging for debugging and tracking purposes. Logging is designed to trace the execution of the code and log important events, not the target payloads themselves (which are returned by called functions in child classes).

<https://fastcrc.readthedocs.io/en/latest/> contains more info on the CRC8 methodology used.

### Example

Instantiating and using a child class (e.g., `AutomotiveHandler`) with context manager:

```
>>> with AutomotiveHandler(serial_connection_label="/dev/ttyS4") as my_auto:
...     # Perform operations with my_auto
...     pass
```

Or using `DioHandler`:

```
>>> with DioHandler(serial_connection_label="/dev/ttyS5") as my_dio:
...     # Perform operations with my_dio
...     pass
```

**Parameters:** `serial_connection_label` (*str*) – The label or device path for the serial connection (e.g., `"/dev/ttyS4"`). This is passed when a child class is instantiated.

**serial\_connection\_label**

Stores the label for the serial connection.

**Type:** str

**port**

The serial port object for communication, which is an instance of `serial.Serial` from the *pyserial* library once the port is opened, or `None` if not yet opened/set up.

**Type:** `serial.Serial` | `None`

**is\_setup**

Indicates if the serial connection has been successfully set up (`True`) or not (`False`).

**Type:** bool

**\_\_del\_\_()**

Destroy the object and end device communication gracefully.

**\_\_init\_\_(serial\_connection\_label: str = None)**

Initialize the OnLogicM031Manager class.

**Parameters:** **serial\_connection\_label** (*str*) – The label or device path for the serial connection (e.g., “/dev/ttySx” or ). This is passed when a child class is instantiated.

**\_\_str\_\_()**

String representation of class.

**claim(serial\_connection\_label=None) → None**

Claim the serial port and set up the connection.

This method initializes the serial port with the given baudrate and device descriptor. If the port is not specified, it will search for the device with the given VID and PID when used for DIO Utility. Otherwise, it will simply initialize the port with the given serial connection label.

**Parameters:** **serial\_connection\_label** (*str*) – The label or device path for the serial connection (e.g., “/dev/ttySx” or “COMx”). This is passed when a child class is instantiated.

**Returns:** None

**Raises:**

- **serial.SerialException** – If the port cannot be opened or configured.
- **ValueError** – If the port is not found or cannot be opened.

**get\_version() → str | int**

Get the firmware version of the microcontroller.

Retrieves the firmware of the microcontroller by using the GET\_FIRMWARE\_VERSION command and the LPMCU protocol discussed in the documentation and README. The version is returned as a string in the format “X.X.X”,

**Params:**

None

**Returns:** The firmware version of the microcontroller in the format “X.X.X”. If the version cannot be retrieved, it returns `StatusTypes.SEND_CMD_FAILURE`. If the payload is empty, it returns `StatusTypes.FORMAT_NONE_ERROR`.

**Return type:** str

**list\_all\_available\_ports(verbose: bool = False) → None**

List all available serial ports.

A convenient method provided to list all available serial ports on the system. The user should be able to the DIO card by it's Identifier if plugged in, but unfortunately, the device descriptor is not available for the Sequence MCU for Automotive control. It is inheritable from the base class and can be called from both the Automotive and DIO classes.

**Parameters:** **verbose** (*bool*) – If True, prints detailed information about each port. If False, prints only the port names.

**Returns:** This method does not return anything. It prints the available ports to the console.

**Return type:** None

```
release ()
```

Release the serial port, reset the buffers, and reset the connection.

```
abstractmethod show_info() → None
```

Show information about the target MCU depending on the child class.

This method is used to display information about the target MCU. It is an inheritable method provided to the base class. It will print a manual to the console, which is a text file with the documentation of the target MCU. It closely interacts with the `_read_files` method, which is also an inheritable method.

**Parameters:**    **None**

**Returns:** None

**Raises:**

- **FileNotFoundException** – If the file is not found.
- **IOException** – If there is an error reading the file.
- **Exception** – For any other exceptions that may occur.

## src.dio\_handler Module

DioHandler module for managing Digital Input/Output (DIO) operations.

Provides methods that allow users to access nearly all features of DIO card.

```
class src.dio_handler.DioHandler(serial_connection_label: str = None)
```

Bases: OnLogicM031Manager

Handles Digital Input/Output (DIO) operations for the DIO card.

This class provides methods to get and set the states of digital input and output pins. It can get and set input and output pins individually, or in bulk. It uses the LPMCU methods and protocols inherited from OnLogicM031Manager. A new feature in this line of DIO cards is the ability to set the contact type of the pins to Wet or Dry.

As a reminder, the DIO card has 8 digital input pins and 8 digital output pins.

The digital input pins are active-low, meaning that a 0 indicates that the pin is on, and a 1 indicates that the pin is off. The digital output pins are active-high, meaning that a 0 indicates that the pin is off, and a 1 indicates that the pin is on.

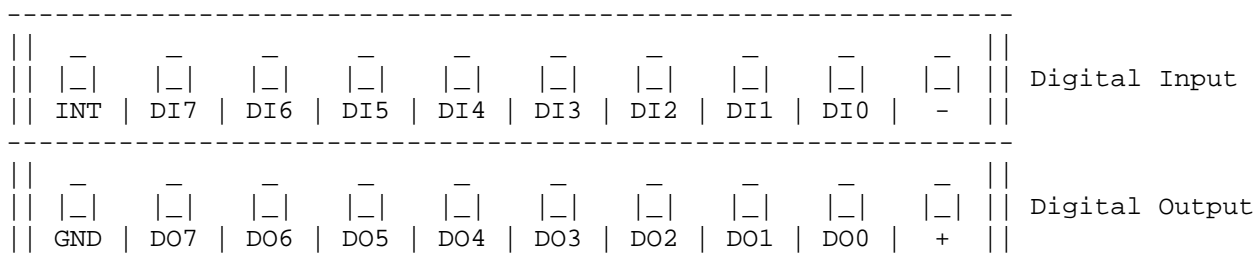
The DIO card also has a contact type for each pin, which can be either Wet or Dry. The contact type is set using the `set_di_contact` and `set_do_contact` methods. The contact type can be gotten by using the `get_di_contact` and `get_do_contact` methods.

The claim method is used to connect to the DIO card and the parameter can be left blank for an auto-lock onto the port. The release method disconnects from the DIO card.

## Note

More information on the DIO card can be found in the Data sheet, and also by calling the `show_info` method.

### Pin Diagram:



**serial\_connection\_label**

The label of the serial connection. If None, the class will attempt to find the correct port automatically.

**Type:** str

## Examples

Claim and release port for the DioHandler class with either:  
`my_dio = DioHandler() my_dio.claim() ... my_dio.release()`  
 or

**with DioHandler() as dio\_handler:**

...

**\_\_init\_\_(serial\_connection\_label: str = None)**

Initializes the DioHandler class.

**Parameters:** **serial\_connection\_label** (*str*) – The label of the serial connection. If None, the class will attempt to find the correct port automatically.

**get\_all\_input\_states ()** → list

`all_input_states = []`

**for i in range(0, 8):**

`all_input_states.append()`

`return all_input_states`

**get\_all\_io\_states ()** → list

**get\_all\_output\_states ()** → list

Gets the states of all digital output pins.

This is a wrapper method that calls `get_do` for each pin in the range of 0-7. The method returns a list of 8 binary values, one for each output pin. Indices 0-7 correspond to output pins 0-7 respectively.

**Parameters:** **None**

**Returns:** A list of 8 binary-valued pin states, if successful, all values should be between 0 and 1. values < 0 indicate failure.

**Return type:** list[int]

**get\_di (di\_pin: int)** → int

Gets the state of active-low digital inputs on the DIO card.

**Parameters:** **di\_pin** (*int*) – digital input pin with range [0-7]

**Returns:** 0, indicating on, 1, indicating off, StatusTypes.SEND\_CMD\_FAILURE if command send failed, or any other negative value indicating failure.

**Return type:** int

**Raises:**

- **ValueError** – if di\_pin is not in the range of 0-7
- **TypeError** – if di\_pin is not an integer

## Example

```
>>> with DioHandler() as dio_handler:
...     dio_handler.get_di(0)
1
```

**get\_di\_contact ()** → int

Gets the contact state of the digital input pins on the DIO card.

0 indicates that DI is in Wet Contact mode and 1 Indicates that DI is in Dry Contact Mode.

**Parameters:** None

**Returns:** 0, indicating Wet Contact, 1, indicating Dry Contact, StatusTypes.SEND\_CMD\_FAILURE if command send failed, or any other negative value indicating failure.

**Return type:** int

**Raises:**

- **TypeError** – if contact\_type is not an integer
- **ValueError** – if contact\_type is not in the range of 0-1

## Note

Consult DIO description section of the data sheet for more details on the specification of contact types.

## Example

```
>>> with DioHandler() as dio_handler:
...     dio_handler.get_di_contact()
0
```

**get\_do (do\_pin: int)** → int

Gets the state of active-high digital outputs on the DIO card.

**Parameters:** **do\_pin** (*int*) – digital output pin with range [0-7]

**Returns:** 0, indicating off, 1, indicating on, StatusTypes.SEND\_CMD\_FAILURE if command send failed, or any other negative value indicating failure.

**Return type:** int

**Raises:**

- **ValueError** – if do\_pin is not in the range of 0-7
- **TypeError** – if do\_pin is not an integer

## Example

```
>>> with DioHandler() as dio_handler:
...     dio_handler.get_do(0)
1
```

**get\_do\_contact ()** → int

Gets the contact state of the digital output pins on the DIO card.

0 indicates that DO is in Wet Contact mode and 1 Indicates that DO is in Dry Contact Mode.

**Parameters:** None

**Returns:** 0, indicating Wet Contact, 1, indicating Dry Contact, StatusTypes.SEND\_CMD\_FAILURE if command send failed, or any other negative value indicating failure.

**Return type:** int

**Raises:**

- **TypeError** – if contact\_type is not an integer
- **ValueError** – if contact\_type is not in the range of 0-1

## Note

Consult DIO description section of the data sheet for more details on the specification of contact types.

## Example

```
>>> with DioHandler() as dio_handler:
...     dio_handler.get_do_contact()
0
```

**set\_all\_output\_states** (do\_lst: list) → list

Sets the states of all digital output pins given an input list of binary inputs.

This is a wrapper method that calls set\_do for each pin in the range of 0-7. Sets the states of all digital output pins given an input list of binary inputs. Indices 0-7 correspond to output pins 0-7 respectively.

**Parameters:** **do\_lst** (*list[int]*) – A list of 8 values (0 or 1), one for each output pin.

**Returns:** A list of status codes for each pin operation, there should be 7 in total. 0 indicates success; < 0 indicates failure.

**Return type:** list[int]

**Raises:**

- **TypeError** – If do\_lst is not a list or is None.
- **ValueError** – If do\_lst does not contain exactly 8 values or contains invalid values, i.e. if any value in do\_lst is not a binary valued integer in the range of 0-1.

**set\_di\_contact** (contact\_type: int) → int

Sets the contact state of the digital input pins on the DIO card.

0 indicates that DI is in Wet Contact mode and 1 Indicates that DI is in Dry Contact Mode.

**Parameters:** **contact\_type** (*int*) – 0 for Wet Contact, 1 for Dry Contact

**Returns:** 0, indicating success, StatusTypes.SEND\_CMD\_FAILURE if command send failed, or any other negative value indicating failure.

**Return type:** int

**Raises:**

- **TypeError** – if contact\_type is not an integer
- **ValueError** – if contact\_type is not in the range of 0-1

## Note

Consult DIO description section of the data sheet for more details on the specification of contact types.

**set\_do** (pin: int, value: int) → int

Sets the state of active-high digital outputs on the DIO card.

**Parameters:**

- **pin** (*int*) – digital output pin with range [0-7]
- **value** (*int*) – binary value to set the pin to, either 0 or 1

**Returns:** 0, indicating success, StatusTypes.SEND\_CMD\_FAILURE if command send failed, or any other negative value indicating failure.

**Return type:** int

**Raises:**

- **ValueError** – if pin is not in the range of 0-7
- **TypeError** – if pin or value is not an integer

## Example

```
>>> with DioHandler() as dio_handler:
...     dio_handler.set_do(0, 1)
0
```

```
set_do_contact (contact_type: int) → int
```

Sets the contact state of the digital output pins on the DIO card.

0 indicates that DO is in Wet Contact mode and 1 Indicates that DO is in Dry Contact Mode.

**Parameters:** **contact\_type** (*int*) – 0 for Wet Contact, 1 for Dry Contact

**Returns:** 0, indicating success, `StatusTypes.SEND_CMD_FAILURE` if command send failed, or any other negative value indicating failure.

**Return type:** int

**Raises:**

- **TypeError** – if `contact_type` is not an integer
- **ValueError** – if `contact_type` is not in the range of 0-1

## Note

Consult DIO description section of the data sheet for more details on the specification of contact types.

**show info ()** → None

Displays DIO card info located in the docs folder, required OnLogicM031Manager.

## src.automotive\_handler Module

Provides the AutomotiveHandler class for managing automotive features of the OnLogic Nuvoton MCU.

This module includes methods to interact with the automotive features of the OnLogic Nuvoton MCU. For more information about the automotive mode settings, please refer to the Automotive section of the documentation or the docstrings below.

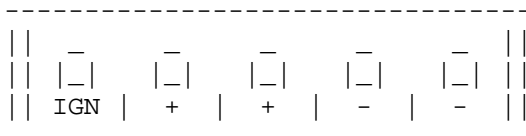
```
class src.automotive handler.AutomotiveHandler(serial connection label=None)
```

Bases: OnLogicM031Manager

AutomotiveHandler class for managing automotive features of the OnLogic Nuvoton MCU.

This class provides methods to interact with the automotive features of the OnLogic Nuvoton MCU. For more information about the automotive features, please refer to the AutomotiveModeDescription section of the documentation or the docstrings below.

### K52x Ignition Pin Diagram:



```
serial connection label
```

The label of the serial connection.

Type: str

## Examples

Claim and release port for the Automotive class with either:



```

my_auto = AutomotiveHandler()

# will be "/dev/ttySX" on Linux or "COMX" on Windows
my_auto.claim("COMX")
# ... (other operations)
my_auto.release()

or

with AutomotiveHandler() as my_auto:
    # ... (operations within the context manager)
    # Port is claimed on entry and released on exit
    pass

```

**\_\_init\_\_(serial\_connection\_label=None)**

Initialize the AutomotiveHandler class.

**Parameters:** **serial\_connection\_label** (*str*) – The label of the serial connection.

**get\_all\_automotive\_settings** (output\_to\_console: bool = False) → dict

Get all automotive settings from the sequence MCU.

This method is a wrapper that calls all get automotive attributes and formats them in one dictionary. It provides the option to print the settings to the console for easy viewing, similar to the screen provided in the BIOS settings. This method uses the LPMCU protocol discussed in the README and documentation to get the automotive settings from the sequence MCU.

(Note numpy commenting format was used as this was the only way to get the docstring to render correctly in Sphinx)

## Parameters

**output\_to\_console** : *bool*

If True, print the settings to the console.

## Returns

**automotive\_settings** : *dict*

A dictionary containing the current automotive settings of the device. The keys and their corresponding value types and meanings are:

- 'amd' (int): Current state of Automotive Mode (0 for enabled, 1 for disabled).
- 'lpe' (int): Current state of Low Power Enable (0 for enabled, 1 for disabled).
- 'sut' (int): Current setting for the Start Up Timer (Seconds).
- 'sot' (int): Current setting for the Soft Off Timer (Seconds).
- 'hot' (int): Current setting for the Hard Off Timer (Seconds).
- 'sdv' (int): Current setting for the Shutdown Voltage (millivolts).

## Examples

```
>>> automotive_settings = my_auto.get_all_automotive_settings()
>>> print(automotive_settings)
{
    "amd": 1,
    "lpe": 1,
    "sut": 10,
    "sot": 5,
    "hot": 15,
    "sdv": 1200
}
```

**get\_automotive\_mode ()** → int

Get the automotive mode of the device.

Automotive-mode enables or disables system automotive features, this method uses the LPMCU protocol discussed in the README and documentation to get the automotive mode from the Sequence MCU.

**Parameters:** None

**Returns:** The automotive mode of the device. 0 indicates that the device is not in automotive mode, 1 indicates that the device is in automotive mode. A value < 0 indicates an error in the command or response.

**Return type:** int

### Example

```
>>> auto_mode = get_automotive_mode()
>>> print(f"Automotive Mode: {auto_mode}")
Automotive Mode: 1
```

**get\_hard\_off\_timer ()** → int

Get the existing hard-off timer value from the MCU.

The number of seconds that the ignition input can be low before the MCU will force the system to power down. This method uses the LPMCU protocol discussed in the README and documentation to set the hard-off timer of the sequence MCU.

**Parameters:** None

**Returns:** The hard-off timer value of the device in seconds. The hard-off timer can be configured between 1 - 1048575 seconds. A value < 0 indicates an error in the command or response.

**Return type:** int

### Example

```
>>> hard_off_timer = my_auto.get_hard_off_timer()
>>> print(f"Hard Off Timer: {hard_off_timer}")
Hard Off Timer: 15
```

**get\_low\_power\_enable ()** → int

Get the low power enable status value from the MCU.

Low Power Enable enables entering a very low power state when the system powers off. The system can only wake from the power-button and the ignition switch when in this power state. This method uses the LPMCU protocol discussed in the README and documentation to get the low power enable status from the Sequence MCU.

**Parameters:** None

**Returns:** The low power enable status of the device. 0 indicates that low power mode is disabled, 1 indicates that low power mode is enabled. A value < 0 indicates an error in the command or response.

**Return type:** int

## Example

```
>>> low_power_enable = my_auto.get_low_power_enable()
>>> print(f"Low Power Enable: {low_power_enable}")
Low Power Enable: 1
```

**get\_low\_voltage\_timer()** → int

Get the existing low voltage timer value from the MCU.

The low voltage timer controls the number of seconds that the measured voltage can be lower than the shutdown threshold before a forced shutdown will occur. This method uses the LPMCU protocol discussed in the README and documentation to set the low voltage timer of the sequence MCU.

**Parameters:** None

**Returns:** The low voltage timer value of the device in seconds. The low voltage timer can be configured between 1 - 1048575 seconds. A value < 0 indicates an error in the command or response.

**Return type:** int

## Example

```
>>> low_voltage_timer = my_auto.get_low_voltage_timer()
>>> print(f"Low Voltage Timer: {low_voltage_timer}")
Low Voltage Timer: 20
```

**get\_shutdown\_voltage()** → int

Get the existing shutdown voltage value from the MCU.

The shutdown voltage value dictates threshold voltage for triggering low-voltage shutdown events. This method uses the LPMCU protocol discussed in the README and documentation to get the shutdown voltage threshold of the sequence MCU.

**Parameters:** None

**Returns:** The shutdown voltage value of the device in centi-volts. The shutdown voltage can be configured between 1.000 - 48.000. A value < 0 indicates an error in the command or response.

**Return type:** int

## Example

```
>>> shutdown_voltage = my_auto.get_shutdown_voltage()
>>> print(f"Shutdown Voltage: {shutdown_voltage}")
Shutdown Voltage: 600
```

**get\_soft\_off\_timer()** → int

Get the existing soft-off timer value from the MCU.

The soft-off timer controls the number of seconds that the ignition input can be low before the mcu requests the system powers down via a virtual power button event. This method uses the LPMCU protocol discussed in the README and documentation to set the soft-off timer of the sequence MCU.

**Parameters:** None

**Returns:** The soft-off timer value of the device in seconds. The soft-off timer can be configured between 1 - 1048575 seconds. A value < 0 indicates an error in the command or response.

**Return type:** int

## Example

```
>>> soft_off_timer = my_auto.get_soft_up_timer()
>>> print(f"Soft Off Timer: {soft_off_timer}")
Soft Off Timer: 5
```

**get\_start\_up\_timer()** → int

Get the start-up timer value from the MCU.

The start-up timer controls the number of seconds that the ignition input must be stable before the system will power on. This method uses the LPMCU protocol discussed in the README and documentation to get the start up timer.

**Parameters:** None

**Returns:** the start-up timer value of the device in seconds. the start-up timer can be configured between 1 - 1048575 seconds. A value < 0 indicates an error in the command or response.

**Return type:** int

## Example

```
>>> start_up_timer = my_auto.get_start_up_timer()
>>> print(f"Start Up Timer: {start_up_timer}")
Start Up Timer: 10
```

**set\_all\_automotive\_settings(setting\_inputs: list)** → list

Sets all automotive settings based on a provided input list of desired states.

(Note numpy commenting format was used as this was the only way to get the docstring to render correctly in Sphinx)

## Args

**setting\_input** : list

A list of values corresponding to the desired states for:

- automotive\_mode\_enabled (int)
- low\_power\_enabled (int)
- start\_up\_timer\_seconds (int)
- soft\_off\_timer\_seconds (int)
- hard\_off\_timer\_seconds (int)
- shutdown\_voltage\_mv (int)

## Returns

**operation\_results** : list

A list of integer status codes from each `set_*` method, detailing the outcome of each operation. The items in this list correspond to:

- Result of `set_automotive_mode()` (int): Status code for setting automotive mode.

- Result of `set_low_power_enable()` (int): Status code for enabling/disabling low power mode.
- Result of `set_start_up_timer()` (int): Status code for setting the start-up timer.
- Result of `set_soft_off_timer()` (int): Status code for setting the soft-off timer.
- Result of `set_hard_off_timer()` (int): Status code for setting the hard-off timer.
- Result of `set_shutdown_voltage()` (int): Status code for setting the shutdown voltage.

### Example

```
>>> setting_inputs = [1, 1, 10, 5, 15, 12000] # 12000mV for 12.0V
>>> results = my_auto.set_all_automotive_settings(setting_inputs)
>>> print(results)
[0, 0, 0, 0, 0, 0]
```

**set\_automotive\_mode** (amd: int) → int

Set the automotive mode of the device.

Automotive-mode enables or disables system automotive features. This method uses the LPMCU protocol discussed in the README and documentation to set whether the device is in automotive mode or not.

**Parameters:** **amd** (int) – The automotive mode to set. 0 turn automotive mode off, 1 turn automotive mode on.

**Returns:** The status of the command. 0 indicates success, < 0 indicates an error in the command or response.

**Return type:** int

**Raises:**

- **ValueError** – If the input parameter is not a valid integer or is out of range.
- **TypeError** – If the input parameter is not of type int.

### Example

```
>>> auto_mode = my_auto.set_automotive_mode(1)
>>> print(f"Automotive Mode Set to: {auto_mode}")
Automotive Mode Set to: 0
```

**set\_hard\_off\_timer** (hot: int) → int

Set the hard-off timer value in the sequence MCU.

The number of seconds that the ignition input can be low before the MCU will force the system to power down. This method uses the LPMCU protocol discussed in the README and documentation to set the hard off timer of the sequence MCU.

**Parameters:** **hot** (int) – The hard off timer to set. 1 - 1048575 seconds.

**Returns:** The status of the command. 0 indicates success, < 0 indicates an error in the command or response.

**Return type:** int

**Raises:**

- **ValueError** – If the input parameter is not a valid integer or is out of range.
- **TypeError** – If the input parameter is not of type int.

### Example

```
>>> status = my_auto.set_hard_off_timer(15)
>>> print(f"Success" if status == 0 else f"Error: {status}")
Success
```

**set\_low\_power\_enable** (lpe: int) → int

Set the low power enable status in the sequence MCU.

Low Power Enable enables entering a very low power state when the system powers off. The system can only wake from the power-button and the ignition switch when in this power state. This method uses the LPMCU protocol discussed in the README and documentation to set the low power enable status of the sequence MCU.

**Parameters:** **lpe** (int) – The low power enable status to set. 0 turn low power mode off, 1 turn low power mode on.

**Returns:** The status of the command. 0 indicates success, < 0 indicates an error in the command or response.

**Return type:** int

## Example

```
>>> status = my_auto.set_low_power_enable(1)
>>> print(f"Success" if status == 0 else f"Error: {status}")
Success
```

**set\_low\_voltage\_timer** (lvt: int) → int

Set the low voltage timer value in the sequence MCU.

The number of seconds that the measured voltage can be lower than the shutdown threshold before a forced shutdown will occur. This method uses the LPMCU protocol discussed in the README and documentation to set the low voltage timer of the sequence MCU.

**Parameters:** **lvt** (int) – The low voltage timer to set. 1 - 1048575 seconds.

**Returns:** The status of the command. 0 indicates success, < 0 indicates an error in the command or response.

**Return type:** int

**Raises:**

- **ValueError** – If the input parameter is not a valid integer or is out of range.
- **TypeError** – If the input parameter is not of type int.

## Example

```
>>> status = my_auto.set_low_voltage_timer(20)
>>> print(f"Success" if status == 0 else f"Error: {status}")
Success
```

**set\_shutdown\_voltage** (sdv: int) → int

Set the shutdown voltage value in the sequence MCU.

The shutdown voltage value dictates threshold voltage for triggering low-voltage shutdown events. This method uses the LPMCU protocol discussed in the README and documentation to set the shutdown voltage threshold of the sequence MCU.

**Parameters:** **sdv** (int) – The shutdown voltage to set. 1.000 - 48.000 volts.

**Returns:** The status of the command. 0 indicates success, < 0 indicates an error in the command or response.

**Return type:** int

**Raises:**

- **ValueError** – If the input parameter is not a valid integer or is out of range.
- **TypeError** – If the input parameter is not of type int.

## Example

```
>>> status = my_auto.set_shutdown_voltage(12)
>>> print(f"Success" if status == 0 else f"Error: {status}")
Success
```

**set\_soft\_off\_timer** (sot: int) → int

Set the soft-off timer value in the sequence MCU.

The soft-off timer controls the number of seconds that the ignition input can be low before the mcu requests the system powers down via a virtual power button event. This method uses the LPMCU protocol discussed in the README and documentation to set the soft off timer of the sequence MCU.

**Parameters:** **sot** (*int*) – The soft off timer to set. 1 - 1048575 seconds.

**Returns:** The status of the command. 0 indicates success, < 0 indicates an error in the command or response.

**Return type:** int

**Raises:**

- **ValueError** – If the input parameter is not a valid integer or is out of range.
- **TypeError** – If the input parameter is not of type int.

## Example

```
>>> status = my_auto.set_soft_off_timer(5)
>>> print(f"Success" if status == 0 else f"Error: {status}")
Success
```

**set\_start\_up\_timer** (sut: int) → int

Set the start-up timer value in the sequence MCU.

The start-up timer controls the number of seconds that the ignition input must be stable before the system will power on. This method uses the LPMCU protocol discussed in the README and documentation to set the start up timer of the sequence MCU.

**Parameters:** **sut** (*int*) – The start up timer to set. 1 - 1048575 seconds.

**Returns:** The status of the command. 0 indicates success, < 0 indicates an error in the command or response.

**Return type:** int

**Raises:**

- **ValueError** – If the input parameter is not a valid integer or is out of range.
- **TypeError** – If the input parameter is not of type int.

## Example

```
>>> status = my_auto.set_start_up_timer(10)
>>> print(f"Success" if status == 0 else f"Error: {status}")
Success
```

**show\_info** () → None

Show information about the target MCU depending on the child class.

This method is used to display information about the target MCU. It is an inheritable method provided to the base class. It will print a manual to the console, which is a text file with the documentation of the target MCU. It closely interacts with the `_read_files` method, which is also an inheritable method.

**Parameters:** **None**

**Returns:** None

**Raises:**

- **FileNotFoundError** – If the file is not found.
- **IOError** – If there is an error reading the file.
- **Exception** – For any other exceptions that may occur.

## src.command\_set Module

OnLogic M031 Manager Constants

This file contains constants used throughout the OnLogic M031 Manager. It includes protocol constants, command kinds, status types, target indices, and boundary types.

**Frame format is as follows:**

sof (1 Byte), crc (1 Byte), len (1 Byte), kind (1 Byte), data (0-8 Bytes), NACK (1 Byte)

**class** src.command\_set.**BoundaryTypes**

Bases: **object**

The BoundaryTypes class specifies numeric boundaries for command parameters, such as binary values, digital I/O pin ranges, decimal values, and byte values

**AUTOMOTIVE\_TIMER\_RANGE** = (1, 1048576)

**BASE\_FRAME\_SIZE** = 4

**BINARY\_VALUE\_RANGE** = (0, 1)

**BYTE\_VALUE\_RANGE** = (0, 255)

**DECIMAL\_VALUE\_RANGE** = (0, 9)

**DIGITAL\_IO\_PIN\_RANGE** = (0, 7)

**class** src.command\_set.**Kinds**

Bases: **object**

The Kinds class categorizes the command classifiers for automotive and DIO commands, providing a clear mapping of command types to their respective identifiers.

**The message kind can indicate:**

1. That a message was an error
2. How to decode the incoming body data

**ERR\_ZERO\_KIND** = 0

**GET\_AUTOMOTIVE\_MODE** = 6

**GET\_DI** = 32

**GET\_DI\_CONTACT** = 80

**GET\_DO** = 34

**GET\_DO\_CONTACT** = 81

**GET\_FIRMWARE\_VERSION** = 3

**GET\_HARD\_OFF\_TIMER** = 14

**GET\_IGNITION\_STATE** = 2



```

GET_INPUT_VOLTAGE = 1

GET_LOW_POWER_ENABLE = 8

GET_LOW_VOLTAGE_TIMER = 16

GET_SHUTDOWN_VOLTAGE = 18

GET_SOFT_OFF_TIMER = 12

GET_START_UP_TIMER = 10

SET_AUTOMOTIVE_MODE = 7

SET_DI_CONTACT = 82

SET_DO = 33

SET_DO_CONTACT = 83

SET_HARD_OFF_TIMER = 15

SET_LOW_POWER_ENABLE = 9

SET_LOW_VOLTAGE_TIMER = 17

SET_SHUTDOWN_VOLTAGE = 19

SET_SOFT_OFF_TIMER = 13

SET_START_UP_TIMER = 11

```

```
class src.command_set.ProtocolConstants
```

Bases: `object`

The ProtocolConstants class defines the baud-rate, DIO card device descriptor, transmission values, and serial check values used in the communication protocol with the DIO Card.

## Note

MAX\_NACK\_CLEARANCES number isn't magic. In the worst possible case, the MCU may have an entire maximum-length frame buffered containing only SHELL\_NACK. To be confident that we've received the entire frame and the MCU is ready to accept our next command, we need to receive 4 bytes (header) + 255 bytes (payload) of NACKs. This said, we have yet to implement the the worst-case scenario in Python Package.

```

BAUDRATE = 115200

DIO_MCU_VID_PID_CDC = '353F:A105'

MAX_NACK_CLEARANCES = 259

NACKS_NEEDED = 5

NUM_NACKS = 259

RESPONSE_FRAME_LEN = 7

SHELL_ACK = 13

SHELL_NACK = 7

```

```
SHELL_SOF = 1
```

```
STANDARD_DELAY = 0.003
```

```
STANDARD_NACK_CLEARANCES = 64
```

```
TIME_THRESHOLD = 2.5
```

```
class src.command_set.StatusTypes
```

Bases: `object`

The StatusTypes class enumerates the various status codes that can be returned during the lifespan of the communication process, indicating success (0) or different types of errors (< 0).

```
FORMAT_NONE_ERROR = -9
```

```
RECV_FRAME_ACK_ERROR = -5
```

```
RECV_FRAME_CRC_ERROR = -4
```

```
RECV_FRAME_NACK_ERROR = -3
```

```
RECV_FRAME_SOF_ERROR = -6
```

```
RECV_FRAME_VALUE_ERROR = -8
```

```
RECV_PARTIAL_FRAME_VALIDATION_ERROR = -7
```

```
RECV_UNEXPECTED_PAYLOAD_ERROR = -2
```

```
SEND_CMD_FAILURE = -1
```

```
SUCCESS = 0
```

```
class src.command_set.TargetIndices
```

Bases: `object`

The TargetIndices class defines indices used to isolate target data within received frames,

```
CRC = 1
```

```
KIND = 3
```

```
LEN = 2
```

```
NACK = -1
```

```
PAYLOAD_START = 4
```

```
PENULTIMATE = -2
```

```
RECV_PAYLOAD_LEN = 2
```

```
SOF = 0
```

## src.logging\_util Module

An optional logging utility for the OnLogic M031 Manager.

This module provides the optional LoggingUtil class, which is designed to configure and manage logging for the OnLogic M031 Manager.

## Note

it is completely optional and not required for the use of the OnLogic M031 Manager, but is provided for convenience and ease of use. The user can make create their own custom logger if desired.

```
class src.logging_util.LoggingUtil (logger_name, logger_level, handler_mode)
```

Bases: **object**

LoggingUtil class for configuring logging in the OnLogic M031 Manager.

This class provides methods to set up a logger with a specified name, level, and handlers.

**logger\_name**

The name of the logger. It is converted to lowercase and stripped of whitespace. Note, please use 'root' for the root logger unless you are willing to add a field to the target class to allow for a custom logger utility.

**Type:** str

**logger\_level**

The logging level (e.g., 'info', 'debug', 'error'). It is converted to lowercase and stripped of whitespace.

**Type:** str

**handler\_mode**

The mode for the handler (e.g., 'console', 'file', 'both'). It is converted to lowercase and stripped of whitespace.

**Type:** str

**logger**

The configured logger instance.

**Type:** logging.Logger

**format**

The format for log messages, specified in the initialization of the class.

**Type:** str

## Example

For debug logging to console:

```
config_logger = LoggingUtil(
    logger_name='root', logger_level="DEBUG", handler_mode="CONSOLE"
)
config_logger.config_logger_elements()
```

For info logging to file:

```
config_logger = LoggingUtil(
    logger_name='root', logger_level="INFO", handler_mode="FILE"
)
config_logger.config_logger_elements()
```

For error logging to both console and file:

```
config_logger = LoggingUtil(
    logger_name='root', logger_level="ERROR", handler_mode="BOTH"
)
config_logger.config_logger_elements()
```

**`__init__(logger_name, logger_level, handler_mode)`**

Initialize the LoggingUtil class with the specified logger name, level, and handler mode. :param logger\_name: The name of the logger. It is converted to lowercase and stripped of whitespace.

Note, please use 'root' for the root logger unless you are willing to add a field to the target class to allow for a custom logger utility.

**Parameters:**

- **logger\_level** (*str*) – The logging level (e.g., 'info', 'debug', 'error'). It is converted to lowercase and stripped of whitespace.
- **handler\_mode** (*str*) – The mode for the handler (e.g., 'console', 'file', 'both'). It is converted to lowercase and stripped of whitespace.

**`config_logger_elements()`** → `Logger` | `None`

Configure the logger with the specified name, level, and handlers.

This method will set up a logger based on logging configurations provided to the LoggingUtil class constructor. It supports 3 logging level configurations, DEBUG, INFO, and ERROR. And the log messages can be directed at the Console, an external file, or both.

It chooses the file handler object based off cleaned input mode selection. The format of the logger is set to match the default format defined in the class. After, it will set the logger as a null handler if there is handler objects configured. It will make sure not to propagate the logging configurations or messages to other loggers.

**Params:**

None

**Returns:** The configured logger instance.

**Return type:** `logging.Logger`

## Example

```
config_logger = LoggingUtil(
    logger_name='root', logger_level="DEBUG", handler_mode="CONSOLE"
) config_logger.config_logger_elements()
```

## Indices and tables

- `genindex`
- `modindex`
- `search`

# Index

`__del__()`  
(`src.onlogic_m031_manager.OnLogicM031Manager`  
method)  
(`src.OnLogicM031Manager` method)  
`__init__()` (`src.automotive_handler.AutomotiveHandler`  
method)  
(`src.AutomotiveHandler` method)  
(`src.dio_handler.DioHandler` method)  
(`src.DioHandler` method)  
(`src.logging_util.LoggingUtil` method)  
(`src.LoggingUtil` method)  
(`src.onlogic_m031_manager.OnLogicM031Manag`  
`er` method)  
(`src.OnLogicM031Manager` method)  
`__str__()`  
(`src.onlogic_m031_manager.OnLogicM031Manager`  
method)  
(`src.OnLogicM031Manager` method)

## A

`AUTOMOTIVE_TIMER_RANGE`  
(`src.command_set.BoundaryTypes` attribute)  
`AutomotiveHandler` (class in `src`)  
(class in `src.automotive_handler`)

## B

`BASE_FRAME_SIZE`  
(`src.command_set.BoundaryTypes` attribute)  
`BAUDRATE` (`src.command_set.ProtocolConstants`  
attribute)  
`BINARY_VALUE_RANGE`  
(`src.command_set.BoundaryTypes` attribute)  
`BoundaryTypes` (class in `src.command_set`)  
`BYTE_VALUE_RANGE`  
(`src.command_set.BoundaryTypes` attribute)

## C

`claim()`  
(`src.onlogic_m031_manager.OnLogicM031Manager`  
method)  
(`src.OnLogicM031Manager` method)  
`config_logger_elements()` (`src.logging_util.LoggingUtil`  
method)  
(`src.LoggingUtil` method)

`CRC` (`src.command_set.TargetIndices` attribute)

## D

`DECIMAL_VALUE_RANGE`  
(`src.command_set.BoundaryTypes` attribute)  
`DIGITAL_IO_PIN_RANGE`  
(`src.command_set.BoundaryTypes` attribute)  
`DIO_MCU_VID_PID_CDC`  
(`src.command_set.ProtocolConstants` attribute)  
`DioHandler` (class in `src`)  
(class in `src.dio_handler`)

## E

`ERR_ZERO_KIND` (`src.command_set.Kinds` attribute)

## F

`format` (`src.logging_util.LoggingUtil` attribute)  
(`src.LoggingUtil` attribute)  
`FORMAT_NONE_ERROR`  
(`src.command_set.StatusTypes` attribute)

## G

`get_all_automotive_settings()`  
(`src.automotive_handler.AutomotiveHandler` method)  
(`src.AutomotiveHandler` method)  
`get_all_input_states()` (`src.dio_handler.DioHandler`  
method)  
(`src.DioHandler` method)  
`get_all_io_states()` (`src.dio_handler.DioHandler`  
method)  
(`src.DioHandler` method)  
`get_all_output_states()` (`src.dio_handler.DioHandler`  
method)  
(`src.DioHandler` method)  
`GET_AUTOMOTIVE_MODE` (`src.command_set.Kinds`  
attribute)  
`get_automotive_mode()`  
(`src.automotive_handler.AutomotiveHandler` method)  
(`src.AutomotiveHandler` method)  
`GET_DI` (`src.command_set.Kinds` attribute)  
`get_di()` (`src.dio_handler.DioHandler` method)  
(`src.DioHandler` method)  
`GET_DI_CONTACT` (`src.command_set.Kinds` attribute)  
`get_di_contact()` (`src.dio_handler.DioHandler` method)  
(`src.DioHandler` method)  
`GET_DO` (`src.command_set.Kinds` attribute)

`get_do()` (`src.dio_handler.DioHandler` method)  
(`src.DioHandler` method)

`GET_DO_CONTACT` (`src.command_set.Kinds` attribute)

`get_do_contact()` (`src.dio_handler.DioHandler` method)  
(`src.DioHandler` method)

`GET_FIRMWARE_VERSION` (`src.command_set.Kinds` attribute)

`GET_HARD_OFF_TIMER` (`src.command_set.Kinds` attribute)

`get_hard_off_timer()`  
(`src.automotive_handler.AutomotiveHandler` method)  
(`src.AutomotiveHandler` method)

`GET_IGNITION_STATE` (`src.command_set.Kinds` attribute)

`GET_INPUT_VOLTAGE` (`src.command_set.Kinds` attribute)

`GET_LOW_POWER_ENABLE`  
(`src.command_set.Kinds` attribute)

`get_low_power_enable()`  
(`src.automotive_handler.AutomotiveHandler` method)  
(`src.AutomotiveHandler` method)

`GET_LOW_VOLTAGE_TIMER`  
(`src.command_set.Kinds` attribute)

`get_low_voltage_timer()`  
(`src.automotive_handler.AutomotiveHandler` method)  
(`src.AutomotiveHandler` method)

`GET_SHUTDOWN_VOLTAGE`  
(`src.command_set.Kinds` attribute)

`get_shutdown_voltage()`  
(`src.automotive_handler.AutomotiveHandler` method)  
(`src.AutomotiveHandler` method)

`GET_SOFT_OFF_TIMER` (`src.command_set.Kinds` attribute)

`get_soft_off_timer()`  
(`src.automotive_handler.AutomotiveHandler` method)  
(`src.AutomotiveHandler` method)

`GET_START_UP_TIMER` (`src.command_set.Kinds` attribute)

`get_start_up_timer()`  
(`src.automotive_handler.AutomotiveHandler` method)  
(`src.AutomotiveHandler` method)

`get_version()`  
(`src.onlogic_m031_manager.OnLogicM031Manager` method)  
(`src.OnLogicM031Manager` method)

## H

`handler_mode` (`src.logging_util.LoggingUtil` attribute)  
(`src.LoggingUtil` attribute)

## I

`is_setup`  
(`src.onlogic_m031_manager.OnLogicM031Manager` attribute)  
(`src.OnLogicM031Manager` attribute)

## K

`KIND` (`src.command_set.TargetIndices` attribute)  
`Kinds` (class in `src.command_set`)

## L

`LEN` (`src.command_set.TargetIndices` attribute)

`list_all_available_ports()`  
(`src.onlogic_m031_manager.OnLogicM031Manager` method)  
(`src.OnLogicM031Manager` method)

`logger` (`src.logging_util.LoggingUtil` attribute)  
(`src.LoggingUtil` attribute)

`logger_level` (`src.logging_util.LoggingUtil` attribute)  
(`src.LoggingUtil` attribute)

`logger_name` (`src.logging_util.LoggingUtil` attribute)  
(`src.LoggingUtil` attribute)

`LoggingUtil` (class in `src`)  
(class in `src.logging_util`)

## M

`MAX_NACK_CLEARANCES`  
(`src.command_set.ProtocolConstants` attribute)

### module

`src`  
`src.automotive_handler`  
`src.command_set`  
`src.dio_handler`  
`src.logging_util`  
`src.onlogic_m031_manager`

## N

`NACK` (`src.command_set.TargetIndices` attribute)

`NACKS_NEEDED`  
(`src.command_set.ProtocolConstants` attribute)

`NUM_NACKS` (`src.command_set.ProtocolConstants` attribute)

## O

OnLogicM031Manager (class in src)  
(class in src.onlogic\_m031\_manager)

## P

PAYLOAD\_START (src.command\_set.TargetIndices attribute)

PENULTIMATE (src.command\_set.TargetIndices attribute)

port  
(src.onlogic\_m031\_manager.OnLogicM031Manager attribute)

(src.OnLogicM031Manager attribute)

ProtocolConstants (class in src.command\_set)

## R

RECV\_FRAME\_ACK\_ERROR  
(src.command\_set.StatusTypes attribute)

RECV\_FRAME\_CRC\_ERROR  
(src.command\_set.StatusTypes attribute)

RECV\_FRAME\_NACK\_ERROR  
(src.command\_set.StatusTypes attribute)

RECV\_FRAME\_SOF\_ERROR  
(src.command\_set.StatusTypes attribute)

RECV\_FRAME\_VALUE\_ERROR  
(src.command\_set.StatusTypes attribute)

RECV\_PARTIAL\_FRAME\_VALIDATION\_ERROR  
(src.command\_set.StatusTypes attribute)

RECV\_PAYLOAD\_LEN  
(src.command\_set.TargetIndices attribute)

RECV\_UNEXPECTED\_PAYLOAD\_ERROR  
(src.command\_set.StatusTypes attribute)

release()  
(src.onlogic\_m031\_manager.OnLogicM031Manager method)

(src.OnLogicM031Manager method)

RESPONSE\_FRAME\_LEN  
(src.command\_set.ProtocolConstants attribute)

## S

SEND\_CMD\_FAILURE  
(src.command\_set.StatusTypes attribute)

serial\_connection\_label  
(src.automotive\_handler.AutomotiveHandler attribute)

(src.AutomotiveHandler attribute)

(src.dio\_handler.DioHandler attribute)

(src.DioHandler attribute)

(src.onlogic\_m031\_manager.OnLogicM031Manager attribute)

(src.OnLogicM031Manager attribute)

set\_all\_automotive\_settings()  
(src.automotive\_handler.AutomotiveHandler method)

(src.AutomotiveHandler method)

set\_all\_output\_states() (src.dio\_handler.DioHandler method)

(src.DioHandler method)

SET\_AUTOMOTIVE\_MODE (src.command\_set.Kinds attribute)

set\_automotive\_mode()  
(src.automotive\_handler.AutomotiveHandler method)

(src.AutomotiveHandler method)

SET\_DI\_CONTACT (src.command\_set.Kinds attribute)

set\_di\_contact() (src.dio\_handler.DioHandler method)

(src.DioHandler method)

SET\_DO (src.command\_set.Kinds attribute)

set\_do() (src.dio\_handler.DioHandler method)

(src.DioHandler method)

SET\_DO\_CONTACT (src.command\_set.Kinds attribute)

set\_do\_contact() (src.dio\_handler.DioHandler method)

(src.DioHandler method)

SET\_HARD\_OFF\_TIMER (src.command\_set.Kinds attribute)

set\_hard\_off\_timer()  
(src.automotive\_handler.AutomotiveHandler method)

(src.AutomotiveHandler method)

SET\_LOW\_POWER\_ENABLE  
(src.command\_set.Kinds attribute)

set\_low\_power\_enable()  
(src.automotive\_handler.AutomotiveHandler method)

(src.AutomotiveHandler method)

SET\_LOW\_VOLTAGE\_TIMER  
(src.command\_set.Kinds attribute)

set\_low\_voltage\_timer()  
(src.automotive\_handler.AutomotiveHandler method)

(src.AutomotiveHandler method)

SET\_SHUTDOWN\_VOLTAGE  
(src.command\_set.Kinds attribute)

set\_shutdown\_voltage()  
(src.automotive\_handler.AutomotiveHandler method)

(src.AutomotiveHandler method)

SET\_SOFT\_OFF\_TIMER (src.command\_set.Kinds attribute)

set\_soft\_off\_timer()  
(src.automotive\_handler.AutomotiveHandler method)

(src.AutomotiveHandler method)

SET\_START\_UP\_TIMER (src.command\_set.Kinds attribute)

set\_start\_up\_timer()  
(src.automotive\_handler.AutomotiveHandler method)  
(src.AutomotiveHandler method)

SHELL\_ACK (src.command\_set.ProtocolConstants attribute)

SHELL\_NACK (src.command\_set.ProtocolConstants attribute)

SHELL\_SOF (src.command\_set.ProtocolConstants attribute)

show\_info()  
(src.automotive\_handler.AutomotiveHandler method)  
(src.AutomotiveHandler method)  
(src.dio\_handler.DioHandler method)  
(src.DioHandler method)  
(src.onlogic\_m031\_manager.OnLogicM031Manager method)  
(src.OnLogicM031Manager method)

SOF (src.command\_set.TargetIndices attribute)

**src**

module

**src.automotive\_handler**

module

**src.command\_set**

module

**src.dio\_handler**

module

**src.logging\_util**

module

**src.onlogic\_m031\_manager**

module

STANDARD\_DELAY  
(src.command\_set.ProtocolConstants attribute)

STANDARD\_NACK\_CLEARANCES  
(src.command\_set.ProtocolConstants attribute)

StatusTypes (class in src.command\_set)

SUCCESS (src.command\_set.StatusTypes attribute)

## T

TargetIndices (class in src.command\_set)

TIME\_THRESHOLD  
(src.command\_set.ProtocolConstants attribute)



# Python Module Index

## s

[src](#)

[src.automotive\\_handler](#)

[src.command\\_set](#)

[src.dio\\_handler](#)

[src.logging\\_util](#)

[src.onlogic\\_m031\\_manager](#)