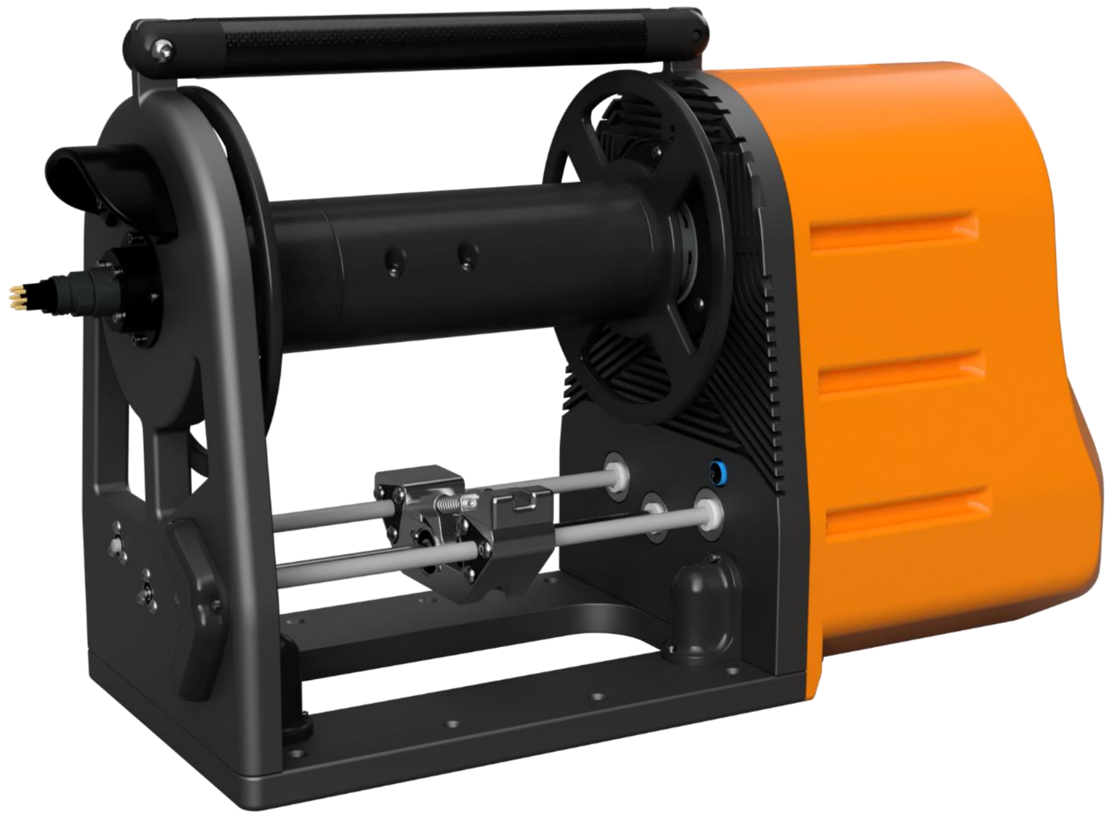
TMS (UNI-WINCH MINI)

Software Overview



Hybrid Robotics
REV1.3

Created by: Aaron Bottke Month: 12/2023

Table of Contents

Marine Tether Management Winch System (Software Overview)

The marine tether management winch system represents an advanced integration of data processing, mechanical control, and communication interfacing. Comprised of three vital components - CanBusListener.py, ServoDriver.py, and Server.py - these scripts together create a dynamic system proficient in overseeing winch operations while ensuring adaptability and real-time responsiveness.

Overview of Key Components:

1. **CanBusListener.py**
   - *Function:* This script serves as the system's primary data acquisition and processing unit, dedicated to interpreting telemetry data from motor controllers. It also sets up all UltraDict shared memory spaces and launches the other two.
   - *Role:* It acts as a crucial data hub, offering real-time monitoring and essential insights critical for maintaining optimal system performance.
2. **ServoDriver.py**
   - *Function:* Responsible for translating digital commands into precise mechanical actions, this script manages the mechanical actions vital for the winch's operation.
   - *Role:* It plays a pivotal role in the physical execution of operations, dynamically adapting to changes in system conditions and control commands.
3. **Server.py**
   - *Function:* This component facilitates external communication through a series of API endpoints, using FastAPI for efficient data handling and response.
   - *Role:* It functions as the central communication interface, enabling real-time data access and control for effective system monitoring and operational control.

Basic Linux Commands:

Remember that Linux commands are case-sensitive, so ensure you use the correct capitalization when working with files and directories. These basic commands and folder navigation techniques will help you navigate and perform essential tasks in the TMS’s Linux environment.

1. **`pwd` (Print Working Directory):** Displays the current directory or folder path you are in.
2. **`ls` (List):** Lists the files and directories in the current directory.
   - Use `ls -l` for a detailed listing.
   - Use `ls -a` to show hidden files (those starting with a dot `.`).
3. **`cd` (Change Directory):** Allows you to navigate to a different directory.
   - Use `cd directory_name` to enter a specific directory.
   - Use `cd ..` to move up one directory level.
   - Use `cd` without an argument to go to your home directory.
4. **`mkdir` (Make Directory):** Creates a new directory in the current location.
   - Example: `mkdir new_directory_name`.
5. **`touch`:** Creates an empty file.
   - Example: `touch file_name.txt`.
6. **`rm` (Remove):** Deletes files or directories.
   - Use `rm file_name` to delete a file.
   - Use `rm -r directory_name` to delete a directory and its contents.
7. **`cp` (Copy):** Copies files or directories.
   - Example: `cp source_file destination`.
8. **`mv` (Move):** Moves or renames files or directories.
   - Example: `mv old_file new_file` (rename).
   - Example: `mv file_name directory_name` (move).
9. **`cat` (Concatenate):** Displays the content of a file.
   - Example: `cat file_name.txt`.
10. **`more` and `less`:** Display file content one page at a time, allowing scrolling.
    - Example: `more file_name.txt` or `less file_name.txt`.

Folder Navigation:

1. **`cd` (Change Directory):** Use this command followed by the directory path to navigate to a specific folder.
2. **`.` (Dot):** Represents the current directory. For example, `./script.sh` runs a script in the current directory.
3. **`..` (Double Dot):** Represents the parent directory. For example, `cd ..` moves up one level in the directory tree.
4. **`~` (Tilde):** Represents the user's home directory. For example, `cd ~` takes you to your home directory.
5. **`/` (Forward Slash):** Represents the root directory. For example, `/var/log` represents the log directory at the root of the file system.
6. **Tab Completion:** You can use the Tab key to auto-complete directory and file names. This is useful for navigating quickly and avoiding typing errors.
7. **`ls` (List):** Use `ls` to list the contents of the current directory. This helps you see what's available for navigation.

Integrated System Architecture:

Each component contributes significantly to the system's functionality. CanBusListener.py provides foundational data necessary for operational insights and control decisions. ServoDriver.py utilizes this data to execute precise mechanical responses, ensuring that the system's physical operations align with control commands and data inputs. Server.py bridges the internal functionality with external control interfaces, providing a user-friendly and responsive system monitoring and control platform. The web-based GUI that is included with the TMS utilizes the same API that is available for custom control implementation..

Using Detached Screen Windows for Process Management:

To streamline the operation of the marine tether management winch system, the components are launched in detached screen windows. This approach allows for processes to run in the background while maintaining easy access and management when needed.

Launching the Processes:

The initiation of CanBusListener.py, ServoDriver.py, and Server.py components in detached screen sessions is accomplished through the following bash commands:

Launching Server.py:

bash
```
screen -dm -S TMS_Server python3 /home/pi/SC25/Server.py
```

Launching ServoDriver.py:

bash
```
screen -dm -S TMS_ServoDriver python3 /home/pi/SC25/ServoDriver.py
```

Launching CanBusListener.py:

bash
```
screen -dm -S TMS_CanBusListener python3 /home/pi/SC25/CanBusListener.py
```

Managing Screen Sessions:

When working with the marine tether management winch system's components in detached screen sessions, it's crucial to master various tasks such as attaching, detaching, terminating, and entering copy mode. These commands are essential for effectively controlling and monitoring the processes running within these sessions:

Attaching to a Screen Session:

- To access and interact with a specific detached screen session associated with a particular process, use the `screen -r session_name` command.
- Replace `session_name` with the actual name of the session you want to attach to.
- Attaching to a session brings you into the interactive environment where the process is running. This allows you to view its output, send commands, and monitor its status in real-time.

Detaching from a Screen Session:

- If you are already inside a screen session and wish to detach from it without terminating the underlying process, use the following keyboard shortcut:
  - Press `Ctrl` + `A` followed by `D` (Ctrl+A, D).
- Detaching allows you to return to your terminal or shell while keeping the process running in the background within the screen session.

Terminating a Screen Session:

- To close a screen session and terminate its associated process, attach to the session if you are not already attached.
- Inside the session, stop the process using an appropriate command (e.g., using `Ctrl` + `C` for programs that respond to interruption) or by executing the relevant termination command.
- After stopping the process, exit the screen session by typing `exit` or by pressing `Ctrl` + `D`. This effectively closes the session and frees up system resources.

Terminating a Screen Session (Alternative Method):

```
bash
screen -X -S session_name quit
```

- Replace `session_name` with the actual name of the screen session you want to terminate. This command instructs the screen session to exit and terminates both the session and the associated process without the need to attach to it.

Entering Copy Mode (Scrollback):

- Screen sessions offer a useful feature called copy mode, allowing you to review and scroll through the session's history, including past terminal output.
- To enter copy mode, press `Ctrl` + `A` followed by `[` (Ctrl+A, `[`).
- Navigate through the session's content using the arrow keys or other navigation keys on your keyboard.
- To exit copy mode and return to the regular session, press `Q`.

CanBusListener.py

Overview

CanBusListener.py is a pivotal component in the marine tether management winch system, dedicated to handling and interpreting telemetry data from motor controllers. It acts as the system's sensory organ, capturing real-time operational data. It is critical that both. Servo driver and Server are launched after CanBusListener is; if CanBusListener is closed, you must close both ServoDriver and Server and restart everything.

Core Functionalities

1. **Telemetry Data Acquisition**: It continuously listens to the CAN bus for telemetry data transmitted by motor controllers. This data includes operational parameters, status reports, and any alerts or errors.
2. **Data Interpretation**: Upon receiving the data, CanBusListener.py interprets these messages, translating raw binary or coded data into a structured and understandable format. This involves decoding various parameters like speed, torque, temperature, and other critical operational metrics.
3. **UltraDict Integration**: For efficient data management and sharing, CanBusListener.py utilizes an UltraDict. This advanced data structure is optimized for high-speed data storage and retrieval, facilitating quick access by other system components.

Key Roles in the System

- **Data Hub**: Serves as the central repository for all telemetry data in the system. Its effective functioning is critical for the accurate monitoring and control of the marine tether management system.
- **Real-Time Monitoring**: Provides real-time insights into the operational status of the motor controllers, enabling proactive maintenance and immediate response to any anomalies.

Integration with Other Components

- **Dependency**: Both ServoDriver.py and Server.py depend on CanBusListener.py for their operation. This dependency underscores its vital role in the overall system.
- **Data Provider to ServoDriver.py**: Supplies processed telemetry data to ServoDriver.py, which uses this information to make informed decisions about servo movements and adjustments.
- **Backend Support for Server.py**: Acts as a data backend for Server.py, which manages the system's API. The API endpoints provided by Server.py might offer access to telemetry data processed by CanBusListener.py for external monitoring and control.

System Architecture and Data Flow

- **Central Node in Data Flow**: CanBusListener.py is at the heart of the system's data flow architecture. It not only gathers and processes data but also ensures that this data is readily available for other processes.
- **Concurrent Operations and Efficiency**: The script is designed to operate concurrently with other system processes, maintaining efficiency and ensuring that data latency is minimized.

Conclusion

CanBusListener.py is more than just a data processing script; it is integral to the functionality and efficiency of the marine tether management winch system. Its ability to accurately interpret and disseminate telemetry data ensures that the system operates smoothly and effectively, making it indispensable in the overall architecture of the system.

ServoDriver.py

Overview

ServoDriver.py is a critical component in the marine tether management winch system, focusing on the precise execution of mechanical movements based on command inputs and processed data. This script acts as the primary interface between the digital control commands and the system's mechanical actions, translating software instructions into physical movements.

Core Functionalities

 Governor Processes: Executes precise mechanical movements by controlling servo mechanisms according to digital command inputs.

- **Leadscrew Governor**: Manages the leadscrew motor for essential linear movement control within the system.
- **Tensioner Governor**: Controls the tensioner motor to maintain the correct tension on the tether.
- **Primary Motor Governor**: Operates the primary motor, crucial for the tether's winding and unwinding actions. Primary Motor Governor also disengages the brake when necessary.

 Real-Time Tether Parameter Adjustment (TetherParameterAdjust): Dynamically updates the start and stop points for tether layers on the drum. This real-time adjustment is vital for adapting the system's movements to the actual operational conditions, ensuring precision and efficiency in the tether management.

 Brake Process: Critical for ensuring the safety and operational integrity of the system. The Brake Process is responsible for engaging the brakes in response to system commands or safety protocols. This functionality is crucial in scenarios where immediate halting or controlled movements of the winch system are required, providing an additional layer of safety and precision in operations.

Key Roles in the System

- Physical Operation Executor: Serves as the key driver of physical operations within the tether management system, translating digital instructions into tangible actions.
- Dynamic Response Mechanism: Offers real-time adaptive control, essential for maintaining accurate operations in the ever-changing marine environment.

Integration with Other Components

- Dependent on CanBusListener.py: Relies on the telemetry data processed by CanBusListener.py for informed and accurate control decisions.
- Collaboration with Server.py: Contributes to the broader functionalities provided by Server.py, including control and monitoring capabilities accessible through the system's API.

System Architecture and Data Flow

- Operational Link in System Chain: Occupies a central role in converting digital commands into mechanical responses, crucial for the winch system's operation.
- Seamless Inter-component Collaboration: Works in concert with CanBusListener.py, ensuring a fluid transition from data acquisition to action implementation.

Conclusion

ServoDriver.py is essential for the effective operation of the marine tether management winch system. Its role in managing physical movements, combined with its capacity for real-time adaptive control, makes it a cornerstone in the system's architecture and functionality.

Server.py (API_Server)

Overview

API_Server is a key component in the marine tether management winch system, built using FastAPI. This script establishes a server that acts as a communication interface for the system, handling requests and responses through defined API endpoints.

Core Functionalities

- API Endpoints Creation: The script uses FastAPI to create a range of API endpoints for data communication, system control, and status checks.
- Data Handling and Response: It processes requests sent to these endpoints and returns the appropriate responses, handling system data and command execution.
- Integration with System Components: Interacts with components like CanBusListener.py and ServoDriver.py to fetch or relay data and commands, facilitating system control and data access.
- Server.py features a monitoring mechanism that automatically restarts the server if a 10-second lapse in "put" request activity indicates potential inactivity or malfunction. To avert this auto-restart and the triggering of safety protocols (like setting drum_speed to zero and engaging the stop command), **it's crucial when using the API to consistently send 'put' operations to the commands endpoint every 0.5 seconds.** This frequent communication acts as a heartbeat, assuring the server of ongoing API client activity and thus maintaining system stability and continuous operation, while preventing unnecessary downtime due to perceived inactivity.

API Endpoints and Their Functions

1. **General Information:**
   - /general_info/: Fetches compiled general parameters of the system, including voltage, current, power, temperature, layer information, and more.
   - /general_info/{param}: Retrieves a specific general parameter based on the provided parameter name.
2. **Tether Parameters:**
   - /tetherparams/: Returns all tether parameters, crucial for understanding the tether's current status and characteristics.
   - /tetherparams/{param}: Provides a specific tether parameter, identified by the parameter name.
3. **Motor Parameters:**
   - /motorparams/: Retrieves all parameters related to the motors, including their status and operational data.
   - /motorparams/{param}: Accesses a specific motor's parameters based on the motor ID.
4. **Commands Handling:**
   - /commands/: Reads all available commands in the system, reflecting current values and data types.
   - /commands/{command}: Fetches a specific command's current value and data type.
   - /commands/{command} (PUT): Updates the value of a specified command.

Key Roles in the System

- Central Communication Hub: Provides a web-based interface for external clients to interact with the system, facilitating real-time control and monitoring.
- Real-Time Data Access and Control: Enables access to up-to-date system data and control mechanisms, enhancing the system's responsiveness.

FastAPI Features and Advantages

- Asynchronous Support: Ensures high-performance handling of simultaneous requests, crucial in a data-intensive environment.
- Data Validation and Documentation: Offers built-in request data validation and interactive API documentation, improving usability and developer experience.

Integration with Other Components

- Data Access and Control Commands: Enables access to telemetry data and control commands, linking with CanBusListener.py and ServoDriver.py for comprehensive system control.

System Architecture and Data Flow

- API-Centric Architecture: Establishes an API-driven architecture, centralizing data flow and system interactions through web requests.
- Efficient Data Management: Streamlines data handling and command processing for effective system operation.

Conclusion

API_Server, leveraging FastAPI, is an integral part of the marine tether management winch system, enabling robust, efficient web-based interactions for comprehensive system control and monitoring. Its various API endpoints play a pivotal role in system communication and operational effectiveness.

API Examples with FastAPI Server Configuration:

Commands Endpoint:

These examples demonstrate how to interact with the `commands` endpoint to GET the current value and PUT an update for the `drum_speed` command. You can replicate this pattern for other commands like `set_tether_diameter`, `set_tether_length`, etc., by changing the endpoint URL and the JSON payload as required.

Available Commands:

stop: Data Type – `bool`
drum_speed: Data Type – `int`
set_tether_diameter: Data Type – `float`
set_tether_length: Data Type – `int`
calibrate_levelwind_position: Data Type – `bool`
find_home_position: Data Type – `bool`
pi_oninit: Data Type – `bool`
pi_restart: Data Type - `bool`

Python 3 Example (GET and PUT)

```
import requests

# Example command: Get the current 'drum_speed'
response = requests.get('http://tmscontrol.local:5000/commands/drum_speed')
current_drum_speed = response.json()
print("Current Drum Speed (GET):", current_drum_speed)

# Example command: Update the 'drum_speed'
new_drum_speed = 100
response = requests.put('http://tmscontrol.local:5000/commands/drum_speed', json={"value":
new_drum_speed})
updated_drum_speed = response.json()
print("Updated Drum Speed (PUT):", updated_drum_speed)
```

JavaScript Example (GET and PUT using Axios)

```javascript
const axios = require('axios');

// Example command: Get the current 'drum_speed'
axios.get('http://tmscontrol.local:5000/commands/drum_speed')
  .then((response) => {
    const currentDrumSpeed = response.data;
    console.log("Current Drum Speed (GET):", currentDrumSpeed);

    // Example command: Update the 'drum_speed'
    const newDrumSpeed = 100;
    return axios.put('http://tmscontrol.local:5000/commands/drum_speed', { value: newDrumSpeed });
  })
  .then((response) => {
    const updatedDrumSpeed = response.data;
    console.log("Updated Drum Speed (PUT):", updatedDrumSpeed);
  })
  .catch((error) => {
    console.error(error);
  })
```

C++ Example (GET and PUT using libcurl)

```
#include <iostream>
#include <curl/curl.h>

int main() {
    CURL* curl = curl_easy_init();
    if (curl) {
        // Example command: Get the current 'drum_speed'
        curl_easy_setopt(curl, CURLOPT_URL, "http://tmscontrol.local:5000/commands/drum_speed");
        CURLcode res = curl_easy_perform(curl);
        if (res == CURLE_OK) {
            std::cout << "Current Drum Speed (GET): " << curl_easy_escape(curl, curl_easy_strerror(res),
0) << std::endl;
        }

        // Example command: Update the 'drum_speed'
        const char* putData = R"({"value": 100})";
        curl_easy_setopt(curl, CURLOPT_URL, "http://tmscontrol.local:5000/commands/drum_speed");
        curl_easy_setopt(curl, CURLOPT_CUSTOMREQUEST, "PUT");
        curl_easy_setopt(curl, CURLOPT_POSTFIELDS, putData);
        res = curl_easy_perform(curl);
        if (res == CURLE_OK) {
            std::cout << "Updated Drum Speed (PUT): " << curl_easy_escape(curl, curl_easy_strerror(res),
0) << std::endl;
        }

        curl_easy_cleanup(curl);
    }
    return 0;
}
```

General Info Endpoint:

These examples will make GET requests to the specified endpoint and output the retrieved general information, enabling external scripts to easily access and display data from your FastAPI server.

In the C++ example, a callback function `WriteCallback` is used to handle the data returned by libcurl. This function appends the fetched data to a `std::string` object (`readBuffer`), which is then printed to the console.

Python 3 Example (GET)

```
import requests

# GET request to fetch compiled general parameters
response = requests.get('http://tmscontrol.local:5000/general_info/')
general_info = response.json()
print("General Info (GET):", general_info)
```

JavaScript Example (GET using Axios)

```
const axios = require('axios');

// GET request to fetch compiled general parameters
axios.get('http://tmscontrol.local:5000/general_info/')
  .then((response) => {
    const generalInfo = response.data;
    console.log("General Info (GET):", generalInfo);
  })
  .catch((error) => {
    console.error(error);
  });
```

C++ Example (GET using libcurl)

```cpp
#include <iostream>
#include <curl/curl.h>

// Callback function for handling the data returned by libcurl
static size_t WriteCallback(void *contents, size_t size, size_t nmemb, void *userp) {
    ((std::string*)userp)->append((char*)contents, size * nmemb);
    return size * nmemb;
}

int main() {
    CURL* curl = curl_easy_init();
    if (curl) {
        std::string readBuffer;

        // GET request to fetch compiled general parameters
        curl_easy_setopt(curl, CURLOPT_URL, "http://tmscontrol.local:5000/general_info/");
        curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, WriteCallback);
        curl_easy_setopt(curl, CURLOPT_WRITEDATA, &readBuffer);
        CURLcode res = curl_easy_perform(curl);
        if (res == CURLE_OK) {
            std::cout << "General Info (GET): " << readBuffer << std::endl;
        } else {
            std::cerr << "CURL Error: " << curl_easy_strerror(res) << std::endl;
        }

        curl_easy_cleanup(curl);
    }
    return 0;
}
```

Specific General Info Endpoint:

These examples demonstrate how to make GET requests to fetch a specific general parameter from the FastAPI server.

The C++ example uses the `WriteCallback` function to handle the data returned by libcurl and appends the fetched data to a `std::string` object (`readBuffer`).

General Info Parameters:

voltage: Data Type - `float`
current: Data Type - `float`
power: Data Type - `float`
temp: Data Type - `float`
current_layer: Data Type - `int`
layer_fill: Data Type - `float`
tether_diameter: Data Type - `float`
tether_length: Data Type - `float`
tether_length_out: Data Type - `float`
tether_length_max: Data Type - `float`
current_time: Data Type - `float` or `string` (depending on format)
leadscrew_motor_faultbits: Data Type - `int` or `string`
tensioner_motor_faultbits: Data Type - `int` or `string`
leadscrew_motor_faults: Data Type - `string`
tensioner_motor_faults: Data Type - `string`

Python 3 Example (GET)

```
import requests

# Replace 'voltage' with the desired parameter name
param_name = 'voltage'

# GET request to fetch a specific general parameter
response = requests.get(f'http://tmscontrol.local:5000/general_info/{param_name}')
specific_param = response.json()
print(f"{param_name} (GET):", specific_param)
```

JavaScript Example (GET using Axios)

```javascript
const axios = require('axios');

// Replace 'voltage' with the desired parameter name
const paramName = 'voltage';

// GET request to fetch a specific general parameter
axios.get(`http://tmscontrol.local:5000/general_info/${paramName}`)
  .then((response) => {
    const specificParam = response.data;
    console.log(`${paramName} (GET):`, specificParam);
  })
  .catch((error) => {
    console.error(error);
  });
```

C++ Example (GET using libcurl)

```cpp
#include <iostream>
#include <curl/curl.h>
#include <string>

// Callback function for handling the data returned by libcurl
static size_t WriteCallback(void *contents, size_t size, size_t nmemb, void *userp) {
    ((std::string*)userp)->append((char*)contents, size * nmemb);
    return size * nmemb;
}

int main() {
    CURL* curl = curl_easy_init();
    if (curl) {
        std::string readBuffer;

        // Replace 'voltage' with the desired parameter name
        std::string paramName = "voltage";

        // GET request to fetch a specific general parameter
        std::string url = "http://tmscontrol.local:5000/general_info/" + paramName;
        curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
        curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, WriteCallback);
        curl_easy_setopt(curl, CURLOPT_WRITEDATA, &readBuffer);
        CURLcode res = curl_easy_perform(curl);
        if (res == CURLE_OK) {
            std::cout << paramName << " (GET): " << readBuffer << std::endl;
        } else {
            std::cerr << "CURL Error: " << curl_easy_strerror(res) << std::endl;
        }

        curl_easy_cleanup(curl);
    }
    return 0;
}
```

Motor Parameters Endpoint:

In these examples, a GET request is sent to the `motorparams` endpoint, and the motor parameters are retrieved and printed. These examples demonstrate how various programming languages can interact with the FastAPI server to fetch important data.

In the C++ example, a callback function WriteCallback is used to handle the data returned by libcurl. This function appends the fetched data to a std::string object (readBuffer), which is then printed to the console. These examples will make GET requests to the specified endpoint and output the retrieved motor parameters, enabling external scripts to easily access and display data from your FastAPI server.

Python 3 Example (GET)

```python
import requests


# GET request to fetch motor parameters
response = requests.get('http://tmscontrol.local:5000/motorparams/')
motor_params = response.json()
print("Motor Parameters (GET):", motor_params)
```

JavaScript Example (GET using Axios)

```javascript
const axios = require('axios');


// GET request to fetch motor parameters
axios.get('http://tmscontrol.local:5000/motorparams/')
  .then((response) => {
    const motorParams = response.data;
    console.log("Motor Parameters (GET):", motorParams);
  })
  .catch((error) => {
    console.error(error);
  });
```

C++ Example (GET using libcurl)

```cpp
#include <iostream>
#include <curl/curl.h>

// Callback function for handling the data returned by libcurl
static size_t WriteCallback(void *contents, size_t size, size_t nmemb, void *userp) {
    ((std::string*)userp)->append((char*)contents, size * nmemb);
    return size * nmemb;
}

int main() {
    CURL* curl = curl_easy_init();
    if (curl) {
        std::string readBuffer;

        // GET request to fetch motor parameters
        curl_easy_setopt(curl, CURLOPT_URL, "http://tmscontrol.local:5000/motorparams/");
        curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, WriteCallback);
        curl_easy_setopt(curl, CURLOPT_WRITEDATA, &readBuffer);
        CURLcode res = curl_easy_perform(curl);
        if (res == CURLE_OK) {
            std::cout << "Motor Parameters (GET): " << readBuffer << std::endl;
        } else {
            std::cerr << "CURL Error: " << curl_easy_strerror(res) << std::endl;
        }

        curl_easy_cleanup(curl);
    }
    return 0;
}
```

Specific Motor Parameters Endpoint:

In these examples, the GET request is sent to the `/motorparams/{param}` endpoint to retrieve all parameters for a specified motor ID. Replace `'motor_id'` with the appropriate ID (101 for leadscrewmotor, 102 for tensionermotor, or 0 for primarymotor) to fetch data for different motors.

These examples demonstrate how to make GET requests to fetch all available parameters from a single motor using the FastAPI server. The C++ example utilizes the `WriteCallback` function to handle the data returned by libcurl and appends it to a `std::string` object (`readBuffer`).

Motor Ids:

101: Leadscrew Motor
102: Tensioner Motor
0: Primary Motor

Python 3 Example (GET)

```python
import requests

# Replace 'motor_id' with the desired motor ID
motor_id = 101  # leadscrewmotor

# GET request to fetch all parameters from a specific motor
response = requests.get(f'http://tmscontrol.local:5000/motorparams/{motor_id}')
motor_params = response.json()
print(f"Motor {motor_id} Parameters (GET):", motor_params)
```

JavaScript Example (GET using Axios)

```javascript
const axios = require('axios');

// Replace 'motor_id' with the desired motor ID
const motorId = 101;  // leadscrewmotor

// GET request to fetch all parameters from a specific motor
axios.get(`http://tmscontrol.local:5000/motorparams/${motorId}`)
  .then((response) => {
    const motorParams = response.data;
    console.log(`Motor ${motorId} Parameters (GET):`, motorParams);
  })
  .catch((error) => {
    console.error(error);
  });
```

C++ Example (GET using libcurl)

```
#include <iostream>
#include <curl/curl.h>
#include <string>

// Callback function for handling the data returned by libcurl
static size_t WriteCallback(void *contents, size_t size, size_t nmemb, void *userp) {
    ((std::string*)userp)->append((char*)contents, size * nmemb);
    return size * nmemb;
}

int main() {
    CURL* curl = curl_easy_init();
    if (curl) {
        std::string readBuffer;

        // Replace 'motor_id' with the desired motor ID
        int motorId = 101;  // leadscrewmotor

        // GET request to fetch all parameters from a specific motor
        std::string url = "http://tmscontrol.local:5000/motorparams/" + std::to_string(motorId);
        curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
        curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, WriteCallback);
        curl_easy_setopt(curl, CURLOPT_WRITEDATA, &readBuffer);
        CURLcode res = curl_easy_perform(curl);
        if (res == CURLE_OK) {
            std::cout << "Motor " << motorId << " Parameters (GET): " << readBuffer << std::endl;
        } else {
            std::cerr << "CURL Error: " << curl_easy_strerror(res) << std::endl;
        }

        curl_easy_cleanup(curl);
    }
    return 0;
}
```

Tether Parameters Endpoint:

In these examples, a GET request is sent to the `tetherparams` endpoint. The received response contains the tether parameters, which are then output for review. These snippets are useful for external scripts that need to retrieve tether-related information from your FastAPI server.

In the C++ example, the `WriteCallback` function is employed to handle data returned by libcurl, appending the fetched data to a `std::string` object (`readBuffer`). The received tether parameters are then printed to the console. These examples demonstrate how to make GET requests to the specified endpoint, facilitating external scripts in accessing and displaying tether parameters data from your FastAPI server.

Python 3 Example (GET)

```python
import requests

# GET request to fetch tether parameters
response = requests.get('http://tmscontrol.local:5000/tetherparams/')
tether_params = response.json()
print("Tether Parameters (GET):", tether_params)
```

JavaScript Example (GET using Axios)

```javascript
Const axios = require('axios');

// GET request to fetch tether parameters
axios.get('http://tmscontrol.local:5000/tetherparams/')
  .then((response) => {
    const tetherParams = response.data;
    console.log("Tether Parameters (GET):", tetherParams);
  })
  .catch((error) => {
    console.error(error);
  });
```

C++ Example (GET using libcurl)

```cpp
#include <iostream>
#include <curl/curl.h>

// Callback function for handling the data returned by libcurl
static size_t WriteCallback(void *contents, size_t size, size_t nmemb, void *userp) {
    ((std::string*)userp)->append((char*)contents, size * nmemb);
    return size * nmemb;
}

int main() {
    CURL* curl = curl_easy_init();
    if (curl) {
        std::string readBuffer;

        // GET request to fetch tether parameters
        curl_easy_setopt(curl, CURLOPT_URL, "http://tmscontrol.local:5000/tetherparams/");
        curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, WriteCallback);
        curl_easy_setopt(curl, CURLOPT_WRITEDATA, &readBuffer);
        CURLcode res = curl_easy_perform(curl);
        if (res == CURLE_OK) {
            std::cout << "Tether Parameters (GET): " << readBuffer << std::endl;
        } else {
            std::cerr << "CURL Error: " << curl_easy_strerror(res) << std::endl;
        }

        curl_easy_cleanup(curl);
    }
    return 0;
}
```

Specific Tether Parameters Endpoint:

In these examples, the GET request is targeted to a specific parameter under the tether information category, such as `tether_diameter`. Replace `'tether_diameter'` with any other parameter name as needed to fetch different data.

These examples demonstrate how to make GET requests to fetch a specific tether parameter from the FastAPI server. The C++ example uses the `WriteCallback` function to handle the data returned by libcurl and appends the fetched data to a `std::string` object (`readBuffer`).

Tether Parameters:

Tether_Diameter: Data Type - `float`
Tether_Length: Data Type - `float`
Reel_Fill_Relief: Data Type - `float`
TurnPerLayer: Data Type - `float`
Travel_Distance_Per_Turn: Data Type - `float`
Maximum_Number_of_Layers: Data Type - `int`
End_Workzone: Data Type - `int`
Degree_Range_Per_Layer:
`layer_1` to `layer_[Maximum_Number_of_Layers]`: Data Type - `list` of `int` or `float`
Circumference_of_Layers:
`layer_1` to `layer_[Maximum_Number_of_Layers]`: Data Type - `float`
Max_Pay_Speed_of_Spool:
`layer_1` to `layer_[Maximum_Number_of_Layers]`: Data Type - `float`
Layer_Tether_Lengths_Per_Degree:
`layer_1` to `layer_[Maximum_Number_of_Layers]`: Data Type - `float`
Tether_Length_Per_Layer:
`layer_1` to `layer_[Maximum_Number_of_Layers]`: Data Type - `float`
Max_Tether_Length: Data Type - `float`
Length_Tether_Out: Data Type - `float`

In this structure, the parameters related to specific layers (`layer_1` to `layer_[Maximum_Number_of_Layers]`) are dynamically determined based on the maximum number of layers that can fit, which is defined by `Maximum_Number_of_Layers`.

Python 3 Example (GET)

```python
import requests

# Replace 'tether_diameter' with the desired parameter name
param_name = 'tether_diameter'

# GET request to fetch a specific tether parameter
response = requests.get(f'http://tmscontrol.local:5000/tetherparams/{param_name}')
specific_param = response.json()
print(f"{param_name} (GET):", specific_param)
```

JavaScript Example (GET using Axios)

```javascript
const axios = require('axios');

// Replace 'tether_diameter' with the desired parameter name
const paramName = 'tether_diameter';

// GET request to fetch a specific tether parameter
axios.get(`http://tmscontrol.local:5000/tetherparams/${paramName}`)
  .then((response) => {
    const specificParam = response.data;
    console.log(`${paramName} (GET):`, specificParam);
  })
  .catch((error) => {
    console.error(error);
  });
```

C++ Example (GET using libcurl)

```cpp
#include <iostream>
#include <curl/curl.h>
#include <string>

// Callback function for handling the data returned by libcurl
static size_t WriteCallback(void *contents, size_t size, size_t nmemb, void *userp) {
    ((std::string*)userp)->append((char*)contents, size * nmemb);
    return size * nmemb;
}

int main() {
    CURL* curl = curl_easy_init();
    if (curl) {
        std::string readBuffer;

        // Replace 'tether_diameter' with the desired parameter name
        std::string paramName = "tether_diameter";

        // GET request to fetch a specific tether parameter
        std::string url = "http://tmscontrol.local:5000/tetherparams/" + paramName;
        curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
        curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, WriteCallback);
        curl_easy_setopt(curl, CURLOPT_WRITEDATA, &readBuffer);
        CURLcode res = curl_easy_perform(curl);
        if (res == CURLE_OK) {
            std::cout << paramName << " (GET): " << readBuffer << std::endl;
        } else {
            std::cerr << "CURL Error: " << curl_easy_strerror(res) << std::endl;
        }

        curl_easy_cleanup(curl);
    }
    return 0;
}
```