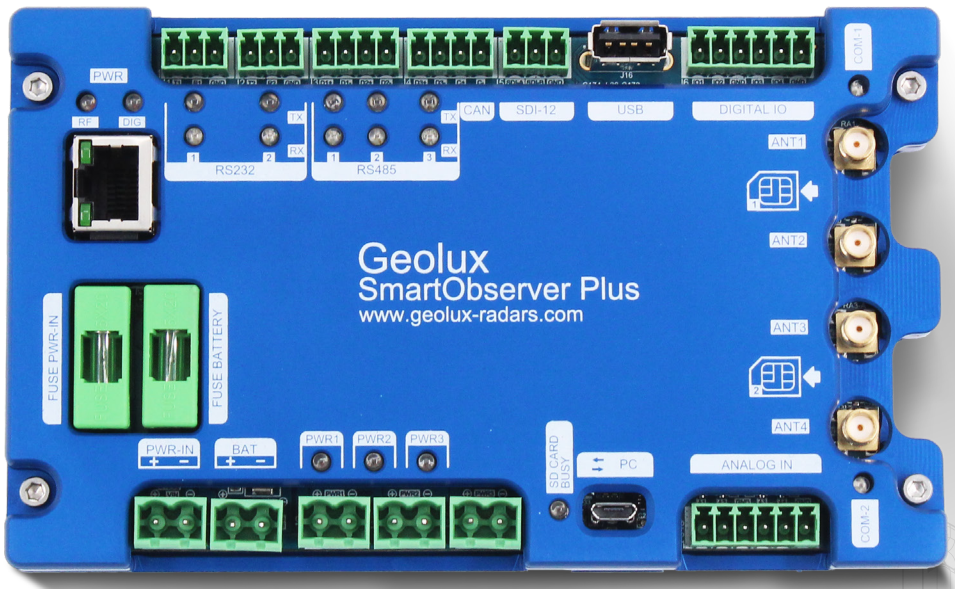
Geolux d.o.o. | Ljudevita Gaja 62 | HR-10430 Samobor Croatia | www.geolux-radars.com | geolux@geolux.hr

SmartObserver **Plus**



MicroPython **Reference** Manual

v1.0.0.

1 Introduction

Welcome to the SmartObserverPlus MicroPython Reference Manual. This guide is your starting point for creating, deploying, and managing Python scripts on the SmartObserverPlus datalogger device. Based on a MicroPhython engine, the SmartObserverPlus enables you to write custom Phython scripts that interact with a wide array of sensors, communication interfaces, and hardware features.

NOTE

Using MicroPhyton is not required for operating SmartObserverPlus. It is provided as an option for advanced users who wish to customize and take full control of the datalogger's operation. Regular users can configure the device's behavior through the intuitive, UI-based PC application. For guidlines on using the UI-based datalogger setup, please refer to the SmartObserverPlus User Manual.

In this manual, you will learn:

How the Engine Works:	Understand the dual-program execution model where the system runs a user-provided Python script (User program) followed by a built-in Main program that handles data acquisition, processing, and transmission.
Script Writing and Execution:	Get acquainted with the guidelines for developing your Python scripts-including defining custom hooks to extend functionality-and learn how your code interacts with the system's core functions.
Module and Library Overview:	Explore detailed documentation of both standard MicroPython libraries and proprietary modules provided by Geolux. These modules give you direct access to hardware features such as analog and digital I/O, SDI-12, Modbus, CAN bus, RS-232, FTP, HTTP, storage, and more.
Practical Examples:	Throughout the manual, you'll find examples illustrating how to configure hardware, manage connectivity, and process sensor data, helping you to build robust and efficient applications. Whether you're developing simple measurement routines or comprehensive data logging solutions, this manual provides the information and examples you need to harness the full potential of the SmartObserverPlus device using MicroPython.

Contents

2 Script Execution and Hooks

SmartObserverPlus supports the execution of user-provided Python scripts using an engine based on the MicroPython implementation of the Python interpreter.

NOTE

This engine has some limitations; see the sections below for details.

DEFINITIONS IN CONTEXT OF THIS DOCUMENT

Script: set of textual data written in Python (i.e., source code).
Program: the execution instance produced when a script is run by the Engine; essentially, the program is the dynamic behavior defined by the script.

The Engine runs two programs in sequence:

- User program (optional)
- Main program (built-in)

The Engine is run periodically, based on the configured scan interval parameter. A User-provided Python script is executed by the Engine to create a corresponding User program. If no script is provided, the User program is skipped. Keep in mind that the descriptions of these processes are generalized, and certain details are omitted on purpose.

Flow:

1. **Wake Up:** SmartObserverPlus exits low-power mode
2. **Run User program:** Execute the User script (if provided)
3. **Run Main program:** Execute the built-in Main script
4. **Sleep:** Set a wake-up time, enter low-power mode, and wait to be woken by a timer.
   Then return to step 1.

2.1. User program

A User-provided Python script is executed by the Engine to create a corresponding User program. If no script is provided, the User program is skipped. This User-provided Python script must adhere to the guidelines outlined in the chapters below. We recommend observing industry best practices for writing Python code. The structure and flow of User program is entirely defined by the User-provided Python script to suit your needs. A User-provided Python script does not have to define an entire program - it can also define Hooks. (See Hooks section below.)

If needed, execution of Main program can be blocked (i.e. Main program will not be run) by either:

1. Putting the device into sleep mode:

a. Define a wake-up time by calling `logger.wakeup_after(...)` and then
b. Call `logger.sleep()`

2. Entering an infinite loop:

a. Place an infinite loop (e.g., `while(True):continue`) at the end of the User script

NOTE

Executing an infinite loop will also block periodic execution.

2.2. Main program

The non-modifiable built-in Main script is executed by the Engine to create the Main program. The Main program collects data from instruments, processes the data, stores it locally, and transmits it to a remote server. It then places SmartObserverPlus into a low-power state to conserve energy.

Flow:

1. Read configuration,
2. Turn on power for the Instruments,
3. Let the instruments perform actions (measurement, actuation),
4. Collect instrument results,
5. Turn off power for the Instruments,
6. Save results to persistent storage,
7. Establish data connection,
8. Send results to server,
9. Terminate data connection.

2.3. Watchdog

If, for any reason, a program gets stuck and Engine becomes unresponsive (i.e., if either the User or Main program fails to terminate), the entire Engine will be restarted by the Python Watchdog. The default Python Watchdog timeout interval is set to 120 minutes (2 hours).

The Python Watchdog timeout interval can be set by calling
`logger.watchdog_enable(timeout_interval)`, see chapter on logger module below.
Python Watchdog can be completely disabled by calling `logger.watchdog_disable().`

USE WATCHDOG FUNCTIONS WITH CARE

Since disabling the Python Watchdog can cause program execution to halt completely until a fault clearing action is performed by the User (either by cycling the power supply OR by sending a special command).

2.4. Python Hooks

Hooks are a mechanism that allows your application to extend or customize its behavior by automatically calling user-defined functions if they exist in the global namespace.

Hooks are defined exclusively in the User script. If User-defined Python script contains function definitions for Hooks, these functions are loaded into global namespace before the Main program runs. In our system, certain events trigger hook execution by searching for callable functions with special, predefined names (see List of Hook names below). If such a function exists, the Main program will call it - allowing you to insert custom logic at key points during operation.

For example, the system supports a hook named `on_modem_connected`, the Main program will check the global namespace for a function by that name. If you provide an implementation for `on_modem_connected` within the User script, it will be automatically executed when the corresponding event occurs. If no such function is defined, the system simply continues without executing any hook.

To use hooks:

- **Define the Hook:** Write a function using one of the predefined hook names (e.g. `on_measurements_done, on_modem_connected`, etc.) in your Python code.
- **Implement the Body:** Provide the desired functionality inside that function.
- **Automatic Execution:** When the event corresponding to the hook occurs, the Main program will execute your hook function.

List of hook functions

`on_sms_received(sms_data)`

Called upon SMS reception. This function receives SMS data as its argument and does not return a value.

`on_modem_connected(detailed_modem_status)`

Called when the modem connects. The function receives detailed modem status dictionary as its argument and does not return a value.

`format_ftp_request(instruments_with_measurements)`

This function is executed only if a User-defined FTP server URL is configured in Data Upload settings, and it runs just before data is transmitted to the FTP server. The function receives a dictionary containing measurement data. It is used to generate the file body in the format that best meets your specific requirements. The function must return a string representing the file content to be uploaded via FTP.

`format_http_request(instruments_with_measurements)`

This function is executed only if a User-defined HTTP server URL is configured in Data Upload settings, and it runs just before data is transmitted to the HTTP server. The function receives a dictionary containing measurement data. It is used to generate a custom HTTP request body in the format that best meets your specific requirements. The function must return a string representing the HTTP request body, to be sent via HTTP.

`on_sdi12_error(instrument, result)`

Called when an SDI-12 error occurs. The function receives an instrument dictionary and a list of measurement results, and does not return a value.

`on_sdi12_result(instrument, result)`

Called upon SDI-12 success. The function receives an instrument dictionary and a list of measurement results, and does not return a value.

`on_modbus_error(instrument, result)`

Called when a Modbus error occurs. The function receives an instrument dictionary and a list of measurement results, and does not return a value.

`on_modbus_result(instrument, result)`

Called upon Modbus success. The function receives an instrument dictionary and a list of measurement results, and does not return a value.

`before_logger_sleep(next_wakeup_time)`

Called before entering sleep mode. The function receives an integer representing the next wake-up UNIX timestamp (observerd in UTC time zone, in seconds) and does not return a value.

`on_measurements_done(instruments_with_measurements)`

Called after the measurements are complete. The function receives a dictionary of instrument and measurement data, and does not return a value.

`on_measurements_start()`

Called when the measurements begin. This function takes no arguments and does not return a value.

`on_line_power_down(line, instruments_on_this_line)`

Called when a power line is shut down.The function receives an integer representing the power line identifier and a dictionary of instruments on that line, and does not return a value.

`on_line_power_up(line, instruments_on_this_line)`

Called when a power line is activated. The function receives an integer representing the power line identifier and a dictionary of instruments on that line, and does not return a value.

Hook examples

Examples below are given for User convenience, for illustration purposes only.

```
def on_modem_connected(detailed_modem_status):
    print(“on_modem_connected”)
    print(“Detailed modem status:”, detailed_modem_status)

def on_measurements_start():
    print(“on_measurements_start”)
    print(“Measurements have begun!”)

def on_measurements_done(instruments_with_measurements):
    print(“on_measurements_done”)
    print(“Measurements are done!”)
```

```
def on_modbus_error(instrument, result):
   print("on_modbus_error")
   print("Results:", result)

def on_modbus_result(instrument, result):
   print("on_modbus_result")
   print("Results:", result)

def before_logger_sleep(next_wakeup_time):
    print("before_logger_sleep")
    print("next wakeup time:", next_wakeup_time)

def on_line_power_down(line, instruments_on_this_line):
    print("on_line_power_down")
    print("Line", line, "is turned OFF")

def on_line_power_up(line, instruments_on_this_line):
    print("on_line_power_up")
    print("Line", line, "is turned ON")

def format_http_request(instruments_with_measurements):
    print("format_http_request")
    return "This is a custom body for HTTP request to a user configured HTTP server!"

def format_ftp_request(instruments_with_measurements):
    print("format_ftp_request")
    return "These are contents of a custom formatted file for upload to a user configured
    FTP server!"
```

3 Modules Reference

WARNING:

MicroPython provides built-in modules that mirror the functionality of the Python standard library (e.g. `os`, `time`), as well as MicroPython-specific modules

Most Python standard library modules implement a subset of the functionality of the equivalent Python module, and in a few cases provide some MicroPython-specific extensions (e.g. `array`, `os`).

Due to resource constraints and other limitations, not all standard Python libraries are available

Available standard Python libraries and MicroPython-specific libraries

For detailed descriptions see MicroPython documentation

- `array` - arrays of numeric data
- `binascii` - binary/ASCII conversions, Base64
- `builtins` - builtin functions and exceptions
- `cmath` - mathematical functions for complex numbers
- `collections` - collection and container types
- `deflate` - deflate compression & decompression
- `gc` - control the garbage collector
- `io` - input/output streams
- `json` - JSON encoding and decoding
- `math` - mathematical functions
- `micropython` - access and control MicroPython internals
- `struct` - pack and unpack primitive data types

Libraries provided by Geolux

Geolux provides libraries listed below, that already come installed with the SmartObserverPlus. The modules contain specific functions related to the hardware of the SmartObserverPlus.

- `analog_io` - Analog-to-digital configuration and reading
- `can` - Simple CAN bus interface
- `datalog` - Store, retrieve, and update measurement records
- `digital_io` - Configure and manage digital input counters
- `http` -Send HTTP GET or POST requests
- `internal_storage` - Manage device, instrument, and script configurations
- `logger` - Schedule wakeups, manage time and watchdog
- `modbus` - Read and write Modbus slave devices
- `modem` - Control cellular modem and network status
- `output_power` - Control power outputs, measure current consumption
- `sdi12` - Send commands over SDI-12 sensor bus and process responses
- `serial` - UART port configure, read, and write
- `sntp_client` - Synchronize time via SNTP server query
- `storage` - Perform file operations on external storage
- `udp` - UDP Datagram Transmission Module

NOTE ON FUNCTION CALLS

This Python implementation **DOES NOT PERMIT** function calls with named-keyword arguments (also known as named parameters).

Only **positional arguments** in function calls are permited.

Example of an unsupported function call using named-keyword arguments:

```
import analog_io

# UNSUPPORTED, Here `channel`, `pin_type` are named-keywords for arguments.
# This call will raise an exception.
analog_io.set_AIN_mode(channel=0, pin_type=0)
```

Example of a supported functon call using positional arguments:

```
import analog_io

# SUPPORTED, function call with positional arguments
analog_io.set_AIN_mode(0, 0)
```

For better understanding the User must read function definitions.

3.1. analog_io
Analog-to-digital configuration and reading

Provides functions to configure ADC channels (voltage/current mode), perform high-resolution analog-to-digital conversions, and manage low-power sleep/wakeup modes.

Functions

- `read(channel, duration_sec, filter_type)`
  Reads ADC value from a specified channel.
- `go_to_sleep()`
  Disables ADC timer for data acquisition.
- `wake_up()`
  Enables ADC timer for data acquisition.
- `set_AIN_mode(channel, pin_type)`
  Sets voltage/current mode for input.
- `get_AIN_mode(channel)`
  Retrieves the channel’s configured input mode.
- `set_range(volt_range)`
  Sets voltage range for input.
- `get_voltage_range()`
  Returns voltage range for input.

```
analog_io.read(channel, duration_sec, filter_type)
```

Get conversion result for the given channel, based on acquisition period duration in seconds and filter type. This function takes an ADC channel, the duration of acquisition period in seconds, and the type of filtering to apply. It returns a dictionary containing the status of the read, the ADC value (or “NA” if unsuccessful), and the number of samples that were actually averaged.

Parameters

- **channel:** *integer*, Analog input channel for which the conversion result is measured. Argument value must be 0, 1, 2 or 3.
- **duration_sec:** *integer,* Duration of the acquisition period in seconds, how long to collect samples needed to get the conversion result.
- **filter_type:** *integer*, Argument value must be 0 (last sample), 1 (average mean), 2 (average median), 3 (average RMS) or 4 (maximum).

Return value

- *dictionary* containing:
    - **status:** *integer*, error status 0 (success) or 1 (error)
    - **adc_value:** *float*, conversion result, (or “NA” if unsuccessful)
    - **num_samples_av:** *integer*, number of samples that can be used to calculate result

Raises:

- *TypeError* if any argument is of incorrect type
- *ValueError* if **channel** or **filter_type** value is out-of-range

```
analog_io.go_to_sleep()
```

Turn OFF Timer, stops periodical data acquisiton from ADC.Use this function to place the analog I/O subsystem into a low-power sleep state.

Parameters: *none*
Return value: *none*
Raises: *none*

```
analog_io.wake_up()
```

Turn ON Timer, starts periodical data acquisiton from ADC. Use this function to restore normal operation of the analog I/O subsystem.

Parameters: *none*
Return value: *none*
Raises: *none*

```
analog_io.set_AIN_mode(channel, pin_type)
```

Configure AIN pin (channel) to work in either voltage mode or current mode. If pin is configured to work in voltage mode then set voltage range (i.e. gain configuration in ADC).

Parameters:

- **channel:** *integer*, Analog input channel (AIN pin). Argument value must be 0, 1, 2 or 3.
- **pin_type:** *integer,* Wanted pin mode, Argument value must be 0 (voltage), 1 (current)

Return value:

- *None*, on failure OR
- *boolean*, `True` on success, `False` on internal error

Raises:

- *ValueError* if any argument’s value is out-of-range, or internal error

`analog_io.get_AIN_mode(channel)`

Retrieves the current analog input (AIN) mode for the specified channel, which can be either *voltage mode* or *current mode*.

Parameters:

- **channel:** *integer*, Analog input channel (AIN pin). Argument value must be 0, 1, 2 or 3.

Return value:

- *integer,* representing the current analog input mode / pin mode 0 (voltage), 1 (current)

Raises:

- *TypeError* if any argument is of incorrect type
- *ValueError* if **channel** value is out-of-range

`analog_io.set_range(volt_range)`

Sets voltage range for all channels. If this function is not called, then the default voltage range is set to 1 (+/-2.048 V).

Parameters:

- **volt_range:** *integer*, Wanted voltage range, Argument value must be 1 (+/-2.048 V (default)), 2 (+/-1.024 V), 3 (+/-0.512 V).

Return value:

- *boolean,* `True` if voltage range was changed, `False` if voltage range was unchanged

Raises:

- *TypeError* if **volt_range** is of incorrect type

`analog_io.get_voltage_range()`

Returns voltage range for input.

Parameters: *none*

Return value:

- *integer,* Retrieved voltage range, value range is 1 (+/-2.048 V), 2 (+/-1.024 V), 3 (+/-0.512 V).

Raises: *none*

Example

1. Set the AIN mode on channel 0: pin type 0 (voltage mode).
2. Retrieve the actual AIN mode for channel 0.
3. Set the Voltage range to 2 (+/-1.024 V).
4. Retrieve the actual voltage range.
5. Enable ADC timer.
6. Delay for 10 seconds, allowing to acquire some data.
7. Disable ADC timer.
8. Read a high-resolution ADC value from channel 0 using 100 samples and filter type 0 (last sample).
9. Read a high-resolution ADC value from channel 0 using 100 samples and filter type 1 (average mean).
10. Read a high-resolution ADC value from channel 0 using 100 samples and filter type 2 (average median).
11. Read a high-resolution ADC value from channel 0 using 100 samples and filter type 3 (average RMS).
12. Read a high-resolution ADC value from channel 0 using 100 samples and filter type 4 (maximum).

```
import analog_io
import logger  # Import logger for delay functionality

filter_type_string_dict = {
    0: “last sample”,
    1: “average mean”,
    2: “average median”,
    3: “average RMS”,
    4: “maximum”
}

# Set the AIN mode on channel 0: pin type 0 (voltage mode).
try:
    if analog_io.set_AIN_mode(0, 0):
        print(“AIN mode set successfully.”)
except Exception as e:
    print(“Error in analog_io.set_AIN_mode:”, e)

# Retrieve the actual AIN mode for channel 0.
try:
    actual_mode = analog_io.get_AIN_mode(0)
    print(“Actual AIN Mode:”, actual_mode)
except Exception as e:
    print(“Error in analog_io.get_AIN_mode:”, e)

# Set the Voltage range to 2 (+/-1.024 V).
try:
    if analog_io.set_range(2):
        print(“Voltage range changed successfully.”)
    else:
        print(“Voltage range not changed.”)
except Exception as e:
    print(“Error in analog_io.set_range:”, e)

# Retrieve the actual voltage range.
try:
    volt_range = analog_io.get_voltage_range()
    print(“Actual Voltage range:”, volt_range)
except Exception as e:
    print(“Error in analog_io.get_voltage_range:”, e)

# Turn ON timer for analog_io hardware, starts periodical acquisition of data.
analog_io.wake_up()

# Wait 10 seconds before reading the ADC value.
try:
    print(“Waiting 10 seconds...”)
    logger.delay_ms(10000)  # 10 seconds delay (10,000 milliseconds)
except Exception as e:
    print(“Error in logger.delay_ms:”, e)

# Turn OFF timer for analog_io hardware, stops periodical acquisition of data.
analog_io.go_to_sleep()

# Read a high-resolution ADC value from channel 0 using 10 seconds for acquisition period and
variying filter type 0 - 4.
for filter_type in range(0,5):
    try:
        adc_result = analog_io.read(0, 10, filter_type)
        print(“ADC result,”, filter_type_string_dict.get(filter_type), “:”)
        print(“   status:”, adc_result.get(“status”))
        print(“   adc_value:”, adc_result.get(“adc_value”))
        print(“   num_samples_av:”, adc_result.get(“num_samples_av”))
    except Exception as e:
        print(“Error in analog_io.read:”, e)
```

3.2. CANbus

Implements a simple interface for setting CAN bus bitrate and transmitting/receiving CAN messages.

Functions

- `set_bitrate(bit_rate)`
  Sets the CAN bus bitrate
- `send_msg(message_id, data_size, data_out)`
  Sends a message on the CAN bus
- `read_msg()`
  Reads & returns a message from the CAN bus.

`can.set_bitrate(bit_rate)`

Sets the bitrate of CAN bus hardware module.

Parameters:

- **bit_rate:** *integer*, bitrate to set the CAN. Argument value must be greater than 0.

Return value:

- *boolean*, `True` on success

Raises:

- *TypeError* if **bit_rate** is of incorrect type
- *ValueError* if **bit_rate** value is out-of-range

`can.send_msg(message_id, data_size, data_out)`

Sends message on the CAN bus

Parameters:

- **message_id:** *integer*, CAN message ID, 11-bit or 29-bit number
- **data_size:** *integer*, The number of data bytes in the message.
- **data_out:** *bytearray*, The message data.

Return value:

- *boolean*, `True` if the message was sent successfully.

Raises:

- *TypeError* if any argument is of incorrect type
- *ValueError* if any argument's value is out-of-range

`can.read_msg()`

Reads & returns a message from CAN bus

Parameters: *none*
Return value

- *dictionary* containing following entries:
  - **msg_ID:** *integer*, message ID of the received CAN packet
  - **data_length:** *integer*, number of data bytes in the received CAN packet, can be between 0 and 8.
  - **data:** *bytearray*, data payload of a CAN packet

Raises: *none*

Example

This example uses CAN interface to configure bitrate, sends a message with ID 0x100 and then reads a response.

```
import can
import logger  # Using logger.delay_ms() for waiting

# Set the CAN bitrate.
try:
    # Provide the bitrate (e.g., 500000 for 500 kbps)
    result = can.set_bitrate(500000)
    if result:
        print(“CAN bitrate set successfully.”)
    else:
        print(“Failed to set CAN bitrate.”)
except Exception as e:
    print(“Error in can.set_bitrate:”, e)

# Send a CAN message.
try:
    # Provide message ID, data size, and message data
    msg_id = 0x100
    data = bytearray([0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88])
    data_size = len(data)
    if can.send_msg(msg_id, data_size, data):
        print(“CAN message sent successfully.”)
    else:
        print(“Failed to send CAN message.”)
except Exception as e:
    print(“Error in can.send_msg:”, e)

# Read a CAN message, trying up to 10 times with a 1-second delay between attempts.
try:
    attempts = 0
    message = None
    while attempts < 10:
        message = can.read_msg()
         if message is not None and message.get(“data_length”) is not None and message.get(“data_
length”) is not 0:
            break
        attempts += 1
        print(“No message received, waiting for 1 seconds (attempt”, attempts, “ of 10)...”)
        logger.delay_ms(5000)  # Wait for 1000 milliseconds (1 second)

    if message is not None and message.get(“msg_ID”) is not None:
        print(“Received CAN message:”)
        print(“   msg_ID:”, message.get(“msg_ID”))
        print(“   data_length:”, message.get(“data_length”))
        print(“   data:”, message.get(“data”))
    else:
        print(“No CAN message received after 10 attempts.”)
except Exception as e:
    print(“Error in can.read_msg:”, e)
```

3.3. Data logging

Manages the clearing, insertion, retrieval, and status update of timestamped measurement records.

Functions

- `insert(time, values, cfg_id)`
  Inserts a record (timestamp, values, configuration).
- `set_status(record_time, flags, status_time)`
  Updates an existing record's status fields.
- `read(start_time, count, unconfirmed_only, read_status_flags)`
  Reads datalog entries with optional status filtering.
- `get_range()`
  Returns earliest and latest stored timestamps.
- `clear()`
  Removes all datalog entries.

```
datalog.insert(time, values, cfg_id)
```

Inserts a record into the datalog storage. The record is identified by its timestamp, and consists of a list of numerical values, and a configuration identifier number.

Parameters:

- **time:** *float*,or 64-bit integer, the UNIX timestamp, holding the exact time of when the measurement has been made; when stored, only integer part with seconds resolution will be stored
- **values:** *list,* of numerical values, either *float* or 32-bit *integer* type, holding measurements to be stored into the datalog; maximum of 255 items are allowed in the list.
- **cfg_id:** 64-bit *integer*, value denotes the index of the active system configuration at the moment when the measurements were made; only lower 16 bits are stored into the datalog

Return value: *none*

Raises:

- *TypeError* if any argument is of incorrect type
- *ValueError* if there are too many items in the **values** list

```
datalog.set_status(record_time, flags, status_time)
```

Updates the status for the existing record in the datalog storage. The existing record is identified by its timestamp. The status consists of a 8-bit flag field, and the timestamp of the status update.

Parameters:

- **record_time:** *float* or 64-bit *integer*, the UNIX timestamp which identifies the existing record in the datalog storage; this is the timestamp that was provided when calling `datalog.insert()` function.
- **flags:** 32-bit *integer*, contains flags that will be stored into the datalog
- **status_time:** *float* or 64-bit *integer*, the UNIX timestamp which should denote the time when the status has been updated

Return value: *none*

Raises:

- *TypeError* if any argument is of incorrect type

```
datalog.read(start, count [ , unconfirmed_only = False [ , read_status_flags = False ] ] )
```

Read up to **count** entries from the datalogger storage, starting at specified **start** time.

Parameters

- **start:** *float* or 64-bit *integer*, the UNIX timestamp at which the datalogger reading will start; the first returned entry will have timestamp at this time, or later time
- **count:** *integer,* the maximum number of entries to read
  unconfirmed_only: *boolean,* if `True`, the function will only return the items that do not have any
- status flags set
  read_status_flags: *boolean,* if `True` and **unconfirmed_only** is `False`, each returned entry will also contain additional fields for the status **flags** and the **status_time** as timestamp

Return value

- *list* of entries, each entry is a dictionary consisting of the following fields:
  - **time:** 64-bit integer, the UNIX timestamp for the entry
  - **values:** *list*, containing numerical measurement values
  - **cfg_id:** 64-bit integer, 64-bit identifier of the active configuration
  - **flags:** Optional *integer*
  - **status_time:** Optional 64-bit integer, the UNIX timestamp for flags

Raises:

- *TypeError* if any argument is of incorrect type

```
datalog.get_range()
```

Return the time range of the earliest and latest entry in the datalog.

Parameters: *none*

Return value

- *tuple* consisting of two UNIX timestamp values
  - first element in the returned tuple is a timestamp of the earliest entry, and
  - second element is the timestamp of the latest entry.

Raises: *none*

```
datalog.clear()
```

Clear all entries from datalog.

Parameters: *none*

Return value: *none*

Raises: *none*

Example 1

Get data values from datalog, include unconfirmed entries. Unconfirmed entries are those that do not have any status flag set, meaning - not sent to server.

```
import datalog

# Attempt to retrieve the time range (earliest and latest timestamps)
# of the stored datalog entries.
try:
    range_datalog = datalog.get_range()
    print(“Datalog Range:”, range_datalog)
except Exception as e:
    print(“Error in datalog.get_range():”, e)
    exit()  # Exit if unable to retrieve datalog range.

# Check if the datalog is empty; if so, inform the user and exit.
if len(range_datalog) < 1:
    print(“No data in datalog”)
    exit()

# Extract the earliest entry’s UNIX timestamp from the retrieved range.
earliest_entry_unixtimestamp = range_datalog[0]

# Attempt to read up to 2 entries from the datalog,
# starting from the earliest entry; the parameters ensure only
# unconfirmed entries are read (read_status_flags is ignored when unconfirmed_only is True).
try:
    data_unconf = datalog.read(earliest_entry_unixtimestamp, 2, True, True)
except Exception as e:
    print(“Error in datalog.read():”, e)
    exit()  # Exit if reading from the datalog fails.

# Calculate the number of unconfirmed data entries retrieved.
unconfirmed_data_len = len(data_unconf)

# Inform the user if there are no unconfirmed entries.
if unconfirmed_data_len <= 0:
    print(“There is no unconfirmed data”)
else:
    print(“data_unconf_len:”, unconfirmed_data_len)
    for item in data_unconf:
        print(“Data_unconf:”, item)

# do something with unconfirmed data, for example send it to server
```

Example 2

1. Insert a new datalog record
2. Update the status of the previously inserted datalog record
3. Read up to 10 datalog entries
4. Retrieve the time range
5. Clear all datalog entries

```
import can
import logger

# Insert a new datalog record using the current UNIX timestamp from logger.get_timestamp()
try:
    current_timestamp = logger.get_timestamp()
    values = [25.5, 50.0, 1013]  # e.g., temperature, humidity, pressure
    config_id = 12345
    datalog.insert(current_timestamp, values, config_id)
    print(“Inserted datalog record at timestamp”, current_timestamp)
except Exception as e:
    print(“Error in datalog.insert:”, e)
```

```
# Update the status of the previously inserted datalog record
try:
    # For example, set a status flag and use offseted timestamp as the status update time.
    status_flags = 3
    # Optionally, you could get the current timestamp again; Here we add an offset;
    status_time = current_timestamp + 5
    datalog.set_status(current_timestamp, status_flags, status_time)
    print(“Updated status for record at timestamp”, current_timestamp)
except Exception as e:
    print(“Error in datalog.set_status:”, e)

# Read up to 10 datalog entries starting from current_timestamp
try:
    records = datalog.read(current_timestamp, 10, False, True)
    print(“Datalog records read:”)
    for rec in records:
        print(rec)
except Exception as e:
    print(“Error in datalog.read:”, e)

# Retrieve the time range (earliest and latest timestamps) of the stored datalog entries
try:
    time_range = datalog.get_range()
    print(“Datalog time range: earliest =”, time_range[0], “, latest =”, time_range[1])
except Exception as e:
    print(“Error in datalog.get_range:”, e)

# Clear all datalog entries
try:
    datalog.clear()
    print(“Datalog cleared.”)
except Exception as e:
    print(“Error in datalog.clear:”, e)
```

3.4. Digital I/O

Configures digital input counters for GPIO channels and provides functions to read and reset input counters.

Functions

- **`set_cnt_type(GPIO_channel, counter_type)`**
  Configures the counter type on a GPIO channel.
- **`get_cnt(GPIO_channel)`**
  Gets current counter value for the channel.
- **`reset_cnt(GPIO_channel)`**
  Resets the channel’s counter to zero.

`digital_io.set_cnt_type(GPIO_channel, counter_type)`

Set counter type on specific GPIO channel, which can be either *pulse* counter or frequency *counter*.

Parameters:

- **GPIO_channel:** *integer*, GPIO channel. Argument value must be 0, 1, 2 or 3.
- **counter_type:** *integer*, Specifies counter type. Argument value must be 0 (pulse counter) or 1 (frequency counter).

Return value:

- *boolean*, `True` on success

Raises:

- *TypeError* if any argument is of incorrect type
- *ValueError* if any argument's value is out-of-range

`digital_io.get_cnt(GPIO_channel)`

Getter for counter variables.

Parameters:

- **GPIO_channel:** *integer*, GPIO channel. Argument value must be 0, 1, 2 or 3.

Return value:

- *integer,* counter value

Raises:

- *TypeError* if **GPIO_channel** is of incorrect type
- *ValueError* if **GPIO_channel** is out-of-range

`digital_io.reset_cnt(GPIO_channel)`

Reset counter variables to zero for specific GPIO channel.

Parameters:

- **GPIO_channel:** *integer*, GPIO channel. Argument value must be 0, 1, 2 or 3.

Return value:

- *boolean*, `True` on success

Raises:

- *TypeError* if **GPIO_channel** is of incorrect type
- *ValueError* if **GPIO_channel** is out-of-range

Example

1. Set Counter Type: Configures the GPIO channel with a specific counter type.
2. Read Counter Value: Retrieves the current counter value.
3. Reset Counter: Resets the counter to zero.
4. Verify Reset: Reads the counter again to confirm reset.

```
import digital_io
import logger  # Import logger for delay functionality

# Define the GPIO channel to use
GPIO_CHANNEL = 1  # Can be 0, 1, 2, or 3

# Step 1: Set the counter type for the GPIO channel
try:
    counter_type = 0  # Example: Set to type 0 (e.g., pulse counting mode)
    if digital_io.set_cnt_type(GPIO_CHANNEL, counter_type):
        print("Counter type set successfully for GPIO channel", GPIO_CHANNEL, ".")
    else:
        print("Failed to set counter type for GPIO channel", GPIO_CHANNEL, ".")
except Exception as e:
    print("Error setting counter type:", e)

# Wait 10 seconds before reading the counter value,
# allowing the user time to toggle GPIO inputs.
try:
    print("Waiting 10 seconds; please toggle GPIO inputs if desired...")
    logger.delay_ms(10000)  # 10 seconds delay (10,000 milliseconds)
except Exception as e:
    print("Error during delay:", e)

# Step 2: Read the current counter value
try:
    counter_value = digital_io.get_cnt(GPIO_CHANNEL)
    print("Current counter value for GPIO channel", GPIO_CHANNEL, ":", counter_value)
except Exception as e:
    print("Error reading counter value:", e)

# Step 3: Reset the counter
try:
    if digital_io.reset_cnt(GPIO_CHANNEL):
        print("Counter reset successfully for GPIO channel", GPIO_CHANNEL, ".")
    else:
        print("Failed to reset counter for GPIO channel", GPIO_CHANNEL, ".")
except Exception as e:
    print("Error resetting counter:", e)

# Step 4: Verify counter reset by reading the value again
try:
    counter_value_after_reset = digital_io.get_cnt(GPIO_CHANNEL)
     print("Counter value after reset for GPIO channel", GPIO_CHANNEL, ":", counter_value_after_
reset)
except Exception as e:
    print("Error verifying counter reset:", e)
```

3.5. FTP client

Provides FTP operations for remote file management, including listing directories, transferring files, and deleting files or directories.

Functions

- `help(netif, url, username, password)`
  Sends the FTP HELP command to the server and retrieves available commands
- `list(netif, url, username, password, uri)`
  Sends the FTP LIST command to list the contents of a directory (or file info) on the server
- `syst(netif, url, username, password)`
  Sends the FTP SYST command to retrieve the server OS type
- `put(netif, url, username, password, uri, file, file_size, offset)`
  Sends the FTP STOR command to upload (store) a file (or file chunk) on the server
- `get(netif, url, username, password, uri)`
  Sends the FTP RETR command to download a file (or file data) from the server
- `delete_file(netif, url, username, password, uri)`
  Sends the FTP DELE command to delete a file on the server
- `delete_path(netif, url, username, password, uri)`
  Sends the FTP RMD command to delete a directory on the server

```
ftp_client.help(netif, url, username, password)
```

Retrieves help information from the FTP server, list the commands available on the server side using FTP HELP command.

Parameters:

- **netif:** *integer,* the ID of the Network Interface through which FTP connection shall be made. Argument value must be one of constants `logger.NETIF_ETHERNET`, `logger.NETIF_MODEM_A`, `logger.NETIF_MODEM_B`
- **url:** *string,* The FTP server URL, must start with “ftp://”, mandatory
- **username:** *string,* username for the connection, can be left empty
- **password:** *string,* password for the connection, can be left empty

Return value:

- *dictionary* containing following entries:
  - **ftp_c_status:** *integer,* FTP client result status
  - **ftp_s_errno:** *integer,* FTP server errorno code
  - **ftp_s_code:** integer, FTP server return code / reply code
  - **ftp_c_bytes:** *integer,* indicating how many bytes were transferred
  - **ftp_s_string:** Optional *bytearray*, FTP server control socket response

Raises:

- *TypeError* if any argument is of incorrect type OR internal error
- *ValueError* if **netif** is unsupported / out-of-range

`ftp_client.list(netif, url, username, password, uri)`

Lists directory contents from the FTP server, using FTP LIST command. The **uri** parameter may specify a particular directory on the server.

Parameters:

- **netif:** *integer,* the ID of the Network Interface through which FTP connection shall be made. Argument value must be one of constants `logger.NETIF_ETHERNET, logger.NETIF_MODEM_A, logger.NETIF_MODEM_B`
- **url:** *string,* The FTP server URL, must start with “ftp://”, mandatory
- **username:** *string,* username for the connection, can be left empty
- **password:** *string,* password for the connection, can be left empty
- **uri:** *string,* The directory path on the FTP server to list, must start with “/”

Return value:

- *dictionary* containing following entries:
    - **ftp_c_status:** *integer,* FTP client result status
    - **ftp_s_errno:** *integer,* FTP server errorno code
    - **ftp_s_code:** integer, FTP server return code / reply code
    - **ftp_c_bytes:** *integer,* number of bytes processed
    - **ftp_s_string:** Optional *bytearray*, FTP server control socket response
    - **ftp_s_data:** Optional *bytearray*, server data response, directory listing (if any)

Raises:

- *TypeError* if any argument is of incorrect type OR internal error
- *ValueError* if **netif** is unsupported / out-of-range / internal error

`ftp_client.syst(netif, url, username, password)`

Retrieves the system type from the FTP server, using FTP SYST command.

Parameters:

- **netif:** *integer,* the ID of the Network Interface through which FTP connection shall be made. Argument value must be one of constants `logger.NETIF_ETHERNET, logger.NETIF_MODEM_A, logger.NETIF_MODEM_B`
- **url:** *string,* The FTP server URL, must start with “ftp://”, mandatory
- **username:** *string,* username for the connection, can be left empty
- **password:** *string,* password for the connection, can be left empty

Return value:

- *dictionary* containing following entries:
    - **ftp_c_status:** *integer,* FTP client result status
    - **ftp_s_errno:** *integer,* FTP server errorno code
    - **ftp_s_code:** integer, FTP server return code / reply code
    - **ftp_c_bytes:** *integer,* number of bytes processed
    - **ftp_s_string:** Optional *bytearray*, FTP server control socket response

Raises:

- *TypeError* if any argument is of incorrect type OR internal error
- *ValueError* if **netif** is unsupported / out-of-range

`ftp_client.put(netif, url, username, password, uri, file, file_size, offset)`

Stores a file on the FTP server, using FTP STOR command. This function issues the FTP STOR command. It sends a file to the FTP server, starting at the specified offset.

Parameters:

- **netif:** *integer,* the ID of the Network Interface through which FTP connection shall be made. Argument value must be one of constants `logger.NETIF_ETHERNET, logger.NETIF_MODEM_A, logger.NETIF_MODEM_B`
- **url:** *string,* The FTP server URL, must start with "ftp://", mandatory
- **username:** *string,* username for the connection, can be left empty
- **password:** *string,* password for the connection, can be left empty
- **uri:** *string,* The destination path on the FTP server where the file will be stored, must start with "/"
- **file_size:** *integer,* The length of the file data in bytes, must be positive value
- **offset:** *integer,* The offset (in bytes) at which to start writing the file, must be positive value

Return value:

- *dictionary* containing following entries:
  - **ftp_c_status:** *integer,* FTP client result status
  - **ftp_s_errno:** *integer,* FTP server errorno code
  - **ftp_s_code:** integer, FTP server return code / reply code
  - **ftp_c_bytes:** *integer,* number of bytes processed
  - **ftp_s_string:** Optional *bytearray*, FTP server control socket response

Raises:

- *TypeError* if any argument is of incorrect type OR internal error
- *ValueError* if **netif** is unsupported / out-of-range

`ftp_client.get(netif, url, username, password, uri)`

Retrieves a file from the FTP server, download data from the server (after PASV (passive mode) or PORT (active mode) commands), using FTP RETR command.

Parameters:

- **netif:** *integer,* the ID of the Network Interface through which FTP connection shall be made. Argument value must be one of constants `logger.NETIF_ETHERNET, logger.NETIF_MODEM_A, logger.NETIF_MODEM_B`
- **url:** *string,* The FTP server URL, must start with "ftp://", mandatory
- **username:** *string,* username for the connection, can be left empty
- **password:** *string,* password for the connection, can be left empty
- **uri:** *string,* The destination path on the FTP server where the file will be stored, must start with "/"

Return value:

- *dictionary* containing following entries:
  - **ftp_c_status:** *integer,* FTP client result status
  - **ftp_s_errno:** *integer,* FTP server errorno code
  - **ftp_s_code:** integer, FTP server return code / reply code
  - **ftp_c_bytes:** *integer,* number of bytes processed
  - **ftp_s_string:** Optional *bytearray*, FTP server control socket response
  - **ftp_s_data:** Optional *bytearray*, FTP server data response containing the file contents.

Raises:

- *TypeError* if any argument is of incorrect type OR internal error
- *ValueError* if **netif** is unsupported / out-of-range

`ftp_client.delete_file(netif, url, username, password, uri)`

Deletes a file from the FTP server, using FTP DELE command.

Parameters:

- **netif:** *integer*, the ID of the Network Interface through which FTP connection shall be made. Argument value must be one of constants `logger.NETIF_ETHERNET, logger.NETIF_MODEM_A, logger.NETIF_MODEM_B`
- **url:** *string*, The FTP server URL, must start with "ftp://", mandatory
- **username:** *string*, username for the connection, can be left empty
- **password:** *string*, password for the connection, can be left empty
- **uri:** *string*, The path to the file to be deleted on the FTP server, must start with "/"

Return value:

- *dictionary* containing following entries:
  - **ftp_c_status:** *integer*, FTP client result status
  - **ftp_s_errno:** *integer*, FTP server errorno code
  - **ftp_s_code:** integer, FTP server return code / reply code
  - **ftp_c_bytes:** *integer*, number of bytes processed
  - **ftp_s_string:** Optional *bytearray*, FTP server control socket response

Raises:

- *TypeError* if any argument is of incorrect type OR internal error
- *ValueError* if **netif** is unsupported / out-of-range

`ftp_client.delete_path(netif, url, username, password, uri)`

Deletes a directory (or path) from the FTP server. This function issues the FTP RMD command to remove a directory from the FTP server.

Parameters:

- **netif:** *integer*, the ID of the Network Interface through which FTP connection shall be made. Argument value must be one of constants `logger.NETIF_ETHERNET, logger.NETIF_MODEM_A, logger.NETIF_MODEM_B`
- **url:** *string*, The FTP server URL, must start with "ftp://", mandatory
- **username:** *string*, username for the connection, can be left empty
- **password:** *string*, password for the connection, can be left empty
- **uri:** *string*, The directory path on the FTP server to remove, must start with "/"

Return value:

- *dictionary*, containing following entries:
  - **ftp_c_status:** *integer*, FTP client result status
  - **ftp_s_errno:** *integer*, FTP server errorno code
  - **ftp_s_code:** integer, FTP server return code / reply code
  - **ftp_c_bytes:** *integer*, number of bytes processed
  - **ftp_s_string:** Optional *bytearray*, FTP server control socket response

Raises:

- *TypeError* if any argument is of incorrect type OR internal error
- *ValueError* if **netif** is unsupported / out-of-range

Example

1. Retrieve FTP HELP – Understand available commands .
2. Retrieve System Information – Understand system details.
3. List Remote Directory – Verify existing files before modifying anything.
4. Upload a File – Send a file to the FTP server.
5. Download the File – Ensure that the uploaded file can be retrieved.
6. Delete the File – Clean up after verifying the file transfer.
7. Delete Directory – Remove the directory if it’s empty.

```
import ftp_client
import logger

# Define FTP connection parameters
NETIF = logger.NETIF_ETHERNET  # Network interface (can be MODEM_A or MODEM_B)
FTP_URL = “ftp://example.com”  # FTP server URL
USERNAME = “user”
PASSWORD = “pass”
REMOTE_DIR = “/remote_folder”  # Directory on the FTP server
REMOTE_FILE = “readme.txt”  # Remote file path
REMOTE_FILE_2 = “/readme_2.txt”  # Remote file path for chunked transfer
LOCAL_FILE_CONTENT = b”Sample data for FTP transfer”  # Sample file content
LOCAL_FILE_CONTENT_2 = (
     b”A9f$1&d@Zq7#P!3W8e%2^R$V0L)M4@!T#7_u8(J%2^K+6?N3=Z$1@(C&9L!5)
     R7!e^C$2b(8D#q9?s%5H@1KM0)w_3+Zx$”b”F&4L)P?2@v#6(n%0!Y3^d+8=R)5]
     L@3#q8&Wr2D!5^MZ7)f$1+Te_9@H%0Bs(4&J!3PV6R^2#Qy8$L@0)NU?5+K]7(W4!n$”b”0@
     Qe7^R2+PD?6#M)1(Z%3!L$8&k9V@0^Sc)5+F#2!B8?d_7RG(1@e%4+O+2^s!5#n0)T$7&8q%3@L1(R4?d_9)
     Ew(6+F!2%”b”Z8B@0^Cv#1$J)7Xm@5&Q+3)Y$4@z!8l(3^N#5?S7)r%2+Kf!0&W$6^u)9Q@1&H#3+T%8(d^5!C$2)
     e+7V@0#r1!S$3Lo(8^W”b”#5?E7)f%2+Mj@0&R$6^q)9V@1&U#3+P%8(k^5!N$2)
     i+7B@0#P$2@t!6&z(4^J#8?W7)l%3+Fc@0&R$5^v)9D@1&Q#3+K%7(“b”b^5!M$2)
     n+8Z@0#M8!p$3&Gu(7^C#5?R6)j%2+Nh@0&R$4^k)9F@1&T#3+W%8(y^5!L$2)
     e+7D@0#X#2@v!7&b(5^L#8?S7)”b”h%3+Jd@0&R$6^f)9Q@2&K#4+T%8(m^5!N$3)o+8R@1#Z@1#x!5&v(6^C#2?P7)
     d%4+Fg@0&R$9^h)8J@3&L#5+T%1(q^7!V$”b”2)w+4B@0#D#3@u!7&s(8^M#4?W7)k%2+Pl@0&R$5^o)9F@
     1&N#3+Q%8(i^5!L$2)a+7D@0#F%4@b!9k(2^P#7?S8)j%3+Ld@”b”0&R$6^c)9H@1&T#3+V%8(m^5!Q$2)
     z+7X@0#J@2#n!6&v(5^D#8?R7)l%2+Ks@0&R$4^w)9G@1&P#3+T%8(y^5!M$2)o+7F@”b”0#U#4@c!7&z(9^H#2?M6)
     k%1+Je@0&R$3^f)8D@2&L#5+S%7(x^3!Q$1)i+4W@0#O$3@r!8&p(7^J#4?T8)g%2+Fm@0&R$5^”b”n)9E@1&R#3+U%6
     (q^5!V$2)b+7Z@0#”
)   # Sample file content

# Step 1: Retrieve available commands from the FTP server
try:
    help_response = ftp_client.help(NETIF, FTP_URL, USERNAME, PASSWORD)
    print(“FTP HELP Response:”, help_response)
except Exception as e:
    print(“Error retrieving FTP HELP information:”, e)

# Step 2: Retrieve system type from the FTP server
try:
    syst_response = ftp_client.syst(NETIF, FTP_URL, USERNAME, PASSWORD)
    print(“FTP System Type Response:”, syst_response)
except Exception as e:
    print(“Error retrieving FTP system information:”, e)

# Step 3: List contents of the remote directory
try:
    list_response = ftp_client.list(NETIF, FTP_URL, USERNAME, PASSWORD, REMOTE_DIR)
    print(“FTP Directory Listing:”, list_response)
except Exception as e:
    print(“Error listing FTP directory:”, e)

logger.delay_ms(5000)
```

```
# Step 4: Upload a file to the server
try:
    file_size = len(LOCAL_FILE_CONTENT)
    upload_response = ftp_client.put(NETIF,
                                     FTP_URL,
                                     USERNAME,
                                     PASSWORD,
                                     REMOTE_FILE,
                                     LOCAL_FILE_CONTENT,
                                     file_size,
                                     0)
    print("File uploaded response:", upload_response)
except Exception as e:
    print("Error uploading file to FTP server:", e)

logger.delay_ms(5000)

# Step 5: Upload a file to the server in chunks (useful when transferring a large file over
# cellular network under poor signal conditions)
# Here using chunks of size 128 bytes
offset = 0
file_size = len(LOCAL_FILE_CONTENT_2)
while offset != file_size:
    packet_size = 0
    if (file_size - offset) > 128:
        packet_size = 128
    else:
        packet_size = file_size - offset
    chunk_upload_response = ftp_client.put(NETIF,
                                           FTP_URL,
                                           USERNAME,
                                           PASSWORD,
                                           REMOTE_FILE_2,
                                           LOCAL_FILE_CONTENT[offset:],
                                           packet_size,
                                           offset)
    print("Chunk upload Response", chunk_upload_response)
    offset += chunk_upload_response.get("ftp_c_bytes")
    if (chunk_upload_response.get("ftp_s_string") is not None    \
        and len(chunk_upload_response.get("ftp_s_string")) == 0) \
        or (chunk_upload_response.get("ftp_s_code") is not None  \
        and chunk_upload_response.get("ftp_s_code") >= 400):
        break;

# Step 6: Download the uploaded file from the server
try:
    download_response = ftp_client.get(NETIF, FTP_URL, USERNAME, PASSWORD, REMOTE_FILE)
    print("Downloaded File Content:", download_response.get("ftp_s_data", b"No Data").decode())
except Exception as e:
    print("Error downloading file from FTP server:", e)

# Step 7: Delete the uploaded files from the server
try:
    delete_response = ftp_client.delete_file(NETIF, FTP_URL, USERNAME, PASSWORD,  REMOTE_FILE)
    print("File deletion response:", delete_response)
    delete_response = ftp_client.delete_file(NETIF, FTP_URL, USERNAME, PASSWORD,  REMOTE_FILE_2)
    print("File deletion response:", delete_response)
except Exception as e:
    print("Error deleting file from FTP server:", e)

# Step 8: Delete the remote directory (if empty)
try:
    delete_dir_response = ftp_client.delete_path(NETIF, FTP_URL, USERNAME, PASSWORD, REMOTE_DIR)
    print("Directory deletion response:", delete_dir_response)
except Exception as e:
    print("Error deleting directory from FTP server:", e)
```

3.6. HTTP client

Supports basic HTTP GET and POST requests with configurable timeouts and returns response details.

Functions

- `send(netif, url, headers, body)`
  Performs an HTTP GET or POST request.
- `set_timeouts(total_request_timeout_sec, max_silence_timeout_sec)`
  Configures HTTP connection timeouts.

`http.send(netif, url, headers, body)`

Sends HTTP request. If BODY is empty or omitted then makes HTTP GET request otherwise makes HTTP POST request.

Parameters:

- **netif:** *integer,* the network interface ID of the Network Interface through which POST request shall be sent. Argument value must be one of constants `logger.NETIF_ETHERNET, logger.NETIF_MODEM_A, logger.NETIF_MODEM_B`
- **url:** *string,* URL of the request, mandatory
- **headers:** Optional *dictionary* OR *list* of custom headers, can be *None* if no custom headers are used, or just omitted
- **body:** Optional *bytes* or string of a body; if exists, the request will be POST request

Return value:

- *None* OR
  critical error, OR
  dictionary containing elements:
  - **status:** the status result
  - **headers:** *bytes,* received headers
  - **body:** *bytes,* received body
  - **code:** *integer,* HTTP result code, one of the `HTTP_STATUS_*` constants)

Raises:

- *TypeError* if any argument is of incorrect type
- *ValueError* if there is a problem with arguments

`http.set_timeouts(total_request_timeout_sec, max_silence_timeout_sec)`

Changes timeout constants for the HTTP connection

Parameters:

- **total_request_timeout_sec:** *integer,* the timeout will be raised if the whole HTTP request does not complete within defined amount of time, given in seconds, must be between 1 and 86400
- **max_silence_timeout_sec:** *string,* the timeout will be raised if no bytes have been received from server for the specified amount of time, given in seconds, must be between 1 and 86400

Return value: *None*

Raises:

- *TypeError* if any argument is of incorrect type
- *ValueError* if any argument's value is out-of-range

Example

1. Set HTTP Timeouts – Ensures the request doesn't hang indefinitely.
2. Perform GET Request – Fetch data from the server.
3. Perform POST Request – Send data to the server.

```
import http
import logger

# Define network interface (Choose one: NETIF_ETHERNET, NETIF_MODEM_A, NETIF_MODEM_B)
NETIF = logger.NETIF_ETHERNET

# Define URLs
GET_URL = "http://example.com/data?submit_json="  # URL for GET request
POST_URL = "http://example.com/submit"      # URL for POST request

# Define headers
HEADERS = {"Content-Type": "application/json", "Accept": "application/json"}

# Define POST request body
BODY = '{"temperature": 25.4, "humidity": 60}'

# Step 1: Configure HTTP timeouts
try:
    http.set_timeouts(120, 30)  # Set total request timeout to 120s, silence timeout to 30s
    print("HTTP timeouts configured successfully.")
except Exception as e:
    print("Error setting HTTP timeouts:", e)

# Step 2: Perform an HTTP GET request
try:
    get_response = http.send(NETIF, GET_URL + BODY, HEADERS)  # Perform GET request
    print("\nGET Request Response:")
    print("  Status:", get_response.get("status"))
    print("  HTTP Code:", get_response.get("code"))
    print("  Headers:", get_response.get("headers"))
    print("  Body:", get_response.get("body").decode())
except Exception as e:
    print("Error during HTTP GET request:", e)

# Step 3: Perform an HTTP POST request
try:
    post_response = http.send(NETIF, POST_URL, HEADERS, BODY)  # Perform POST request
    print("\nPOST Request Response:")
    print("  Status:", post_response.get("status"))
    print("  HTTP Code:", post_response.get("code"))
    print("  Headers:", post_response.get("headers"))
    print("  Body:", post_response.get("body").decode())
except Exception as e:
    print("Error during HTTP POST request:", e)
```

3.7. Internal storage

Manages internal device configurations, Python scripts, instrument settings, and firmware updates in persistent storage.

Constants

```
internal_storage.NONE = 0
internal_storage.GPRS = 1
internal_storage.GPRS_LTE = 2
internal_storage.LTE = 3
```

Functions

- `read_logger_settings()`
  Retrieves basic logger settings as a dictionary.
- `write_logger_settings(config_data, config_id)`
  Updates logger settings with new configuration.
- `read_instrument_config( [ config_id=None ] )`
  Returns instrument settings for the given config.
- `write_instrument_config(config_data, config_id)`
  Writes raw instrument configuration data.
- `write_python_script(script_data [ , cfg_id=None ] )`
  Saves Python script to internal storage.
- `read_python_script()`
  Reads the currently active Python script.
- `write_firmware(firmware_bin)`
  Stores new firmware for bootloader upgrade.
- `get_newest_instrument_config()`
  Returns the newest instrument config ID.
- `get_oldest_instrument_config()`
  Returns the oldest instrument config ID.
- `read_virtual_instrument_config()`
  Retrieves virtual instrument configuration.
- `read_data_upload_config()`
  Returns data upload configuration.

```
internal_storage.read_logger_settings()
```

Returns dictionary containing the basic logger settings

Parameters: *None*

Return value:

- *dictionary* containing:
  - **unique_device_id:** *string*, holding the unique device identifier
  - **config_id:** 64-bit *integer*, internal configuration ID of the current logger settings; this includes encoded timestamp
  - **slot_a:** 32-bit *integer*, type of the device in slot A, one of the defined constants
  - **slot_b:** 32-bit *integer*, type of the device in slot B

- **eth_use_dhcp:** *boolean*
- **eth_ip_address:** *string*, IP address if DHCP is not used
- **eth_subnet_mask:** *string*, the subnet mask if DHCP is not used
- **eth_gateway_address:** *string*, the gateway address if DHCP is not used
- **eth_dns:** *list* of strings of gateway addresses
- **charge_current:** *float*, configured battery charging current, in Amperes
- **max_charge_voltage:** *float*, maximum charging voltage, in Volts
- **max_DPM_current:** *float*, maximum input DPM current, in Amperes
- **input_DPM_voltage:** *float*, input DPM voltage, in Volts
- **min_sys_voltage:** *float*, minimum system voltage, in Volts
- **const_standby_voltage:** *float*, constant standby voltage for battery, in Volts
- **min_charge_current:** *float*, control current when the battery is full
- **min_charge_temperature:** *integer*, minimum charging temperature in °C; if the temperature is below this value, the battery will not be charged
- **enable_charging:** *boolean*, indicates whether the battery charging is enabled or not
- **scan_interval_min:** *integer*, the current scanning interval, in minutes
- **slot_a_config:** *dictionary*, containing slot_a settings;
    - **gprs_apn:** *string*
    - **gprs_username:** *string*,
    - **gprs_password:** *string*,
    - **gprs_dns:** *string*,
    - **eth_dns:** *list* of strings of gateway addresses
- **slot_b_config:** *dictionary*, containing slot_b settings;
    - **gprs_apn:** *string*
    - **gprs_username:** *string*,
    - **gprs_password:** *string*,
    - **gprs_dns:** *string*,
- **save_to_usb:** *boolean*,
- **save_to_sd:** *boolean*,

Raises: *None*

`internal_storage.write_logger_settings(config_data, config_id)`

Updates logger settings with the new configuration.

Parameters:

- **config_data:** *string* or *bytes*, the raw CBOR data object to be saved
- **config_id:** : 64-bit *integer*, the configuration ID for the new config

Return value:

- *boolean*, `True` if saving is successful; `False` otherwise.

Raises:

- *TypeError* if any argument is of incorrect type
- *ValueError* if **config_id** value is zero

`internal_storage.read_instrument_config( [ config_id ] )`

Returns dictionary containing all instrument settings.

Parameters:

- **config_id:** Optional 64-bit *integer,* holding the desired ID. If **config_id** is zero or not provided, the latest settings are returned.

Return value:

- *None* if instrument settings with **config_id** does not exist OR
- *dictionary* containing the instrument settings

Raises:

- *TypeError* if **config_id** is of incorrect type
- *ValueError* if internal error

`internal_storage.write_instrument_config(config_data, config_id)`

Writes raw configuration data to the storage.

Parameters:

- **config_data:** *string* or *bytes*, the raw data object to be saved
- **config_id:** 64-bit *integer*, configuration ID as provided (must be non-zero positive)

Return value:

- *boolean*, `True` if saving is successful; `False` otherwise.

Raises:

- *TypeError* if any argument is of incorrect type
- *ValueError* if **config_id** value is zero

`internal_storage.write_python_script(script_data [ , cfg_id ] )`

Writes Python script to the internal storage.

Parameters:

- **config_data:** *string* or *bytes*, the raw data containing Python code to be executed by core engine
- **config_id:** Optional 64-bit *integer*, configuration ID for the script; if missing a new one will be automatically created

Return value:

- *boolean*, `True` if saving is successful; `False` otherwise.

Raises:

- *TypeError* if any argument is of incorrect type

`internal_storage.read_python_script()`

Reads and returns active Python script.

Parameters: *None*

Return value:

- *None* if the script with **config_id** does not exist OR
- *string* with contents of Python script.

Raises: *None*

`internal_storage.write_firmware(firmware_bin)`

Writes new Firmware binary file to internal storage. After rebooting, the bootloader will use this file to upgrade the firmware.

Parameters:

- **firmware_bin:** *string* or *bytes*, the raw data containing firmware binary

Return value:

- *boolean*, `True` if saving is successful; `False` otherwise.

Raises:

- *TypeError* if **firmware_bin** is of incorrect type

`internal_storage.get_newest_instrument_config()`

Returns the config ID of the newest instrument configuration file, or None if no config file exists.

Parameters: *None*

Return value:

- *None* if instrument configuration with **config_id** does not exist OR
- 64-bit *integer,* config ID of the newest instrument configuration file

Raises: *None*

`internal_storage.get_oldest_instrument_config()`

Returns the config ID of the oldest instrument configuration file, or None if no config file exists.

Parameters: *None*

Return value:

- *None* if instrument configuration with **config_id** does not exist OR
- 64-bit *integer,* config ID of the oldest instrument configuration file

Raises: *None*

`internal_storage.read_virtual_instrument_config()`

Returns dictionary containing all virtual instrument settings.

Parameters: *None*

Return value:

- *dictionary* containing the virtual instrument settings.

Raises:

- *ValueError* if internal error

`internal_storage.read_data_upload_config()`

Returns dictionary containing all data upload settings.

Parameters: *None*

Return value:

- *None* if data upload settings do not exist OR
- *dictionary* containing the data upload settings:

Raises:

- *ValueError* if internal error

Example

1. Reads current settings then Writes modified settings back.
2. Reads existing configuration then Writes a modified version back
3. Reads stored scripts then Writes new script
4. Simulates writing a firmware binary.
5. Gets the newest and oldest instrument configurations.
6. Reads Virtual Instrument and Data Upload Configuration

```
import internal_storage
import logger

# Step 1: Read and display current logger settings
try:
    print("\nReading current logger settings...")
    logger_settings = internal_storage.read_logger_settings()
    print("Logger Settings:", logger_settings)
except Exception as e:
    print("Error reading logger settings:", e)

# Step 2: Modify and update logger settings
try:
    print("\nUpdating logger settings...")
    new_logger_settings = logger_settings.copy()
    new_logger_settings["enable_charging"] = False  # Example modification
    config_id = logger_settings.get("config_id")  # Retain the existing config ID
    # You must write Your own function dict_to_cbor() that converts dictionary to CBOR
    cbor_new = dict_to_cbor(new_logger_settings)
    success = internal_storage.write_logger_settings(cbor_new, config_id)
    if success:
        print("Logger settings updated successfully.")
    else:
        print("Failed to update logger settings.")
except Exception as e:
    print("Error writing logger settings:", e)

# Step 3: Read latest instrument configuration
try:
    print("\nReading latest instrument configuration...")
    instrument_config = internal_storage.read_instrument_config()
    if instrument_config:
        print("Instrument Configuration:", instrument_config)
    else:
        print("No instrument configuration found.")
except Exception as e:
    print("Error reading instrument configuration:", e)

# Step 4: Write a new instrument configuration
try:
    print("\nWriting new instrument configuration...")
    if instrument_config:
        new_instrument_config = instrument_config.copy()
        new_instrument_config["sample_rate"] = 5  # Example modification
        config_id = instrument_config["config_id"]
        success = internal_storage.write_instrument_config(new_instrument_config, config_id)
        if success:
            print("Instrument configuration updated successfully.")
        else:
            print("Failed to update instrument configuration.")
except Exception as e:
    print("Error writing instrument configuration:", e)
```

```
# Step 5: Read a stored Python script
try:
    print("\nReading stored Python script...")
    python_script = internal_storage.read_python_script()
    if python_script:
        print("Python Script Contents:\n", python_script)
    else:
        print("No Python script found.")
except Exception as e:
    print("Error reading Python script:", e)

# Step 6: Write a new Python script
# NOTE: Writing a new Python script will overwrite the exiting/this script
try:
    print("\nWriting a new Python script...")
    new_script = "print('Hello from internal storage!')"
    success = internal_storage.write_python_script(new_script)
    if success:
        print("Python script written successfully.")
    else:
        print("Failed to write Python script.")
except Exception as e:
    print("Error writing Python script:", e)

# Step 7: Update firmware,
# NOTE: USE WITH CARE, USE ONLY ORIGINAL FIRMWARE PROVIDED BY GEOLUX,
#       FOLLOW INSTRUCTIONS PROVIDED BY GEOLUX
try:
    print("\nWriting new firmware (simulation)...")
    # Example firmware data (binary), not actual
    # Usually, firmware binary is retrieved from
    # e.g. HTTP Server, FTP Server, SD-card, USB thumb drive.
    firmware_bin = b'\x01\x02\x03\x04\x05'
    success = internal_storage.write_firmware(firmware_bin)
    if success:
        print("Firmware written successfully.")
    else:
        print("Failed to write firmware.")
except Exception as e:
    print("Error writing firmware:", e)

# Step 8: Get newest instrument configuration ID
try:
    print("\nFetching newest instrument configuration ID...")
    newest_config_id = internal_storage.get_newest_instrument_config()
    if newest_config_id is not None:
        print("Newest Instrument Config ID:", newest_config_id)
    else:
        print("No instrument configurations available.")
except Exception as e:
    print("Error getting newest instrument config:", e)

# Step 9: Get oldest instrument configuration ID
try:
    print("\nFetching oldest instrument configuration ID...")
    oldest_config_id = internal_storage.get_oldest_instrument_config()
    if oldest_config_id is not None:
        print("Oldest Instrument Config ID:", oldest_config_id)
    else:
        print("No instrument configurations available.")
except Exception as e:
    print("Error getting oldest instrument config:", e)
```

```
# Step 10: Read virtual instrument configuration
try:
    print("\nReading virtual instrument configuration...")
    virtual_config = internal_storage.read_virtual_instrument_config()
    print("Virtual Instrument Configuration:", virtual_config)
except Exception as e:
    print("Error reading virtual instrument config:", e)

# Step 11: Read data upload configuration
try:
    print("\nReading data upload configuration...")
    data_upload_config = internal_storage.read_data_upload_config()
    print("Data Upload Configuration:", data_upload_config)
except Exception as e:
    print("Error reading data upload config:", e)
```

3.8. Logger system

Provides functions for system time management, wakeup scheduling, watchdog operations, overall system status reporting and delaying execution.

Constants

```
logger.NETIF_ETHERNET = 0
logger.NETIF_MODEM_A = 1
logger.NETIF_MODEM_B = 2
logger.UNKNOWN = 0
logger.POWER_ON_RESET = 1
logger.RTC_WAKEUP = 2
logger.GPIO_WAKEUP = 3
logger.WATCHDOG = 4
```

Enumeration to be used in a call to `http.send()`, `http_client.get_request()`, `http_client.post_request()`, `ftp_client.xxx()` etc functions as argument for **netif_id** and **netif** parameters.

Functions

- `wakeup_at(wakeup_at_time)`
  Schedules a wake-up at a specific UTC time.
- `wakeup_after(wakeup_after_s)`
  Schedules a wake-up after given seconds.
- `sleep()`
  Enters deep sleep mode (restarts device).
- `mult(a, b)`
  Multiplies two numbers and returns result.
- `set_time(time_string)`
  Sets system clock time in UTC.
- `set_time_utc(time_int_obj)`
  Sets the system clock time (UTC time in seconds)
- `get_time()`
  Gets system time as a formatted string.
- `get_timestamp()`
  Gets system time as a UNIX timestamp.

- `delay_ms(millis)`
  Delays Python execution for specified milliseconds.
- `get_status()`
  Returns current system and battery status.
- `watchdog_enable(sec_timeout)`
  Activates the watchdog with given timeout.
- `watchdog_disable()`
  Deactivates the watchdog feature.
- `watchdog_reset()`
  Resets the watchdog counter.
- `set_measuring_cycle_status(status, cycle_done=None)`
  Logs measurement cycle details.

`logger.wakeup_at(wakeup_at_time)`

Set time for next wake up at time given by **wakeup_at_time.**

Parameters:

- **wakeup_at_time:** *string* expected format "YYYY-MM-DD hh:mm:ss" in UTC timezone

Return value:

- *boolean*, `True` on success

Raises:

- *TypeError* if **wakeup_at_time** is of incorrect type OR internal error
- *ValueError* if **wakeup_at_time** value is in past or in future but less than 10 s or more than 10 h

`logger.wakeup_after(wakeup_after_s)`

Set time for next wake up at current time incremented by value **wakeup_after_s** (in seconds).

Parameters:

- **wakeup_after_s:** *integer*, number of seconds after current time when SOP will wake up.

Return value:

- *boolean*, `True` on success

Raises:

- *TypeError* if **wakeup_after_s** is of incorrect type
- *ValueError* if **wakeup_after_s** value is after timestamp less then 10 s or more then 10 h OR internal error

`logger.sleep()`

Puts the system into deep sleep mode. Effectively restarts the SOP.

Parameters: *None*

Return value:

- *boolean*, `True` on success , (but will restart before returning)

Raises:

- *ValueError* on internal error

`logger.mult(a, b)`

Multiply two numbers

Parameters:

- **a:** *float* or *integer,* first operand
- **b:** *float* or *integer, second operand*

Return value:

- *float,* result of multiplication, product of **a** and **b**

Raises: *None*

`logger.set_time(time_string)`

Sets the system clock time, which is in UTC time zone.

Parameters:

- **time_string:** *string,* time and date in the following format: "YYYY-MM-DD hh:mm:ss"; in UTC time zone

Return value:

- *boolean,* `True` on success, `False` otherwise

Raises:

- *TypeError* if **time_string** is of incorrect type

`logger.set_time_utc(time_int_obj)`

Sets the system clock time (UTC time in seconds)

Parameters:

- **time_int_obj:** *integer,* time in seconds as UNIX timestamp, in UTC time zone

Return value:

- *boolean,* `True` on success, `False` otherwise

Raises:

- *TypeError* if **time_int_obj** is of incorrect type
- *ValueError* if **time_int_obj** is less or equal than 0

`logger.get_time()`

Returns the system clock time as string in "YYYY-MM-DD hh:mm:ss" format in UTC time zone.

Parameters: *None*

Return value:

- *None* if cannot get system time (internal error) or
- *string,* the current time in the following format: "YYYY-MM-DD hh:mm:ss"; in UTC time zone

Raises: *None*

`logger.get_timestamp()`

Gets the system clock time, as UNIX timestamp.

Parameters: *None*

Return value:

- *integer,* current system clock time, as UNIX timestamp

Raises: *None*

`logger.delay_ms(millis)`

Pauses the execution of Python script for a given number of milliseconds

Parameters:

- **millis:** *integer,* time duration, in milliseconds, to pause the execution of Python script

Return value: *None*

Raises:

- *TypeError* if **millis** is of incorrect type

`logger.get_status()`

Returns the current datalogger status as a dictionary.

Parameters: *None*

Return value:

- *dictionary* of the following structure:
    - **temperature:** *float,* on-board temperature in °C, range of values from -45.0 to 130.0
    - **relative_humidity:** *float,* on-board relative humidity in %, range of values from 0.0 to 100.0
    - **input_voltage:** *float,* voltage at the input terminals in millivolts (mV)
    - **battery_voltage:** *float,* voltage at the battery terminals in millivolts (mV)
    - **charging_current:** *float,* current at the input terminals in milliamperes (mA), positive value
    - **discharging_current:** *float,* current at the battery terminals in milliamperes (mA), negative value
    - **battery_gauge_temperature:** float, battery gauge chip temperature in degrees Celsius (°C)
    - **battery_gauge_coulombs:** *float,* accumulated charge in battery, in Coulombs (C)
    - **battery_gauge_Ah:** *float,* accumulated charge in battery, in milliampere-hours (mAh)
    - **battery_gauge_current:** *float,* current at battery terminals in milliamperes (mA)
    - **ethernet_link_status:** *boolean,* `True` if Ethernet link is **UP**, `False` if Ethernet link is **DOWN**
    - **firmware_version:** *string,* firmware version in format MAJOR.MINOR.PATCH (ex “1.0.0”)
    - **firmware_build_string:** *string,* uniquely identifies the build of firmware image
    - **script_config_id:** 64-bit *integer*, internal configuration ID of the current Python script; this includes encoded timestamp
    - **start_event:** *integer,* must be one of the constants `logger.UNKNOWN, logger.POWER_ON_RESET, logger.RTC_WAKEUP, logger.GPIO_WAKEUP, logger.WATCHDOG`

Raises: *None*

`logger.watchdog_enable(sec_timeout)`

Activates a monitoring system that automatically restarts the Python script if it becomes unresponsive.

Parameters:

- **sec_timeout:** *integer,* Number of seconds before considering the program unresponsive (must be **> 0**)

Return value:

- *boolean*, `True` if watchdog activation succeeds

Raises:

- *TypeError* if **sec_timeout** is of incorrect type
- *ValueError* on internal error

`logger.watchdog_disable()`

Deactivates the automatic monitoring and restart functionality of Python script.

Parameters: *None*

Return value:

- *boolean*, `True` if deactivation succeeds

Raises:

- *ValueError* on internal error

`logger.watchdog_reset()`

Resets the internal counter of the Watchdog for the Python script to its initial state.

Parameters: *None*

Return value:

- *boolean*, `True` Always

Raises: *None*

`logger.set_measuring_cycle_status(status [ , cycle_done = False ] )`

Merges the provided **status** information into the current measurement-cycle Status Log. The Status Log can be retrieved and sent to the PC application to display useful info to the users.

Parameters:

- **status:** *dictionary* with hierarchical status information
- **cycle_done:** Optional *boolean*, `True` when this is the last Log entry in the cycle (resets this Log) , `False` otherwise

Return value:

- *dictionary* holding merged status data

Raises:

- *TypeError* if any argument is of incorrect type

Example

1. Reads and Updates System Time
2. Sets a wakeup at a specific UTC timestamp.
3. Sets a wakeup after a delay.
4. Enables Watchdog Timer - Ensures the system resets if it hangs.
5. Fetches real-time system parameters.
6. Performs Arithmetic Operations - Demonstrates the built-in multiplication function.
7. Implements Execution Delay - Uses delay_ms() for controlled wait periods.
8. Merges new cycle data and logs the status.
9. Resets watchdog counter periodically.
10. Disables watchdog before shutdown.
11. Puts System into Sleep Mode

```
import logger

# Step 1: Get and Set System Time
try:
    print("\nFetching current system time...")
    current_time = logger.get_time()
    print("Current System Time:", current_time)

    print("\nSetting new system time...")
    success = logger.set_time("2025-02-04 10:30:00")  # Example new time
    if success:
        print("System time updated successfully.")
    else:
        print("Failed to update system time.")
except Exception as e:
    print("Error handling system time:", e)

# Step 2: Fetch and Print Current System Timestamp
try:
    print("\nFetching system timestamp...")
    timestamp = logger.get_timestamp()
    print("Current UNIX Timestamp:", timestamp)
except Exception as e:
    print("Error fetching system timestamp:", e)

# Step 3: Set Wakeup Schedule
try:
    print("\nSetting wakeup time to a specific UTC timestamp...")
    wakeup_success = logger.wakeup_at("2025-02-05 08:00:00")
    if wakeup_success:
        print("Wakeup time set successfully.")
    else:
        print("Failed to set wakeup time.")
except Exception as e:
    print("Error setting wakeup time:", e)

try:
    print("\nSetting wakeup timer after 300 seconds...")
    wakeup_after_success = logger.wakeup_after(300)
    if wakeup_after_success:
        print("Wakeup timer set successfully.")
    else:
        print("Failed to set wakeup timer.")
except Exception as e:
    print("Error setting wakeup timer:", e)
```

```
# Step 4: Enable Watchdog
try:
    print("\nEnabling watchdog with a timeout of 120 seconds...")
    watchdog_success = logger.watchdog_enable(120)
    if watchdog_success:
        print("Watchdog enabled successfully.")
    else:
        print("Failed to enable watchdog.")
except Exception as e:
    print("Error enabling watchdog:", e)

# Step 5: Fetch and Display System Status
try:
    print("\nFetching system status...")
    status = logger.get_status()
    print("System Status:", status)
except Exception as e:
    print("Error fetching system status:", e)

# Step 6: Perform a Multiplication Operation
try:
    print("\nPerforming multiplication of 7.5 and 2.3...")
    result = logger.mult(7.5, 2.3)
    print("Expected value is    : 17.25")
    print("Multiplication Result:", result)
except Exception as e:
    print("Error performing multiplication:", e)

# Step 7: Implement a Delay in Execution
try:
    print("\nDelaying execution for 2 seconds (2000 ms)...")
    time_before = logger.get_timestamp()
    logger.delay_ms(2000)
    time_after = logger.get_timestamp()
    print("Delay completed.")
    print("Elapsed time (in seconds):", time_after - time_before)
except Exception as e:
    print("Error executing delay:", e)

# Step 8: Update Measurement Cycle Status
try:
    print("\nLogging measurement cycle status...")
    cycle_status = {"Sensor Status": {"Temperature": 22.5, "Humidity": 55.0}}
    merged_status = logger.set_measuring_cycle_status(cycle_status)
    print("Updated Measurement Cycle Status:", merged_status)
except Exception as e:
    print("Error updating measurement cycle status:", e)

# Step 9: Reset Watchdog Timer
try:
    print("\nResetting watchdog timer...")
    watchdog_reset_success = logger.watchdog_reset()
    print("Watchdog timer reset successfully.")
except Exception as e:
    print("Error resetting watchdog:", e)

# Step 10: Disable Watchdog
try:
    print("\nDisabling watchdog...")
    watchdog_disable_success = logger.watchdog_disable()
    if watchdog_disable_success:
        print("Watchdog disabled successfully.")
    else:
        print("Failed to disable watchdog.")
except Exception as e:
    print("Error disabling watchdog:", e)
```

```
# Step 11: Put System to Sleep
try:
    print("\nPutting system into deep sleep mode...")
    logger.sleep()  # Note: System will reboot before returning
except Exception as e:
    print("Error putting system to sleep:", e)
```

3.9. Modbus master

Provides master-side Modbus functions to query and command Modbus slave devices.

Constants

```
modbus.STATUS_IDLE = -1
modbus.STATUS_OK = 0
modbus.STATUS_EXCEPTION = 1
modbus.STATUS_PACKET_ERROR = 2
modbus.STATUS_CRC_ERROR = 3
modbus.STATUS_BUS_ERROR = 4
modbus.STATUS_TIMEOUT_ERROR = 5
modbus.STATUS_ADDRESS_ERROR = 6
modbus.STATUS_FN_CODE_ERROR = 7
modbus.STATUS_GENERAL_ERROR = 8
```

Enumeration used in returned dictionary entry **status.**

Functions

- `read_coils(port, device_id, start_address, count)`
  Reads coil values from a slave device.
- `read_discrete_inputs(port, device_id, start_address, count)`
  Reads discrete inputs from a slave device.
- `read_holding_registers(port, device_id, start_address, count)`
  Reads holding registers from a slave device.
- `read_input_registers(port, device_id, start_address, count)`
  Reads input registers from a slave device.
- `write_coils(port, device_id, start_address, values)`
  Writes boolean coil values to a slave device.
- `write_holding_registers(port, device_id, start_address, values)`
  Writes integer registers to a slave device.
- `read_exception_status(port, device_id)`
  Reads and returns the exception status.

```
modbus.read_coils(port, device_id, start_address, count)
```

Reads coils from the Modbus slave device.

Parameters:

- **port:** *integer*, the numeric port identifier, argument value must be between 0 and 2, where 0 = RS485_1, 1 = RS485_2, 2 = RS485_3
- **device_id:** *integer*, the Modbus slave ID, argument value must be between 1 and 247
- **start_address:** *integer*, the starting coil address number, argument value must be between 1 and 65535
- **count:** *integer*, number of coils to read

Return value:

- *None* on internal error
- *dictionary* containing three elements:
  - **status:** *: integer,* the status result which can be any constant for status or error such as `modbus.STATUS_OK`, `modbus.STATUS_EXCEPTION`, `modbus.STATUS_CRC_ERROR`, etc.
  - **values:** *list* of integer result values
  - **exception:** *integer,* Modbus exception code if the returned status is `modbus.STATUS_EXCEPTION`

Raises:

- *TypeError* if any argument is of incorrect type
- *ValueError* if there is a problem with arguments

```
modbus.read_discrete_inputs(port, device_id, start_address, count)
```

Reads discrete inputs from the Modbus slave device.

Parameters:

- **port:** *integer*, the numeric port identifier, argument value must be between 0 and 2, where 0 = RS485_1, 1 = RS485_2, 2 = RS485_3
- **device_id:** *integer*, the Modbus slave ID, argument value must be between 1 and 247
- **start_address:** *integer*, the starting input address number, argument value must be between 1 and 65535
- **count:** *integer*, number of coils to read

Return value:

- *dictionary* containing three elements:
  - **status:** *: integer,* the status result which can be `modbus.STATUS_OK` or any error status `modbus.STATUS_xxx_ERROR`
  - **values:** *list* of integer result values
  - **exception:** *integer,* Modbus exception code if the returned status is `modbus.STATUS_EXCEPTION`

Raises:

- *TypeError* if any argument is of incorrect type
- *ValueError* if there is a problem with arguments

`modbus.read_holding_registers(port, device_id, start_address, count)`

Reads holding registers from the Modbus slave device.

Parameters:

- **port:** *integer*, the numeric port identifier, argument value must be between 0 and 2, where 0 = RS485_1, 1 = RS485_2, 2 = RS485_3
- **device_id:** *integer*, the Modbus slave ID, argument value must be between 1 and 247
- **start_address:** *integer*, the starting input address number, argument value must be between 1 and 65535
- **count:** *integer*, number of registers to read

Return value:

- *dictionary* containing three elements:
    - **status:** : *integer,* the status result which can be `modbus.STATUS_OK` or any error status `modbus.STATUS_xxx_ERROR`
    - **values:** *list* of positive/unsigned integer result values
    - **exception:** *integer,* Modbus exception code if the returned status is `modbus.STATUS_EXCEPTION`

Raises:

- *TypeError* if any argument is of incorrect type
- *ValueError* if there is a problem with arguments

`modbus.read_input_registers(port, device_id, start_address, count)`

Reads input registers from the Modbus slave device.

Parameters:

- **port:** *integer*, the numeric port identifier, argument value must be between 0 and 2, where 0 = RS485_1, 1 = RS485_2, 2 = RS485_3
- **device_id:** *integer*, the Modbus slave ID, argument value must be between 1 and 247
- **start_address:** *integer*, the starting input address number, argument value must be between 1 and 65535
- **count:** *integer*, number of registers to read

Return value:

- *dictionary* containing three elements:
    - **status:** : *integer,* the status result which can be `modbus.STATUS_OK` or any error status `modbus.STATUS_xxx_ERROR`
    - **values:** *list* of positive/unsigned integer result values
    - **exception:** *integer,* Modbus exception code if the returned status is `modbus.STATUS_EXCEPTION`

Raises:

- *TypeError* if any argument is of incorrect type
- *ValueError* if there is a problem with arguments

`modbus.write_coils(port, device_id, start_address, values)`

Writes coil values to the Modbus device.

Parameters:

- **port:** *integer*, the numeric port identifier, argument value must be between 0 and 2, where 0 = RS485_1, 1 = RS485_2, 2 = RS485_3
- **device_id:** *integer*, the Modbus slave ID, argument value must be between 1 and 247
- **start_address:** *integer*, the starting input address number, argument value must be between 1 and 65535
- **values:** *list* of boolean values to write

Return value:

- *dictionary* containing containing two elements:
    - **status:** *: integer,* the status result which can be `modbus.STATUS_OK` or any error status `modbus.STATUS_xxx_ERROR`
    - **exception:** *integer,* Modbus exception code if the returned status is `modbus.STATUS_EXCEPTION`

Raises:

- *TypeError* if any argument is of incorrect type
- *ValueError* if there is a problem with arguments

`modbus.write_holding_registers(port, device_id, start_address, values)`

Writes holding register values to the Modbus device.

Parameters:

- **port:** *integer*, the numeric port identifier, argument value must be between 0 and 2, where 0 = RS485_1, 1 = RS485_2, 2 = RS485_3
- **device_id:** *integer*, the Modbus slave ID, argument value must be between 1 and 247
- **start_address:** *integer*, the starting input address number, argument value must be between 1 and 65535
- **values:** *: list* of integers values to write, 16-bit values, the function will automatically switch between unsigned and signed

Return value:

- *dictionary* containing two elements:
    - **status:** *: integer,* the status result which can be `modbus.STATUS_OK` or any error status `modbus.STATUS_xxx_ERROR`
    - **exception:** *integer,* Modbus exception code if the returned status is `modbus.STATUS_EXCEPTION`

Raises:

- *TypeError* if any argument is of incorrect type
- *ValueError* if there is a problem with arguments

```
modbus.read_exception_status(port, device_id)
```

Reads exception status from the Modbus device.

Parameters:

- **port:** *integer*, the numeric port identifier, argument value must be between 0 and 2, where 0 = RS485_1, 1 = RS485_2, 2 = RS485_3
- **device_id:** *integer*, the Modbus slave ID, argument value must be between 1 and 247

Return value:

- *dictionary* containing two elements:
    - **status:** : *integer,* the status result which can be `modbus.STATUS_OK` or any error status `modbus.STATUS_xxx_ERROR`
    - **exception:** *integer,* Modbus exception code if the returned status is `modbus.STATUS_EXCEPTION`

Raises:

- *TypeError* if any argument is of incorrect type
- *ValueError* if there is a problem with arguments

Example

Prerequisite: Modbus slave device is connected at RS485_2 port and is independently supplied with power.

1. Configures communication port
2. Retrieves coil states from a Modbus slave device.
3. Reads discrete input values for monitoring.
4. Retrieves 16-bit holding registers from the Modbus slave.
5. Reads Input Registers - Fetches analog input values.
6. Sets coil values on the Modbus slave.
7. Sends new values to holding registers.
8. Checks if the Modbus device has encountered an error.

```
import modbus
import serial  # Import serial for configuring port

modbus_status_string = {
    modbus.STATUS_IDLE : "STATUS_IDLE",
    modbus.STATUS_OK : "STATUS_OK",
    modbus.STATUS_EXCEPTION : "STATUS_EXCEPTION",
    modbus.STATUS_PACKET_ERROR : "STATUS_PACKET_ERROR",
    modbus.STATUS_CRC_ERROR : "STATUS_CRC_ERROR",
    modbus.STATUS_BUS_ERROR : "STATUS_BUS_ERROR",
    modbus.STATUS_TIMEOUT_ERROR : "STATUS_TIMEOUT_ERROR",
    modbus.STATUS_ADDRESS_ERROR : "STATUS_ADDRESS_ERROR",
    modbus.STATUS_FN_CODE_ERROR : "STATUS_FN_CODE_ERROR",
    modbus.STATUS_GENERAL_ERROR : "STATUS_GENERAL_ERROR"
}

# Modbus port settings
modbus_port = 1    # RS485_2
device_id = 1      # Modbus slave ID
start_address = 1  # Starting address for registers/coils
num_items = 5      # Number of registers/coils to read/write

# Step 1: Configure Modbus port settings
serial.configure(serial.RS485_2, 9600, serial.PARITY_EVEN, serial.DATA_EIGHT, serial.STOP_ONE)

# Step 2: Read Coils
try:
    print("\nReading coils from Modbus slave...")
    response = modbus.read_coils(modbus_port, device_id, start_address, num_items)
    if response['status'] == modbus.STATUS_OK:
        print("Coil Values:", response['values'])
    else:
        print("Error reading coils. Modbus Status:", modbus_status_string.get(response['status']))
except Exception as e:
    print("Error reading coils:", e)

# Step 3: Read Discrete Inputs
try:
    print("\nReading discrete inputs from Modbus slave...")
    response = modbus.read_discrete_inputs(modbus_port, device_id, start_address, num_items)
    if response['status'] == modbus.STATUS_OK:
        print("Discrete Input Values:", response['values'])
    else:
        print("Error reading discrete inputs. Modbus Status:", modbus_status_string.
get(response['status']))
except Exception as e:
    print("Error reading discrete inputs:", e)

# Step 4: Read Holding Registers
try:
    print("\nReading holding registers from Modbus slave...")
    response = modbus.read_holding_registers(modbus_port, device_id, start_address, num_items)
    if response['status'] == modbus.STATUS_OK:
        print("Holding Register Values:", response['values'])
    else:
        print("Error reading holding registers. Modbus Status:", modbus_status_string.
get(response['status']))
except Exception as e:
    print("Error reading holding registers:", e)

# Step 5: Read Input Registers
try:
    print("\nReading input registers from Modbus slave...")
    response = modbus.read_input_registers(modbus_port, device_id, start_address, num_items)
    if response['status'] == modbus.STATUS_OK:
        print("Input Register Values:", response['values'])
    else:
        print("Error reading input registers. Modbus Status:", modbus_status_string.
get(response['status']))
except Exception as e:
    print("Error reading input registers:", e)

# Step 6: Write to Coils
coil_values = [1, 0, 1, 1, 0]  # Example coil states to write
try:
    print("\nWriting to coils on Modbus slave...")
    response = modbus.write_coils(modbus_port, device_id, start_address, coil_values)
    if response['status'] == modbus.STATUS_OK:
        print("Successfully wrote to coils.")
    else:
        print("Error writing coils. Modbus Status:", modbus_status_string.get(response['status']))
except Exception as e:
    print("Error writing coils:", e)

# Step 7: Write to Holding Registers
register_values = [100, 200, 300, 400, 500]  # Example register values to write
try:
    print("\nWriting to holding registers on Modbus slave...")
    response = modbus.write_holding_registers(modbus_port, device_id, start_address, register_
values)
    if response['status'] == modbus.STATUS_OK:
        print("Successfully wrote to holding registers.")
    else:
        print("Error writing to holding registers. Modbus Status:", modbus_status_string.
get(response['status']))
except Exception as e:
    print("Error writing holding registers:", e)

# Step 8: Read Exception Status
try:
    print("\nReading exception status from Modbus slave...")
    response = modbus.read_exception_status(modbus_port, device_id)
    if response['status'] == modbus.STATUS_OK:
        print("Exception Status Code:", response['exception'])
    else:
        print("Error reading exception status. Modbus Status:", modbus_status_string.
get(response['status']))
except Exception as e:
    print("Error reading exception status:", e)
```

3.10. Modem control

Controls cellular modem connectivity and retrieves both basic and detailed network status information.

Constants

`modem.STATUS_OFF = 0`

the modem is currently turned off and needs to be connected to the network in order to be used

`modem.STATUS_CONNECTING = 1`

the modem is opening a connection; more details on the connection stage can be retrieved by calling `modem.get_full_status()`

`modem.STATUS_CONNECTED = 2`

the modem is connected to the network

`modem.STATUS_DISCONNECTING = 3`

the modem is disconnecting from the network; it cannot be used anymore for ending/receiving the data, but the disconnect process is not completed yet

Detailed modem state

`modem.STATE_TURNING_ON = 0`

the modem is currently being turned ON - the power is applied, but no commands were sent to the modem yet

`modem.STATE_NOT_RESPONDING = 1`

the modem is powered up, and initialization commands were sent to the modem; the modem (still) has not responded to any of the commands; it is normal for `modem.STATE_NOT_RESPONDING` state to last for few seconds up to 20 seconds; if the modem stays longer in `modem.STATE_NOT_RESPONDING` state it is then an error

```
modem.STATE_INITIALIZING = 2
```

the modem is powered-up, it has responded to basic initialization commands and it is initializing; once the initialization is complete, the modem will try to attach to the cellular network

```
modem.STATE_SIM_NOT_FOUND = 3
```

the SIM card is not present, and the modem cannot connect to the network for that reason

```
modem.STATE_SIM_LOCKED = 4
```

the SIM card is locked (PIN or PUK need to be entered)

```
modem.STATE_SIM_ERROR = 5
```

there is some general problem with the SIM card - maybe it is not properly inserted in the slot, or is completely broken; try to re-insert it, or replace it with another SIM card

```
modem.STATE_SEARCHING_FOR_NETWORK = 6
```

the modem is now searching for the cellular network; depending on the SIM card status and signal strength, this can take a while (sometimes for newly activated cards, it can take up to 5-10 minutes)

```
modem.STATE_CONNECTED_TO_NETWORK = 7
```

the modem is connected to the cellular network; but still there is no data connection, however SMS messages can be sent and received - the modem will now try to open data connection as well

```
modem.STATE_OPENING_DATA_LINK = 8
```

the modem is connected to the cellular network and is now trying to open data connection

```
modem.STATE_DIALING_PPP = 9
```

the modem is connected to the cellular network, basic data connection is ready and the PPP client is dialing/ opening PPP connection which is required to make the data link ready for connections to the Internet

```
modem.STATE_PPP_DIAL_FAILED = 10
```

the modem failed to establish PPP connection - check if the APN, Username and Password fields are correct, and that the SIM card has not reached monthly data limit, or that the SIM card has not expired

```
modem.STATE_CONNECTED_TO_INTERNET = 11
```

the modem is connected to the Internet

```
modem.STATE_GENERAL_ERROR = 255
```

unknown/unspecified error occurred. If this error is encountered, it is recommended to fully power down the modem, wait a few seconds, and the restart it.

Network connection status

```
modem.NET_NOT_REGISTERED = 0
```

the modem is not registered/attached to any network

```
modem.NET_REGISTERED_HOME = 1
```

the modem is registered/attached to the default home network

```
modem.NET_SEARCHING = 2
```

the modem is scanning/searching for available networks

```
modem.NET_REGISTRATION_DENIED = 3
```

the modem has failed to register/attach to the network - check that the SIM card has been activated, and that it has not expired (due to unpaid bills for example)

```
modem.NET_UNKNOWN = 4
```

unknown error related to the registration of the modem to the network

```
modem.NET_REGISTERED_ROAMING = 5
```

the modem is registered/attached to the roaming network

Functions

- `connect(apn, username=None, password=None)`
  Initiates connection using APN, optional credentials.
- `disconnect()`
  Starts modem disconnect sequence.
- `get_status()`
  Returns a numeric status indicating the connection state (using one of the defined status constants).
- `get_full_status()`
  Returns detailed modem status and cellular network parameters.

```
modem.connect(apn [ , username, password ] )
```

Starts modem connection process.

Parameters:

- **apn:** *string*, the GPRS APN
- **username:** Optional *string*, username for the connection, can be left empty
- **password:** Optional *string*, username for the connection, can be left empty

Return value:

- *boolean*, `True` if connection sequence has started; `False` if the modem is already connected

Raises:

- *TypeError* if any argument is of incorrect type
- *ValueError* if there is a problem with arguments

`modem.disconnect()`

Initiates modem disconnect sequence. If the modem is not connected, no action is taken.

Parameters: *None*

Return value: *None*

Raises: *None*

`modem.get_status()`

Returns the current connection state of the modem. The state is one of the constants: `modem.STATUS_OFF, modem.STATUS_CONNECTING, modem.STATUS_CONNECTED, modem.STATUS_DISCONNECTING`

Parameters: *None*

Return value:

- *integer,* current modem connection state

Raises: *None*

`modem.get_full_status()`

Returns the dictionary holding the keys that describe the detailed status of the modem. If the modem is turned OFF (so that no detailed status is available), this function returns *None*.

Parameters: *None*

Return value:

- *None* in case when the modem is turned OFF so there is no status available OR
- *dictionary* containing the following keys:
    - **modem_state:** *integer*, current modem state, see Detailed modem state constants `modem.`
    - `STATE_xxx`
    - **network_status:** *integer,* the network attachment status, see network status constants
    - `modem.NET_xxx`
    - **rssi:** *integer,* last measured RSSI (signal strength) from the modem
    - **imei:** *string,* the modem IMEI (International Mobile Equipment Identity), empty if no modem
    - **modem_name:** *string,* the name by which the modem device identifies itself, empty if no
    - modem
    - **imsi:** *string,* the SIM card IMSI (International Mobile Subscriber Identity), empty if no SIM
    - card OR SIM error
    - **network_type:** *string* which identifies the active cellular network System (LTE, GSM, etc.)
    - **network_mode:** *string* describing the network status (Online, Offline, etc)
    - **mcc:** *integer,* Mobile Country Code identifier of the network to which the modem is
    - attached
      mnc: *integer,* Mobile Network Code identifier of the network to which the modem is
    - attached
    - **lac:** Optional *integer*, Location Area Code of the cellular tower to which the modem is attached
    - **cell_id:** Optional *integer*, the Cell ID of the cellular tower to which the modem is attached
    - **tac:** Optional *integer*, Tracking Area Code of the cellular tower to which the modem is attached
    - **s_cell_id:** Optional *integer*, the S-cell ID of the cellular tower to which the modem is attached
    - **p_cell_id:** Optional *integer*, the P-cell ID of the cellular tower to which the modem is attached
    - **channel_name:** *string*, the descriptive name of the channel/band currently used by the modem, can be empty
    - **ip_address:** *string*, The IP address assigned to the open connection, can be empty

Raises: *None*

Example

Prerequisite: Modbus slave device is connected at RS485_2 port and is independently supplied with power.

1. Starts a timer using logger.get_timestamp() to track elapsed time.
2. Checks the modem's status every 5 seconds:
    1. Prints detailed modem state each cycle using `modem.get_full_status()`.
    2. If `modem.STATUS_OFF`, it stops waiting (modem is turned off).
    3. If `modem.STATUS_CONNECTED`, it stops waiting (connection successful).
3. Uses `logger.delay_ms(15000)` to wait 5 seconds per cycle.
4. Exits after 10 minutes (600 seconds) if neither condition is met.

```
import modem
import logger  # Import logger to get timestamps

# Status string lookup dictionaries
modem_status_string = {
    modem.STATUS_OFF : "STATUS_OFF",
    modem.STATUS_CONNECTING : "STATUS_CONNECTING",
    modem.STATUS_CONNECTED : "STATUS_CONNECTED",
    modem.STATUS_DISCONNECTING : "STATUS_DISCONNECTING"
}

modem_state_string = {
    modem.STATE_GENERAL_ERROR : "STATE_TURNING_ON",
    modem.STATE_CONNECTED_TO_INTERNET : "STATE_NOT_RESPONDING",
    modem.STATE_PPP_DIAL_FAILED : "STATE_INITIALIZING",
    modem.STATE_DIALING_PPP : "STATE_SIM_NOT_FOUND",
    modem.STATE_OPENING_DATA_LINK : "STATE_SIM_LOCKED",
    modem.STATE_CONNECTED_TO_NETWORK : "STATE_SIM_ERROR",
    modem.STATE_SEARCHING_FOR_NETWORK : "STATE_SEARCHING_FOR_NETWORK",
    modem.STATE_SIM_ERROR : "STATE_CONNECTED_TO_NETWORK",
    modem.STATE_SIM_LOCKED : "STATE_OPENING_DATA_LINK",
    modem.STATE_SIM_NOT_FOUND : "STATE_DIALING_PPP",
    modem.STATE_INITIALIZING : "STATE_PPP_DIAL_FAILED",
    modem.STATE_NOT_RESPONDING : "STATE_CONNECTED_TO_INTERNET",
    modem.STATE_TURNING_ON : "STATE_GENERAL_ERROR"
}

network_status_string = {
    modem.NET_NOT_REGISTERED : "NET_NOT_REGISTERED",
    modem.NET_REGISTERED_HOME : "NET_REGISTERED_HOME",
    modem.NET_SEARCHING : "NET_SEARCHING",
    modem.NET_REGISTRATION_DENIED : "NET_REGISTRATION_DENIED",
    modem.NET_UNKNOWN : "NET_UNKNOWN",
    modem.NET_REGISTERED_ROAMING : "NET_REGISTERED_ROAMING",
}

# Modem configuration parameters
apn = "internet.apn"  # Set APN for network connection
username = "user"  # Optional username
password = "pass"  # Optional password

# Maximum wait time for connection (10 minutes = 600 seconds)
max_wait_time = 600  # seconds (10 minutes)
wait_interval = 5  # seconds (each cycle)
elapsed_time = 0  # Initialize elapsed time counter

# Step 1: Check the modem's initial status
try:
    print("\nChecking modem status before connecting...")
    status = modem.get_status()
    print("Modem status:", status, modem_status_string.get(status))
except Exception as e:
    print("Error getting modem status:", e)

# Step 2: Get full modem status (detailed information)
try:
    print("\nFetching full modem status...")
    full_status = modem.get_full_status()
    if full_status is not None:
        print("Full Modem Status:")
        for key, value in full_status.items():
            if key == "modem_status":
                print("    ", key, ":", value, modem_state_string.get(value))
            elif key == "network_status":
                print("    ", key, ":", value, network_status_string.get(value))
            else:
                print("    ", key, ":", value)
    else:
        print("Modem is OFF. No detailed status available.")
except Exception as e:
    print("Error getting full modem status:", e)

# Step 3: Connect the modem to the network
try:
    print("\nAttempting to connect the modem...")
    connected = modem.connect(apn, username, password)
    if connected:
        print("Modem connection sequence started successfully.")
    else:
        print("Modem is already connected.")
except Exception as e:
    print("Error connecting modem:", e)

# Step 4: Wait for modem to connect or fail, with timeout
try:
    print("\nWaiting for modem connection...")

    start_time = logger.get_timestamp()  # Get start timestamp

    while elapsed_time < max_wait_time:
        # Get current modem status
        status = modem.get_status()
         print("\nElapsed time:", elapsed_time, "seconds | Modem status:", status, modem_status_
string.get(status))

        # Get full status and print details
        full_status = modem.get_full_status()
        if full_status is not None:
            print("Modem Detailed Status:")
            for key, value in full_status.items():
                if key == "modem_status":
                    print("    ", key, ":", value, modem_state_string.get(value))
                elif key == "network_status":
                    print("    ", key, ":", value, network_status_string.get(value))
                else:
                    print("    ", key, ":", value)
        else:
            print("Modem is OFF. No detailed status available.")

        # If modem is OFF or CONNECTED, stop waiting
        if status == modem.STATUS_OFF:
            print("Modem is OFF. Stopping wait loop.")
            break
        elif status == modem.STATUS_CONNECTED:
            print("Modem successfully connected!")
            break

        # Wait for the next cycle
        logger.delay_ms(wait_interval * 1000)  # Convert seconds to milliseconds
        elapsed_time = logger.get_timestamp() - start_time  # Update elapsed time

    # Final status check
    print("\nFinal Modem Status Check...")
    final_status = modem.get_status()
    print("Final Modem Status:", final_status, modem_status_string.get(final_status))

except Exception as e:
    print("Error while waiting for modem connection:", e)

# Step 5: Retrieve and display system timestamp
try:
    print("\nFetching the current system timestamp...")
    current_time = logger.get_time()
    print("Current System Time:", current_time)
except Exception as e:
    print("Error getting system time:", e)

# Step 6: Disconnect the modem
try:
    print("\nAttempting to disconnect the modem...")
    modem.disconnect()
    print("Modem disconnect sequence initiated.")
except Exception as e:
    print("Error disconnecting modem:", e)

# Step 7: Check the modem's status after disconnection
try:
    print("\nFinal modem status check after disconnection...")
    status = modem.get_status()
    print("Final Modem Status:", status, modem_status_string.get(status))
except Exception as e:
    print("Error getting final modem status:", e)
```

3.11. Output power management

Manages power output ports by enabling/disabling them and measuring current and energy consumption.

Functions

- **`set(port, state)`**
  Enables or disables a power output port.
- **`get_current(ADC_VOUT_channel)`**
  Returns value of current at power output port.
- **`get_total_energy_consumption(ADC_VOUT_channel)`**
  Returns value of energy consumption at power output port.

```
output_power.set(port, state)
```

Sets the specified output power port to given state.

Parameters:

- **port:** *integer,* the output power port numeric identifier, argument value must be from 0 to 2
- **state:** *boolean*, specifying the desired output power state. `True` to turn the port ON, `False` to turn it OFF.

Return value: *None*

Raises:

- *TypeError* if any argument is of incorrect type
- *ValueError* if there is a problem with arguments

```
output_power.get_current(ADC_VOUT_channel)
```

Returns value of current at power output port.

Parameters:

- **ADC_VOUT_channel:** *integer,* the output power port numeric identifier, argument value must be from 1 to 3

Return value:

- *float*, value of current at power output port as requested by **ADC_VOUT_channel**, in amperes (A)

Raises:

- *TypeError* if any argument is of incorrect type
- *ValueError* if there is a problem with arguments OR internal error

```
output_power.get_total_energy_consumption(ADC_VOUT_channel)
```

Returns value of energy consumption at power output port.

Parameters:

- **ADC_VOUT_channel:** *integer,* the output power port numeric identifier, argument value must be from 1 to 3

Return value:

- *float*, value of energy consumption at power output port as requested by **ADC_VOUT_channel**, in Watt-hours (Wh)

Raises:

- *TypeError* if any argument is of incorrect type
- *ValueError* if there is a problem with arguments OR internal error

Example

Prerequisite: Modbus slave device is connected at RS485_2 port and is independently supplied with power.

1. Turn ON the power output port.
2. Measure current consumption for power output port.
3. Measure total energy consumption for the same power output port.
4. Turn OFF the power output port.

```python
import output_power
import logger  # Import logger for timestamps

# Define power output port and ADC channel
power_port = 0  # Port 0 (valid values: 0, 1, 2)
adc_power_port = power_port + 1  # numeric identifier for measurement (valid values: 1, 2, 3)

# Step 1: Log the system timestamp before power operations
try:
    print("\nFetching the current system timestamp before operations...")
    current_time = logger.get_time()
    print("Current System Time:", current_time)
except Exception as e:
    print("Error getting system time:", e)

# Step 2: Turn ON the specified power port
try:
    print("\nTurning ON power output port", power_port)
    output_power.set(power_port, True)
    print("Power output port", power_port, "is now ON.")
except Exception as e:
    print("Error setting power output:", e)

# Step 3: Wait a few seconds before measuring (simulate load operation)
try:
    print("\nWaiting 5 seconds to allow stable measurement...")
    logger.delay_ms(5000)  # Wait for 5 seconds
except Exception as e:
    print("Error during delay:", e)

# Step 4: Measure the current at the specified ADC channel
try:
    print("\nMeasuring current at ADC channel", adc_power_port)
    current_value = output_power.get_current(adc_power_port)
    print("Measured Current on ADC channel", adc_power_port, ":", current_value, "A")
except Exception as e:
    print("Error getting current measurement:", e)

# Step 5: Measure total energy consumption at the same ADC channel
try:
    print("\nMeasuring total energy consumption at ADC channel", adc_power_port)
    energy_consumption = output_power.get_total_energy_consumption(adc_power_port)
   print("Total Energy Consumption on ADC channel", adc_power_port, ":", energy_consumption, "Wh")
except Exception as e:
    print("Error getting energy consumption:", e)

# Step 6: Turn OFF the power output port
try:
    print("\nTurning OFF power output port", power_port)
    output_power.set(power_port, False)
    print("Power output port", power_port, "is now OFF.")
except Exception as e:
    print("Error setting power output:", e)

# Step 7: Log the system timestamp after power operations
try:
    print("\nFetching the system timestamp after operations...")
    final_time = logger.get_time()
    print("Final System Time:", final_time)
except Exception as e:
    print("Error getting system time:", e)
```

3.12. SDI-12

Used to send commands over the SDI-12 sensor bus and return measurement values or raw responses.

Constants

```
sdi12.STATUS_OK = 0
```

means that the SDI-12 call has completed successfully

```
sdi12.STATUS_TIMEOUT_ERROR = 1
```

the device did not respond to the sent command

```
sdi12.STATUS_PACKET_ERROR = 2
```

the packet returned by an SDI-12 device is not formatted correctly

```
sdi12.STATUS_COMMAND_ERROR = 3
```

there was a response which does not match the issued command

Functions

- `send(port, cmd)`
  Issues an SDI-12 command and returns results.

```
sdi12.send(port, cmd)
```

Sends the given command via SDI12 bus defined by **port**, and returns the result. Function sends the command asynchronously and waits for a response.

Parameters:

- **port:** *integer*, the numeric physical port identifier. Argument value must be 0 (SDI-12 port 0) or 1 (SDI-12 port 1)
- **cmd:** *string*, the command to be sent

Return value:

- *dictionary* containing three elements:
  - **status:** *: integer*, the status result which can be `sdi12.STATUS_OK` or an error status `sdi12.STATUS_TIMEOUT_ERROR` etc
  - **values:** *list* of *float*, containing result values, which are floating point measurement values; unless the issued command does not return measurement, when *None*, an empty list will be returned
  - **values:** *string*, containing raw response if the values list is empty

Raises:

- *TypeError* if any argument is of incorrect type
- *ValueError* if there is a problem with arguments

Example

Prerequisite: SDI-12 device is connected at SDI-12 port 0 and is independently supplied with power.

1. Send an SDI-12 Address query command, check response
2. Send an SDI-12 identification command to get sensor information.
3. Send an SDI-12 measurement command and handle the response.
4. Extract and display measured values, or raw response if no values are available.
5. Extract and display the identification response.

```
import sdi12
import logger  # Import logger for timestamps

# Define SDI-12 port
sdi12_port = 0  # Valid values: 0 or 1

# Step 1: Log the system timestamp before SDI-12 operations
try:
    print("\nFetching the current system timestamp before operations...")
    current_time = logger.get_time()
    print("Current System Time:", current_time)
except Exception as e:
    print("Error getting system time:", e)

# Step 2: Send an SDI-12 Address query command
try:
    print("\nSending SDI-12 Address query  command to port", sdi12_port)
    response = sdi12.send(sdi12_port, "?!")
    print("SDI-12 Address query  Command Response:", response)
except Exception as e:
    print("Error sending SDI-12 Address query  command:", e)

# Step 3: Check response status and extract Address query
try:
    status_code = response.get("status", -1)
    values = response.get("values", [])
    raw_response = response.get("response", "")
    if status_code == sdi12.STATUS_OK:
        print("SDI-12 command executed successfully.")
        if values:
            print("Extracted Address Values:", values)
        else:
            print("No Address query values returned, raw response:", raw_response)
    else:
        print("SDI-12 command returned an error, status code:", status_code)
except Exception as e:
    print("Error processing SDI-12 response:", e)

# Step 4: Send an SDI-12 identification command
try:
    print("\nSending SDI-12 identification command to port", sdi12_port)
    response = sdi12.send(sdi12_port, "0I!")
    print("SDI-12 Identification Command Response:", response)
except Exception as e:
    print("Error sending SDI-12 identification command:", e)

# Step 4: Extract and display identification response
try:
    status_code = response.get("status", -1)
    raw_response = response.get("response", "")
    if status_code == sdi12.STATUS_OK:
        print("SDI-12 Identification Response:", raw_response)
    else:
        print("SDI-12 identification command failed, status code:", status_code)
except Exception as e:
    print("Error processing SDI-12 identification response:", e)

# Step 5: Send an SDI-12 measurement command
try:
    print("\nSending SDI-12 measurement command to port", sdi12_port)
    response = sdi12.send(sdi12_port, "0M!")
    print("SDI-12 Measurement Command Response:", response)
except Exception as e:
    print("Error sending SDI-12 measurement command:", e)

# Step 6: Check response status and extract values if available
try:
    status_code = response.get("status", -1)
    values = response.get("values", [])
    raw_response = response.get("response", "")
    if status_code == sdi12.STATUS_OK:
        print("SDI-12 command executed successfully.")
        if values:
            print("Extracted Measurement Values:", values)
        else:
            print("No measurement values returned, raw response:", raw_response)

# Step 7: Wait before sending another command (simulating real-world operation)
        print("\nWaiting 10 seconds before sending another command...")
        logger.delay_ms(10000)  # Wait for 10 seconds

# Step 8: Send an SDI-12 Send data command
        print("\nSending SDI-12 Send data command to port", sdi12_port)
        response = sdi12.send(sdi12_port, "0D0!")
        print("SDI-12 Send data Command Response:", response)

# Step 9: Check response status and extract values if available
        status_code = response.get("status", -1)
        values = response.get("values", [])
        raw_response = response.get("response", "")
        if status_code == sdi12.STATUS_OK:
            print("SDI-12 command executed successfully.")
            if values:
                print("Extracted data Values:", values)
            else:
                print("No data values returned, raw response:", raw_response)
        else:
            print("SDI-12 command returned an error, status code:", status_code)
    else:
        print("SDI-12 command returned an error, status code:", status_code)
except Exception as e:
    print("Error processing SDI-12 response:", e)

# Step 10: Log the system timestamp after SDI-12 operations
try:
    print("\nFetching the system timestamp after operations...")
    final_time = logger.get_time()
    print("Final System Time:", final_time)
except Exception as e:
    print("Error getting system time:", e)

print("\nSDI-12 module example completed successfully!")
```

3.13. Serial ports

Configures UART ports and handles serial data transmission and reception across RS-232 and RS-485 interfaces.

NOTE

calling `serial.configure()` on RS-485 ports changes settings and can interfere with operation of modbus module (if `modbus` used). Use with care!

Constants

```
serial.RS232_1 = 0
serial.RS232_2 = 6
serial.RS485_1 = 2
serial.RS485_2 = 3
serial.RS485_3 = 7
```

Enumeration to be used in a call to `serial.configure()`, `serial.write()`, `serial.read()` as argument for port parameter. Each port name correlates to one physical serial **port** on SOP board.

```
serial.PARITY_EVEN = 0
serial.PARITY_NONE = 1
serial.PARITY_ODD = 2
serial.PARITY_MARK = 3
serial.PARITY_SPACE = 4
```

Enumeration to be used in a call to `serial.configure()` as argument for **parity_bits** parameter

```
serial.DATA_FIVE = 5
serial.DATA_SIX = 6
serial.DATA_SEVEN = 7
serial.DATA_EIGHT = 8
```

Enumeration to be used in a call to `serial.configure()` as argument for data_bits parameter

```
serial.STOP_ONE = 1
serial.STOP_TWO = 2
```

Enumeration to be used in a call to `serial.configure()` as argument for stop_bits parameter

Functions

- **`configure(port, baud_rate, parity_bits, data_bits, stop_bits)`**
  Sets serial port parameters.
- **`write(port, data)`**
  Transmits data through a serial port.
- **`read(port, num_bytes, max_timeout)`**
  Receives data from a serial port with timeout.

`serial.configure(port, baud_rate, parity_bits, data_bits, stop_bits)`

Configures serial **port** with arguments values. Arguments for all five parameters must be given. Be aware that calling `serial.configure()` on RS-485 ports changes settings and can interfere with operation of `modbus` module (if `modbus` used).

Parameters:

- **port:** *integer*, must be one of the constants for port `serial.RS232_1`, ..., `serial.RS485_3`
- **baud_rate:** *integer,* defines the speed at which port operates, given in bits per second, (must be >= 100)
- **parity_bits:** *integer,* must be one of the constants for parity_bits `serial.PARITY_NONE` etc
- **data_bits:** *integer,* must be one of the constants for data_bits `serial.DATA_EIGHT etc`
- **baud_rate:** *integer,* defines the speed at which port operates, given in bits per second, (must be >= 100)
- **stop_bits:** *integer,* must be one of the constants for stop_bits `serial.STOP_TWO` etc

Return value: *None*

Raises:

- *TypeError* if any argument is of incorrect type
- *ValueError* if any argument's value is out-of-range

`serial.write(port, data)`

Writes the data into transmit buffer of the serial port for sending. Arguments for all parameters must be given. Be aware that calling `serial.write()` on RS-485 ports can interfere with operation of `modbus` module (if used simultaneously).

Parameters:

- **port:** *integer*, must be one of the constants for port `serial.RS232_1`, ..., `serial.RS485_3`
- **data:** *bytearray* , object containing data to send

Return value:

- *integer*, number bytes written to transmit buffer

Raises:

- *TypeError* if any argument is of incorrect type
- *ValueError* if any argument's value is out-of-range

`serial.read(port, num_bytes, max_timeout)`

Read data bytes from receive buffer of serial **port**, wait until **max_timeout** expires or until **num_bytes** is available then return actual data bytes received. Arguments for all parameters must be given. Be aware that calling `serial.read()` on RS-485 ports can interfere with operation of `modbus` module (if used simultaneously).

Parameters:

- **port:** *integer*, must be one of the constants for port `serial.RS232_1`, ..., `serial.RS485_3`
- **num_bytes:** *integer,* sets a maximum number of bytes to get; if available, function returns this amount of bytes
- **max_timeout:** *integer,* given in units of milliseconds, sets a maximum wait period to wait for
- **num_bytes** to become available; if period expires function returns as much bytes as are available at moment of expiry

Return value:

- *bytearray* received data, returns empty bytearray if no bytes received

Raises:

- *TypeError* if any argument is of incorrect type
- *ValueError* if any argument's value is out-of-range

Example

Prerequisite: SDI-12 device is connected at SDI-12 port 0 and is independently supplied with power.

1. Configure the serial port with appropriate settings.
2. Send a test message over the serial port.
3. Read data from the serial port, handling timeout and received length.

```
import serial
import logger  # Import logger for timestamps and delays

# Port string dictionary
port_string = {
    serial.RS232_1 : "RS232_1",
    serial.RS232_2 : "RS232_2",
    serial.RS485_1 : "RS485_1",
    serial.RS485_2 : "RS485_2",
    serial.RS485_2 : "RS485_2"
}

# Define serial port parameters
serial_port = serial.RS232_2  # Valid values: RS232_1, RS232_2, RS485_1, RS485_2, RS485_3
baud_rate = 115200
parity_bits = serial.PARITY_NONE
data_bits = serial.DATA_EIGHT
stop_bits = serial.STOP_ONE

# Configure the serial port with the specified parameters
try:
    print("\nConfiguring serial port", port_string.get(serial_port), "with baud rate", baud_rate)
    serial.configure(serial_port, baud_rate, parity_bits, data_bits, stop_bits)
    print("Serial port configured successfully.")
except Exception as e:
    print("Error configuring serial port:", e)

# Send a test message over the serial port
try:
    data_to_send = bytearray(b"Hello, Serial Communication!")
    print("\nWriting data to serial port...")
    bytes_written = serial.write(serial_port, data_to_send)
    print("Number of bytes written:", bytes_written)
except Exception as e:
    print("Error writing to serial port:", e)

# Wait a moment before reading (simulate real-world operation)
try:
    print("\nWaiting 2 seconds before reading data...")
    logger.delay_ms(2000)  # Wait for 2 seconds
except Exception as e:
    print("Error during delay:", e)

# Read data from the serial port
try:
    num_bytes_to_read = 128
    max_timeout = 3000  # 3 seconds timeout
    print("\nReading data from serial port...")
    received_data = serial.read(serial_port, num_bytes_to_read, max_timeout)
    received_length = len(received_data)
    print("Received", received_length, "bytes from serial port.")
    if received_length > 0:
        print("Received Data:", received_data.decode("utf-8"))
    else:
        print("No data received within the timeout period.")
except Exception as e:
    print("Error reading from serial port:", e)
```

3.14. NTP/SNTP time synchronization

Retrieves the current time from an SNTP server as a UNIX timestamp for clock synchronization.

Functions

- `get_timestamp(netif, url)`
  Returns the current time from the SNTP server as a UNIX timestamp (UTC).

`sntp_client.get_timestamp(netif, url)`

Sends an SNTP request and returns the received UNIX timestamp.
Note:
The function will block (polling with a delay) until the network communication completes.

Parameters:

- **netif:** *integer*, the network interface ID of the Network Interface through which POST request shall be sent. Argument value must be one of constants `logger.NETIF_ETHERNET`, `logger.NETIF_MODEM_A`, `logger.NETIF_MODEM_B`
- **url:** *string*, URL of the SNTP server, mandatory

Return value:

- *integer*, actual time as fetched from NTP server, as UNIX timestamp in seconds. Returns value of 0 if error or timeout.

Raises: *None*

Example

Prerequisite: SDI-12 device is connected at SDI-12 port 0 and is independently supplied with power.

1. Define SNTP server details (Choose network interface and specify SNTP server).
2. Retrieve the current local system time before synchronization.
3. Request an accurate timestamp from an SNTP server.
4. Check if time was successfully retrieved, then update the system clock.
5. Retrieve the system time after synchronization to verify the update.

```python
import sntp_client
import logger   # Import logger to compare fetched time with system time & network interface
enumeration constants

# Step 1: Define SNTP server details
network_interface = logger.NETIF_ETHERNET   # Choose network interface (Ethernet), can also be
NETIF_MODEM_A or NETIF_MODEM_B
sntp_server = "pool.ntp.org:123"  # Public NTP server with port

# Step 2: Get the current system time before requesting SNTP
try:
    print("\nFetching the current system timestamp before requesting SNTP...")
    local_time_before_sync = logger.get_timestamp()
    print("Current System Time (Before Sync):", local_time_before_sync)
except Exception as e:
    print("Error getting system time:", e)

# Step 3: Fetch the accurate time from the SNTP server
try:
    print("\nRequesting current time from SNTP server...")
    fetched_timestamp = sntp_client.get_timestamp(network_interface, sntp_server)
    if fetched_timestamp == 0:
        print("Failed to fetch time from SNTP server.")
    else:
        print("Received UNIX timestamp from SNTP server:", fetched_timestamp)
except Exception as e:
    print("Error fetching SNTP time:", e)

# Step 4: Convert and print fetched time if successful
if fetched_timestamp != 0:
    try:
        print("\nSetting system clock using the received SNTP timestamp...")
        if logger.set_time_utc(fetched_timestamp):
            print("System time updated successfully.")
        else:
            print("Failed to update system time.")
    except Exception as e:
        print("Error updating system time:", e)

# Step 5: Get the system time after SNTP sync
try:
    print("\nFetching the system timestamp after SNTP sync...")
    local_time_after_sync = logger.get_timestamp()
    print("Current System Time (After Sync):", local_time_after_sync)
except Exception as e:
    print("Error getting system time:", e)

print("\nSNTP client example completed successfully!")
```

3.15. Storage

Provides functions to inspect storage devices and perform file operations (read, write, append, existence check, delete).

Functions

- **`get_info()`**
  Retrieves info about storage devices.
- **`read_file(path)`**
  Returns file contents as bytearray.
- **`write_file(path, data)`**
  Overwrites file contents with given data.
- **`append_file(path, data)`**
  Appends data to an existing file.
- **`file_exists(path)`**
  Checks if file exists at the specified path.
- **`delete_file(path)`**
  Removes a file at the given path.

`storage.get_info()`

Returns dictionary containing the information about available storage types

Parameters: *None*

Return value:

- *dictionary* containing the following elements:
  - **sdcard:** *None* if not available or dictionary containing the information about the storage type
    - **size:** total size
    - **free:** free size
    - **type:** filesystem type
    - **drive:** the name of the drive, sd:/ for sdcard and usb:/ for USB drive
  - **usb:** *None* if not available or dictionary containing the information about the storage type
    - **size:** total size
    - **free:** free size
    - **type:** filesystem type
    - **drive:** the name of the drive, sd:/ for sdcard and usb:/ for USB drive

Raises: *None*

`storage.read_file(path)`

Reads the whole file from storage into bytearray. Returns None on error (If the file cannot be opened or read).

Parameters:

- **path:** *string,* The path of the file to read, format "usb:/folder/file" or "sd:/folder/file"
- **data:** *bytearray* , object containing data to send

Return value:

- *None* on error OR
- *bytearray* A bytes object containing the file's content

Raises:

- *TypeError* if any argument is of incorrect type

`storage.write_file(path, data)`

Writes given data to a file on external storage. Overwrites the file if it exists.

Parameters:

- **path:** *string,* The path of the file to write, format "usb:/folder/file" or "sd:/folder/file"
- **data:** *string* or *bytes* or *bytearray* The data to write to the file.

Return value:

- *boolean,* returns `True` on success; `False` otherwise.

Raises:

- *TypeError* if any argument is of incorrect type

`storage.append_file(path, data)`

Appends given data to a file.

Parameters:

- **path:** *string*, The file path to which data will be appended, format “usb:/folder/file” or “sd:/folder/file”
- **data:** *string* or *bytes* or *bytearray* The data to write to the file.

Return value:

- *boolean*, returns `True` on success; `False` otherwise.

Raises:

- *TypeError* if any argument is of incorrect type

`storage.file_exists(path)`

Checks if a given file exists at the specified path.

Parameters:

- **path:** *string*, The file path to check, format “usb:/folder/file” or “sd:/folder/file”

Return value:

- *boolean*, `True` if the file exists; `False` otherwise.

Raises:

- *TypeError* if any argument is of incorrect type

`storage.delete_file(path)`

Deletes the file at the given path.

Parameters:

- **path:** *string*, The file path to delete, format “usb:/folder/file” or “sd:/folder/file”

Return value:

- *boolean*, `True` if the file was deleted; `False` otherwise.

Raises:

- *TypeError* if any argument is of incorrect type

Example

Prerequisite - an SD-card (of supported size and Filesystem) is inserted.

1. Check available storage devices (Get storage details).
2. Write data to a file on the SD card.
3. Read the file to verify its content.
4. Append data to the file.
5. Check if the file exists before proceeding.
6. Read updated content after appending.
7. Delete the file from storage.
8. Verify deletion to ensure the file was removed.

```
import storage

# Step 1: Get information about available storage devices
try:
    print("\nChecking available storage devices...")
    storage_info = storage.get_info()
    print("Storage Information:", storage_info)
except Exception as e:
    print("Error getting storage info:", e)

# Step 2: Define file paths for operations
file_path_sd = "sd:/example_file.txt"
file_path_usb = "usb:/example_file.txt"

if storage_info.get("sdcard") is not None:
    # Step 3: Write data to a file on SD card
    try:
        print("\nWriting data to file on SD card...")
        write_success = storage.write_file(file_path_sd, "This is a test file.\n")
        if write_success:
            print("File written successfully to SD card.")
        else:
            print("Failed to write file to SD card.")
    except Exception as e:
        print("Error writing file:", e)

    # Step 4: Read file contents from SD card
    try:
        print("\nReading data from SD card file...")
        file_content = storage.read_file(file_path_sd)
        if file_content is not None:
            print("File Read Successfully. Content:")
            print(file_content.decode())  # Decoding bytes to string
        else:
            print("Failed to read file.")
    except Exception as e:
        print("Error reading file:", e)

    # Step 5: Append data to the file on SD card
    try:
        print("\nAppending data to file on SD card...")
        append_success = storage.append_file(file_path_sd, "Appending new data.\n")
        if append_success:
            print("Data appended successfully.")
        else:
            print("Failed to append data.")
    except Exception as e:
        print("Error appending file:", e)

    # Step 6: Check if the file exists on SD card
    try:
        print("\nChecking if the file exists on SD card...")
        file_exists = storage.file_exists(file_path_sd)
        if file_exists:
            print("The file exists.")
        else:
            print("The file does not exist.")
    except Exception as e:
        print("Error checking file existence:", e)

    # Step 7: Read the updated file to verify append operation
    try:
        print("\nReading updated data from SD card file...")
        updated_file_content = storage.read_file(file_path_sd)
        if updated_file_content is not None:
            print("Updated File Content:")
            print(updated_file_content.decode())  # Decoding bytes to string
        else:
            print("Failed to read updated file.")
    except Exception as e:
        print("Error reading updated file:", e)

    # Step 8: Delete the file from SD card
    try:
        print("\nDeleting the file from SD card...")
        delete_success = storage.delete_file(file_path_sd)
        if delete_success:
            print("File deleted successfully.")
        else:
            print("Failed to delete the file.")
    except Exception as e:
        print("Error deleting file:", e)

    # Step 9: Verify file deletion
    try:
        print("\nVerifying file deletion...")
        file_still_exists = storage.file_exists(file_path_sd)
        if not file_still_exists:
             print("The file does not exist.")
        else:
            print("File still exists.")
    except Exception as e:
        print("Error checking file existence after deletion:", e)

print("\nStorage module example completed successfully!")
```

3.16. UDP client

Provides functionality to send UDP datagrams to a specified server using a selected network interface. It is designed for lightweight, connectionless communication.

Functions

- `send()`
  Sends UDP datagram to the specified server.

`udp.send(netif, server, port, data)`

Sends a UDP datagram to the specified server over the given network interface.

Parameters:

- **netif:** *string*, The ID of the network interface to be used for sending the UDP datagram. Argument value must be one of constants `logger.NETIF_ETHERNET`, `logger.NETIF_MODEM_A`, `logger.NETIF_MODEM_B`
- **server:** *string*, hostname or IP address of the UDP server, mandatory
- **port:** *integer*, port number for the UDP connection.
- **data:** *string* or *bytes*, data to be sent to the server over UDP.

Return value:

- *None* on internal error or network issues occurs
- *boolean*, `True` if the datagram was sent successfully; `False` otherwise

Raises:

- *TypeError* if any argument is of incorrect type

Example

1. Define connection parameters and data contents
2. Send UDP datagram

```
import udp
import logger

# Define network interface, server, port, and message
netif = logger.NETIF_ETHERNET
server = "example.com"
port = 12345
message = "Hello, UDP server!"

try:
    # Send UDP datagram
    result = udp.send(netif, server, port, message)

    # Check the result
    if result is True:
        print("UDP datagram sent successfully!")
    elif result is False:
        print("Failed to send UDP datagram.")
    # Handles None case explicitly
    else:
        print("An internal error occurred.")
except Exception as e:
    print("An error occurred:", e)
```