NxBCI API Documentation

This document provides a comprehensive overview of the API for interacting with the EEG/EMG Device, including various modules and their functionalities.

Modules

- `BluetoothController`
- `MQTT_Receiver`
- `Relay`
- `Replay`
- `TCP_Receiver`

`BluetoothController`

A controller for managing Bluetooth Low Energy (BLE) device connections and data exchange. This document details how to use the class to interact with a BLE device.

Contents

Overview & Quick Start

The `BluetoothController` class encapsulates device scanning, connection, data I/O, and notification handling.

Core Features

- Automatically scans for and connects to a specified BLE device.
- Reads essential device information like battery level and TF card storage.
- Receives and parses real-time pose data (accelerometer, gyroscope, thermometer).
- Configures device work modes (TF Card Storage, TCP, MQTT).
- Sets device parameters like gain and sample rate.

Quick Start Example

> Using an `async with` statement is the recommended way to manage the controller's lifecycle, as it handles initialization and cleanup automatically.

```python
import asyncio
import logging
from NxBCI.BluetoothController import BluetoothController

async def main():
    # Instantiate the controller with the target device name
    controller = BluetoothController(device_name="BLE_FOR_EEG")

    try:
        # 'async with' automatically calls initialize() and close()
        async with controller:
            if not controller.bt_GetConnectionStatus():
                logging.error("Failed to initialize controller.")
                return

            # Get device gain
            gain = controller.bt_GetGain()
            print(f"Device Gain: {gain}")

            # Get battery level
            battery = controller.bt_GetBatteryLevel()
            print(f"Battery Level: {battery}%")

            # Configure the pose data sample rate (how many data points to buffer)
            controller.pose_Config(Sample_rate=50) # Buffer 50 samples

            # Loop to get data
            for _ in range(10): # Example: loop 10 times
                # Get the latest pose data
                pose_data = controller.pose_GetData()
                if pose_data:
                    # pose_data is a list of 7 deques for
                    # Accel(X,Y,Z), Temp, Gyro(X,Y,Z)
                    # Print the latest data point (index -1)
                    print(f"Accelerometer: (X:{pose_data[0][-1]:.2f},
                                            Y:{pose_data[1][-1]:.2f},
                                            Z:{pose_data[2][-1]:.2f})")
                    print(f"Temperature: {pose_data[3][-1]:.2f}°C")
                    print(f"Gyroscope: (X:{pose_data[4][-1]:.2f},
                                            Y:{pose_data[5][-1]:.2f},
                                            Z:{pose_data[6][-1]:.2f})")
                await asyncio.sleep(1)

    except Exception as e:
        logging.error(f"An error occurred: {e}")

if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO, format='%(levelname)s: %(message)s')
    asyncio.run(main())
```

Connection Management

Interfaces for managing the BLE device connection lifecycle.

`__init__(self, device_name: str = "BLE_FOR_EEG")`

Initializes a new `BluetoothController` instance.

- **Parameters**:
  - `device_name` (`str`): The name of the target BLE device to connect to. Defaults to `"BLE_FOR_EEG"`.

`async def initialize(self) -> None`

Initializes the controller by scanning for, connecting to, and fetching initial data from the device. This is called automatically when entering an `async with` block.

`async def close(self) -> None`

Disconnects from the BLE device and cleans up resources. This is called automatically when exiting an `async with` block.

`async def bt_ReconnectBluetooth(self)`

Manually attempts to reconnect to the BLE device. This is useful if the connection was lost. If the device is already connected, this method does nothing.

`async def bt_SetBluetoothTarget(self, target: str)`

Switches the connection to a new target BLE device. This will disconnect from the current device (if any) and then initiate a new connection to the specified target.

- **Parameters**:
  - `target` (`str`): The name of the new target device.

Device Information

Methods to retrieve status and configuration information from the device.

`def bt_GetConnectionStatus(self) -> bool`

Checks the current connection status of the BLE device.

- **Returns**: `bool` - `True` if connected, `False` otherwise.

`def bt_GetDeviceName(self) -> str`

Gets the name of the currently connected BLE device.

- **Returns**: `str` - The device name.

`def bt_GetDeviceAddress(self) -> str`

Gets the MAC address of the currently connected BLE device.

- **Returns**: `str` - The device's MAC address.

`def bt_GetBatteryLevel(self) -> int`

Gets the device's current battery level as a percentage.

- **Returns**: `int` - The battery level percentage (0-100).

`def bt_GetGain(self) -> int`

Gets the device's current gain configuration.

- **Returns**: `int` - Returns `100` or `1000`. Returns `0` if the gain has not yet been fetched from the device.

`async def bt_GetSampleRate(self) -> int`

Gets the device's main data acquisition sample rate.

- **Returns**: `int` - The current sample rate in Hz (e.g., 500, 1000, 2000, 4000).

Pose & Temperature Data

Interfaces for configuring and retrieving pose and temperature data from the MPU sensor.

`def pose_Config(self, Sample_rate: int)`

Sets the internal sample rate and buffer size for pose and temperature data. This determines the length of the deques returned by `pose_GetData`.

- **Parameters**:
  - `Sample_rate` (`int`): The desired number of samples to buffer. The valid range is (0, 100].

`def pose_GetData(self) -> Optional[List[deque]]`

Retrieves the buffered list of latest pose and temperature data.

- **Returns**: `Optional[List[deque]]` - A list of 7 `deque` objects. Returns `None` if the Bluetooth is not connected or the buffer is not yet full.
    - **Deque Order**:
        a. `deque[0]` : X-axis acceleration (g)
        b. `deque[1]` : Y-axis acceleration (g)
        c. `deque[2]` : Z-axis acceleration (g)
        d. `deque[3]` : Temperature (°C)
        e. `deque[4]` : X-axis angular velocity (°)
        f. `deque[5]` : Y-axis angular velocity (°)
        g. `deque[6]` : Z-axis angular velocity (°)

Device Configuration

Asynchronous methods for writing new configurations to the device.

`async def bt_SetGain(self, gain: int)`

Sets the device's gain value.

- **Parameters**:
    - `gain` (`int`): The gain value. **Must** be `100` or `1000` .

`async def bt_SetSampleRate(self, sampleRate: int)`

Sets the device's main data acquisition sample rate.

- **Parameters**:
    - `sampleRate` (`int`): The sample rate in Hz. **Must** be one of: `500` , `1000` , `2000` , `4000` .
- **Raises**:
    - `ValueError` : If an unsupported `sampleRate` is provided.

`async def bt_SetTFcardStorageMode(self)`

Sets the device's work mode to "TF Card Storage". The device will store collected data directly on its TF card.

`async def bt_SetTCPMode(self, wifi_name: str, password: str)`

Sets the device's work mode to "TCP Transport Mode", sending data over Wi-Fi.

- **Parameters**:
  - `wifi_name` (`str`): The Wi-Fi network name (SSID) for the device to connect to.
  - `password` (`str`): The Wi-Fi network password.

`async def bt_SetMQTTMode(self, MQTT_URI: str, port: int)`

Sets the device's work mode to "MQTT Transport Mode".

- **Parameters**:
  - `MQTT_URI` (`str`): The address of the MQTT broker.
  - `port` (`int`): The port of the MQTT broker.

`MQTT_Receiver`

A thread-safe class to receive and process data via the MQTT protocol. It handles the connection lifecycle, subscription, and data processing in a background thread, providing a simple and robust interface.

Table of Contents

Overview & Quick Start

This class leverages the `paho-mqtt` library to manage a persistent connection to an MQTT broker. It automatically handles reconnections in its own network thread, allowing your main application to remain focused on its primary tasks.

Quick Start Example

> This example demonstrates the standard workflow: initialize, start, poll for data in a loop, and finally, stop the client gracefully.

```python
import logging
import time
from NxBCI.MQTT_Receiver import MQTT_Receiver

# Configure logging to see status messages from the receiver
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s - %(levelname)s - %(message)s')

# 1. Initialize the receiver (this does not connect yet)
receiver = MQTT_Receiver(Ip="MQTT broker IP", port=1883, topic="esp32-pub-message")

# 2. Start the connection and background network loop
receiver.start()

try:
    while True:
        time.sleep(2)
        # 3. Safely check the connection status
        if receiver.is_connected():
            # 4. Safely get a copy of the latest EEG data
            data = receiver.get_data()
            if data and data:
                print(f"MQTT Connected. Last value on Ch0: {list(data)[-1]:.4f} mV")
            else:
                print("MQTT Connected, but no data received yet.")
        else:
            # The Paho-MQTT client handles reconnections automatically
            print("MQTT Disconnected. Waiting for automatic reconnect...")

except KeyboardInterrupt:
    print("Program interrupted by user.")
finally:
    # 5. Stop the client and disconnect cleanly
    print("Stopping MQTT receiver...")
    receiver.stop()
    print("Receiver stopped.")
```

Lifecycle Methods

These methods control the operational state of the MQTT client.

`__init__(self, channels: int = 16, sample_rate: int = 500, duratio`

Initializes the `MQTT_Receiver`'s configuration. This method only sets up the object's state and does not initiate any network activity.

- **Parameters**:
  - `channels` (`int`): The number of EEG data channels to process. Defaults to `16`.
  - `sample_rate` (`int`): The sampling rate (Hz) of the data, used to calculate buffer size. Defaults to `500`.
  - `duration` (`int`): The duration in seconds of data to store in each channel's buffer. Defaults to `4`.
  - `Ip` (`str`): The IP address or hostname of the MQTT broker.
  - `port` (`int`): The network port of the MQTT broker. Defaults to `1883`.
  - `topic` (`str`): The MQTT topic to subscribe to for data. Defaults to `esp32-pub-message`.

`start(self)`

Connects to the MQTT broker and starts the background network loop (`loop_start`). The loop runs in a separate thread, handling network traffic, callbacks, and automatic reconnections. This method is non-blocking.

`stop(self)`

Stops the network loop and disconnects gracefully from the MQTT broker. This ensures a clean shutdown.

Status & Data Access

These methods provide thread-safe access to the receiver's current state and the data it has collected.

`is_connected(self) -> bool`

Checks if the client is currently connected to the MQTT broker.

- **Returns**: `bool` - `True` if the client is connected, `False` otherwise.

`get_data(self) -> List[Deque[float]]`

Returns a shallow copy of all channel data queues. Returning a copy prevents potential thread-safety issues if your application iterates over the data while the network thread is simultaneously modifying it.

- **Returns**: `List[Deque[float]]` - A list of deques. Each deque contains the buffered data for a single channel, converted to millivolts (mV).

`def pose_GetData(self) -> Optional[List[deque]]`

Retrieves the buffered list of latest pose and temperature data.

- **Returns**: `Optional[List[deque]]` - A list of 7 `deque` objects. Returns `None` if the MQTT connection is not connected or the mpu buffer is empty .
    - **Deque Order**:
        a. `deque[0]`: X-axis acceleration (g)
        b. `deque[1]`: Y-axis acceleration (g)
        c. `deque[2]`: Z-axis acceleration (g)
        d. `deque[3]`: Temperature (°C)
        e. `deque[4]`: X-axis angular velocity (°)
        f. `deque[5]`: Y-axis angular velocity (°)
        g. `deque[6]`: Z-axis angular velocity (°)

`Relay`

A class for relaying data to a cloud-based MQTT broker. It handles connection and data publishing with a simple interface.

Table of Contents

Overview & Quick Start

The `Relay` class is designed to send data to a remote MQTT broker. Upon initialization, it immediately attempts to connect to the specified broker and starts a background network loop. The main interaction with the class involves creating an instance and then calling the `relay_data()` method whenever you need to publish a message.

Quick Start Example

> This example demonstrates how to set up the relay and publish data in a loop.

```python
import logging
import time
import json
from NxBCI.Ralay_EMQX import Relay # Replace with the actual module name

# Configure logging to see status messages
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

# 1. Initialize the Relay. It will attempt to connect immediately.
#    Replace with your actual broker details and credentials.
cloud_relay = Relay(
    cloud_broker_address="your_broker_address.com",
    cloud_port=1883,
    cloud_topic="your/topic",
    client_id="local_relay_client_01",
    username="your_username",
    password="your_password"
)

# Give the client a moment to establish the connection
time.sleep(2)

try:
    for i in range(10):
        # 2. Check the connection status before attempting to send data
        if cloud_relay.isConnected:
            # 3. Relay data
            payload = {"message_id": i, "value": "some_data"}
            print(f"Relaying message: {json.dumps(payload)}")
            cloud_relay.relay_data(json.dumps(payload))
        else:
            print("Relay is not connected. Waiting for connection...")

        time.sleep(1)

finally:
    # Although the class doesn't have a stop() method, this shows a complete lifecycle.
    # The background thread is a daemon, so it will exit when the main program does.
    print("Program finished.")
```

Core Methods

`__init__(self, cloud_broker_address='', cloud_port=1883, cloud_top`

Initializes the `Relay` object and **immediately attempts to connect** to the MQTT broker in the background.

- **Note**: This constructor starts a background network thread (`loop_start`). You do not need to call a separate `start` or `connect` method.
- **Parameters**:
  - `cloud_broker_address` (`str`): The IP address or hostname of the cloud MQTT broker.
  - `cloud_port` (`int`): The network port of the broker. Defaults to `1883` . If set to `8883` , TLS/SSL is automatically enabled.
  - `cloud_topic` (`str`): The default MQTT topic to publish data to.
  - `client_id` (`str`): A unique client ID for the MQTT connection.
  - `username` (`str`): The username for authenticating with the broker.
  - `password` (`str`): The password for authenticating with the broker.
- **Logs**:
  - Logs an error if the connection is refused or another exception occurs during initialization.

`relay_data(self, data)`

Publishes a message to the MQTT topic specified during initialization.

- **Parameters**:
  - `data` (`str` or `bytes`): The data payload to publish. The method internally converts it to a string (`str(data)`) before sending.
- **Logs**:
  - Logs an error message if publishing the message fails (i.e., the status code from the publish call is not 0).

Attributes & Status

`isConnected`

A boolean flag indicating the current connection status to the MQTT broker.

- **Type**: `bool`
- **Details**: This flag is set to `True` within the `on_connect` callback when a successful connection is established and `False` on failure or disconnection.

Configuration Attributes

The parameters passed to the `__init__` method are stored as public attributes for reference:

- `cloud_broker_address` (`str`)
- `cloud_port` (`int`)
- `cloud_topic` (`str`)
- `client_id` (`str`)
- `username` (`str`)
- `password` (`str`)
- `client` (`mqtt_client.Client`): The underlying Paho-MQTT client instance.

`Replay`

A thread-safe class for replaying data from a binary file to simulate a live data feed. It can also be used to load entire datasets into memory for analysis.

Table of Contents

Overview & Modes of Operation

This class is designed to work in two distinct modes, making it a versatile tool for both testing real-time applications and performing offline data analysis.

1. **Playback Mode**: A background thread reads data samples from the file and "plays" them into a buffer at a specified sample rate. This perfectly simulates a live data source like a `TCP_Receiver` or `MQTT_Receiver`, making it ideal for testing data processing and visualization pipelines without a physical device.
2. **Load Mode**: The entire binary file is read and parsed into memory. This allows for quick, random access to any part of the dataset, which is useful for analysis, feature extraction, or training machine learning models.

Quick Start (Playback Mode)

> Using the class as a context manager (`with Replay(...) as ...`) is strongly recommended to ensure that file resources are properly managed and closed.

```python
import time
import logging
from NxBCI.Replay import Replay
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(message)s')

# Use 'with' to ensure the file is closed automatically
with Replay(FilePath='YOUR_DATA.BIN', sample_rate=500) as replayer:
    # Start the background playback thread
    replayer.start_playback()

    for _ in range(5):
        time.sleep(1)
        # Get the current playback buffer (last ~4 seconds of data)
        data_buffer = replayer.get_data()
        if data_buffer and data_buffer:
            print(f"Playback is running. Buffer for Ch0 has {len(data_buffer)} samples.")

with Replay(FilePath='YOUR_DATA.BIN') as replayer:
    # Load the entire file into memory (this is a blocking call)
    replayer.load_all_data()

    print(f"File loaded. Total samples: {replayer.get_total_samples()}")

    # Get a specific segment of the data
    segment = replayer.get_segment(start_sample=1000, end_sample=2000)
    print(f"Retrieved a segment with {len(segment)} samples for Channel 0.")
```

Lifecycle & Context Manager

`__init__(self, FilePath: str, channels: int = 16, sample_rate: int`

Initializes the Replay object. This is a lightweight operation that only sets configuration and does not perform file I/O.

- **Parameters**:

    - `FilePath` (`str`): The path to the binary data file.
    - `channels` (`int`): The number of data channels recorded in the file. Defaults to `16`.
    - `sample_rate` (`int`): The playback speed in samples per second (Hz). Defaults to `500`.
    - `duration` (`int`): For playback mode, the duration in seconds of data to keep in the live buffer. Defaults to `4`.
    - `isLoop` (`bool`): If `True`, playback will loop continuously from the beginning after reaching the end. Defaults to `False`.
- **Raises**:
    - `FileNotFoundError`: If the `FilePath` does not exist.

`__enter__(self)` / `__exit__(self, ...)`

Implements the context manager protocol. The `__enter__` method opens the file and memory-maps it. The `__exit__` method ensures that the `stop()` method is called and all file resources are closed, even if errors occur.

Playback Control

These methods control the background playback thread. They automatically trigger `load_all_data()` if the data isn't already in memory.

`start_playback(self)`

An alias for `restart_playback()`. Starts or restarts the playback from the beginning of the file.

`restart_playback(self)`

Stops any active playback and restarts it from the very beginning of the file (sample 0).

`play_from(self, start_sample: int)`

Starts playback from a specific sample index and continues until the end of the file (or loops if `isLoop` is `True`).

- **Parameters**:
    - `start_sample` (`int`): The sample index (zero-based) to begin playing from.

`play_segment(self, start_sample: int, end_sample: int)`

Plays only a specific segment of the data file, then stops automatically. The `isLoop` setting is ignored.

- **Parameters**:
    - `start_sample` (`int`): The starting sample index of the segment.
    - `end_sample` (`int`): The ending sample index of the segment (exclusive).

`stop(self)`

Stops the background playback thread if it is running. This is called automatically when exiting a `with` block.

Data Loading (Load Mode)

`load_all_data(self)`

Loads the entire file into an in-memory cache. This is a blocking, one-time operation. It is required before using methods like `get_full_dataset()` or `get_segment()`.

Data Retrieval

`get_data(self) -> List[Deque[float]]`

For Playback Mode. Returns a copy of the current playback data buffers (deques). This is the primary method for accessing data during a real-time playback simulation.

- **Returns**: `List[Deque[float]]` - A list of deques, one for each channel, containing the most recent data samples.

`get_full_dataset(self) -> List[List[float]]`

For Load Mode. Returns the entire dataset as a list of lists. Requires `load_all_data()` to have been called.

- **Returns**: `List[List[float]]` - A list where each inner list contains all samples for a single channel.

`get_segment(self, start_sample: int, end_sample: int) -> List[List`

For Load Mode. Returns a specific slice of the dataset. Requires `load_all_data()` to have been called.

- **Returns**: `List[List[float]]` - A list containing the requested segment for each channel.

`get_total_samples(self) -> int`

For Load Mode. Returns the total number of samples detected in the file. Requires `load_all_data()` to have been called.

`def pose_GetData(self) -> Optional[List[deque]]`

Retrieves the buffered list of latest pose and temperature data.

- **Returns**: `Optional[List[deque]]` - A list of 7 `deque` objects. Returns `None` if the playback thread is not running or the MPU buffer is empty .
  - **Deque Order**:
    a. `deque[0]` : X-axis acceleration (g)
    b. `deque[1]` : Y-axis acceleration (g)
    c. `deque[2]` : Z-axis acceleration (g)
    d. `deque[3]` : Temperature (°C)
    e. `deque[4]` : X-axis angular velocity (°)
    f. `deque[5]` : Y-axis angular velocity (°)
    g. `deque[6]` : Z-axis angular velocity (°)

Status Checks

`is_running(self) -> bool`

Checks if the playback thread is currently active and running.

- **Returns**: `bool` - `True` if playback is in progress, `False` otherwise.

`TCP_Receiver`

A thread-safe TCP receiver designed for continuously acquiring data streams from a device. It uses a background worker thread to automatically handle connections, data reception, and reconnections.

Table of Contents

Overview & Quick Start

This class provides a robust solution for handling TCP data streams in a separate thread, ensuring your main application remains responsive. All public methods are thread-safe.

Quick Start Example

> The following example demonstrates how to initialize, start, and interact with the `TCP_Receiver` .

```python
import logging
import time
from NxBCI.TCP_Receiver import TCP_Receiver # Replace with the actual module name

# It's recommended to configure logging in your main script
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

# 1. Initialize the receiver with target IP and parameters
receiver = TCP_Receiver(channels=16, sample_rate=500, Ip="192.168.4.1", port=8080)

# 2. Start the background worker thread
receiver.start()

try:
    while True:
        time.sleep(2)
        # 3. Safely check the connection status
        if receiver.is_connected():
            # 4. Safely get a copy of the latest data
            latest_data_queues = receiver.get_data()
            if latest_data_queues and latest_data_queues:
                # Access the latest value from the first channel's queue
                print(f"Connection is ON. Channel 0 latest value: {list(latest_data_queues)[-1]}")
            else:
                print("Connection is ON, but no data has been received yet.")
        else:
            print("Connection is OFF. Waiting for the receiver to reconnect...")

except KeyboardInterrupt:
    print("Program interrupted by user.")
finally:
    # 5. Stop the receiver to clean up resources
    print("Stopping receiver service...")
    receiver.stop()
    print("Receiver service stopped.")
```

Lifecycle Methods

These methods control the operational state of the receiver.

`__init__(self, channels: int = 16, sample_rate: int = 500, duratio`

Initializes the TCP receiver's configuration. This method only sets up the object's state and does not initiate any network activity.

- **Parameters**:
  - `channels` (`int`): The number of data channels to expect. Defaults to `16`.
  - `sample_rate` (`int`): The sample rates of the received data. Defaults to `500`.
  - `duration` (`int`): The duration in seconds of data to store in the buffer for each channel. Defaults to `4`.
  - `Ip` (`str`): The IP address of the target device. Defaults to `"192.168.4.1"`.
  - `port` (`int`): The TCP port on the target device. Defaults to `8080`.

`start(self)`

Starts the background worker thread. The thread manages connection, data reception, and reconnection logic.

`stop(self)`

Stops the background worker thread and cleans up all resources, including the network socket. This method is thread-safe and blocks until the worker thread has terminated.

Status & Data Access

These methods provide thread-safe access to the receiver's current state and the data it has collected.

`is_connected(self) -> bool`

Checks if the client is currently connected to the server.

- **Returns**: `bool` - `True` if a connection is established, `False` otherwise.

`get_data(self) -> List[Deque[float]]`

Returns a shallow copy of all channel data queues.

- **Returns**: `List[Deque[float]]` - A list of deques. Each deque contains the buffered data for a single channel, converted to millivolts (mV).