Servomotor Python API Documentation

Version 1.4

Generated: 2025-08-31

Latest Firmware Versions

At the time of generating this API reference, the latest released firmware versions for the servomotors are:

• **Model M17:** servomotor_M17_fw0.13.0.0_scc3_hw1.4.firmware

If you are experiencing problems, you can try to set the firmware of your product to this version and try again, and report the problem to us using the feedback page.

Table of Contents

Install the Python Library

You need to install the servomotor Python library before you can use it in your code. Run this command:

```
pip3 install servomotor
```

If you want to work in a virtual environment then you can create it, activate it, and install the requirements. You can create the following requirements.txt file and then run the following commands:

Create requirements.txt:

```
servomotor
```

For macOS/Linux:

```
# Create virtual environment
python3 -m venv venv

# Activate virtual environment
source venv/bin/activate

# Install requirements
pip3 install -r requirements.txt
```

For Windows:

```
# Create virtual environment
python -m venv venv

# Activate virtual environment
venv\Scripts\activate

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

After installation, you can verify the servomotor library is installed correctly by running:

```
python3 -c "import servomotor; print('Servomotor library installed successfully!')"
```

Getting Started

This section provides a complete example showing how to initialize and control a servomotor.

Complete Example Program:

```
import servomotor
import time

# Connect to the servomotor
motor = servomotor.M3(port='/dev/ttyUSB0')

# Enable the mosfets (power on the motor)
motor.enable_mosfets()

# Wait for motor to be ready
time.sleep(0.5)

# Perform a trapezoid move
# Move 10000 steps with acceleration 1000, velocity 5000
motor.trapezoid_move(
    position=10000,  # Target position in steps
    velocity=5000,   # Maximum velocity in steps/second
    acceleration=1000 # Acceleration in steps/second^2
)

# Wait for move to complete
motor.wait_for_move_complete()

# Move back to starting position
motor.trapezoid_move(
    position=0,
    velocity=5000,
    acceleration=1000
)

# Wait and disable mosfets
motor.wait_for_move_complete()
motor.disable_mosfets()

# Close connection
motor.close()
```

Data Types

This section describes the various data types used in the Servomotor API commands.

Integer Data Types

Type	Size (bytes)	Range	Description
i16	2	-32,768 to 32,767	16-bit signed integer
i24	3	-8,388,608 to 8,388,607	24-bit signed integer
i32	4	-2,147,483,648 to 2,147,483,647	32-bit signed integer
i48	6	-549,755,813,888 to 549,755,813,887	48-bit signed integer
i64	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	64-bit signed integer
i8	1	-128 to 127	8-bit signed integer
u16	2	0 to 65,535	16-bit unsigned integer
u24	3	0 to 16,777,215	24-bit unsigned integer
u32	4	0 to 4,294,967,295	32-bit unsigned integer
u48	6	0 to 1,099,511,627,775	48-bit unsigned integer
u64	8	0 to 18,446,744,073,709,551,615	64-bit unsigned integer
u8	1	0 to 255	8-bit unsigned integer

Special Data Types

Type	Size (bytes)	Description
buf10	10	10 byte long buffer containing any binary data
crc32	4	32-bit CRC
firmware_page	2058	This is the data to upgrade one page of flash memory. Contents includes the product model code (8 bytes), firmware compatibility code (1 byte), page number (1 byte), and the page data itself (2048 bytes).
general_data	Variable	This is some data. You will need to look elsewhere at some documentation or into the source code to find out what this data is.

list_2d	Variable	A two dimensional list in a Python style format, for example: [[1, 2], [3, 4]]
string8	8	8 byte long string with null termination if it is shorter than 8 bytes
string_null_term	Variable	This is a string with a variable length and must be null terminated
success_response	Variable	Indicates that the command was received successfully and is being executed. The next command can be immediately transmitted without causing a command overflow situation.
u24_version_number	3	3 byte version number. the order is patch, minor, major
u32_version_number	4	4 byte version number. the order is development number, patch, minor, major
u64_unique_id	8	The unique ID of the device (8-bytes long)
u8_alias	1	This can hold an ASCII character where the value is represented as an ASCII character if it is in the range 33 to 126, otherwise it is represented as a number from 0 to 255
unknown_data	Variable	This is an unknown data type (work in progress; will be corrected and documented later)

Command Reference

This section documents all available commands organized by category.

Basic Control

Disable MOSFETs

Disables the MOSFETS (note that MOSFETs are disabled after initial power on).

Example:

```
motor.disable_mosfets()
```

Enable MOSFETs

Enables the MOSFETS.

Example:

```
motor.enable_mosfets()
```

Reset time

Resets the absolute time to zero (call this first before issuing any movement commands)

Example:

```
motor.reset_time()
```

Emergency stop

Emergency stop (stop all movement, disable MOSFETS, clear the queue)

Example:

```
motor.emergency_stop()
```

Zero position

Make the current position the position zero (origin)

Example:

```
motor.zero_position()
```

System reset

System reset / go to the bootloader. The motor will reset immediately and will enter the bootloader. If there is no command sent within a short time, the motor will exit the bootloader and run the application from the beginning.

Example:

```
motor.system_reset()
```

Configuration

Set maximum velocity

Sets maximum velocity (this is not used at this time)

Parameters:
• *maximumVelocity*: u32: Maximum velocity.

Example:

```
maximumVelocity_value = 0  # Replace with your desired value

motor.set_maximum_velocity(maximumVelocity=maximumVelocity_value)
```

Set maximum acceleration

Sets max acceleration

Parameters:
• *maximumAcceleration*: u32: The maximum acceleration.

Example:

```
maximumAcceleration_value = 0  # Replace with your desired value

motor.set_maximum_acceleration(
    maximumAcceleration=maximumAcceleration_value
)
```

Start calibration

Starts a calibration, which will determine the average values of the hall sensors and will determine if they are working correctly

Example:

```
motor.start_calibration()
```

Set maximum motor current

Set the maximum motor current and maximum regeneration current. The values are stored in non-volatile memory and survive a reset.

Parameters:
• *motorCurrent*: u16: The motor current. The units are some arbitrary units and not amps. A value of 150 or 200 is suitable.
• *regenerationCurrent*: u16: The motor regeneration current (while it is braking). This parameter is currently not used for anything.

Example:

```
motorCurrent_value = 0  # Replace with your desired value
regenerationCurrent_value = 0  # Replace with your desired value

motor.set_maximum_motor_current(
    motorCurrent=motorCurrent_value,
    regenerationCurrent=regenerationCurrent_value
)
```

Set safety limits

Set safety limits (to prevent motion from exceeding set bounds)

Parameters:
• *lowerLimit*: i64: The lower limit in microsteps.
• *upperLimit*: i64: The upper limit in microsteps.

Example:

```
lowerLimit_value = 0  # Replace with your desired value
upperLimit_value = 0  # Replace with your desired value

motor.set_safety_limits(
    lowerLimit=lowerLimit_value,
    upperLimit=upperLimit_value
)
```

Test mode

Set or trigger a certain test mode. This is a bit undocumented at the moment. Don't use this unless you are a developer working on test cases.

Parameters:
• *testMode*: u8: The test mode to use or trigger

Example:

```
testMode_value = 0  # Replace with your desired value

motor.test_mode(testMode=testMode_value)
```

Set PID constants

Set PID constants for the control loop that will try to maintain the motion trajectory.

Parameters:
• *kP*: u32: The proportional term constant (P)
• *kI*: u32: The integral term constant (I)
• *kD*: u32: The differential term constant (D)

Example:

```
kP_value = 0  # Replace with your desired value
kI_value = 0  # Replace with your desired value
kD_value = 0  # Replace with your desired value

motor.set_pid_constants(
    kP=kP_value,
    kI=kI_value,
    kD=kD_value
)
```

Set max allowable position deviation

Set the amount of microsteps that the actual motor position (as measured by the hall sensors) is allowed to deviate from the desired position. Throw a fatal error if this is exceeded.

Parameters:
• *maxAllowablePositionDeviation*: i64: The new maximum allowable position deviation setting

Example:

```
maxAllowablePositionDeviation_value = 0  # Replace with your desired value

motor.set_max_allowable_position_deviation(
    maxAllowablePositionDeviation=maxAllowablePositionDeviation_value
)
```

CRC32 control

Enable or disable CRC32 checking for commands

Parameters:
• *enableCrc32*: u8: Control value (1 to enable, 0 to disable CRC32 checking)

Example:

```
enableCrc32_value = 0  # Replace with your desired value

motor.crc32_control(enableCrc32=enableCrc32_value)
```

Device Management

Time sync

Sends the master time to the motor so that it can sync its own clock (do this 10 times per second).

Parameters:
• *masterTime*: u48: The motor absolute time that the motor should sync to (in microseconds).

Returns:
• *timeError*: i32: The error in the motor's time compared to the master time.
• *rccIcscr*: u16: The contents of the RCC-ICSCR register (holds the HSICAL and HSITRIM settings).

Example:

```
masterTime_value = 0  # Replace with your desired value

timeError, rccIcscr = motor.time_sync(masterTime=masterTime_value)
print(f"timeError: {timeError}")
print(f"rccIcscr: {rccIcscr}")
```

Get product specs

Get the update frequency (reciprocal of the time step)

Returns:
• *updateFrequency*: u32: Update frequency in Hz. This is how often the motor executes all calculations for hall sensor position, movement, PID loop, safety, etc.
• *countsPerRotation*: u32: Counts per rotation. When commanding the motor or when reading back position, this is the number of counts per one shaft rotation.

Example:

```
updateFrequency, countsPerRotation = motor.get_product_specs()
print(f"updateFrequency: {updateFrequency}")
print(f"countsPerRotation: {countsPerRotation}")
```

Detect devices

Detect all of the devices that are connected on the RS485 interface. Devices will identify themselves at a random time within one seconde. Chance of collision is possible but unlikely. You can repeat this if you suspect a collision (like if you have devices connected but they were not discovered within one to two seconds).

Returns:
• *uniqueId*: u64_unique_id: A unique ID (unique among all devices manufactured). The response is sent after a random delay of between 0 and 1 seconds.
• *alias*: u8_alias: The alias of the device that has this unique ID.

Example:

```
uniqueId, alias = motor.detect_devices()
print(f"uniqueId: {uniqueId}")
print(f"alias: {alias}")
```

Set device alias

Sets device alias

Parameters:
• *alias*: u8_alias: The alias (which is a one byte ID) ranging from 0 to 251. It cannot be 252 to 254 because those are reserved. You can set it to 255, which will remove the alias.

Example:

```
alias_value = 0  # Replace with your desired value

motor.set_device_alias(alias=alias_value)
```

Get product info

Get product information

Returns:
• *productCode*: string8: The product code / model number (when doing a firmware upgrade, this must match between the firmware file and the target device).
• *firmwareCompatibility*: u8: A firmware compatibility code (when doing a firmware upgrade, this must match between the firmware file and the target device).
• *hardwareVersion*: u24_version_number: The hardware version stored as 3 bytes. The first byte is the patch version, followed by the minor and major versions.
• *serialNumber*: u32: The serial number.
• *uniqueId*: u64_unique_id: The unique ID for the product.
• *reserved*: u32: Not currently used.

Example:

```
(
    productCode,
    firmwareCompatibility,
    hardwareVersion,
    serialNumber,
    uniqueId,
    reserved
) = motor.get_product_info()
print(f"productCode: {productCode}")
print(f"firmwareCompatibility: {firmwareCompatibility}")
print(f"hardwareVersion: {hardwareVersion}")
print(f"serialNumber: {serialNumber}")
print(f"uniqueId: {uniqueId}")
print(f"reserved: {reserved}")
```

Firmware upgrade

This command will upgrade the flash memory of the servo motor. Before issuing a firmware upgrade command, you must do some calculations as shown in the examples.

Parameters:
• *firmwarePage*: firmware_page: The data to upgrade one page of flash memory. Contents includes the product model code (8 bytes), firmware compatibility code (1 byte), page number (1 byte), and the page data itself (2048 bytes).

Example:

```
firmwarePage_value = 0  # Replace with your desired value

motor.firmware_upgrade(firmwarePage=firmwarePage_value)
```

Get product description

Get the product description.

Returns:
• *productDescription*: string_null_term: This is a brief description of the product.

Example:

```
productDescription = motor.get_product_description()
print(f"productDescription: {productDescription}")
```

Get firmware version

Get the firmware version or the bootloader version depending on what mode we are in. This command also returns the status bits, where the least significan bit teels us if we are currently in the bootloader (=1) or the main firmware (=0)

Returns:
• *firmwareVersion*: u32_version_number: The firmware version stored as 4 bytes. The first byte is the development number, then patch version, followed by the minor and major versions.
• *inBootloader*: u8: A flag that tells us if we are in the bootloader (=1) or in the reguslar firmware (=0)

Example:

```
firmwareVersion, inBootloader = motor.get_firmware_version()
print(f"firmwareVersion: {firmwareVersion}")
print(f"inBootloader: {inBootloader}")
```

Ping

Send a payload containing any data and the device will respond with the same data back

Parameters:
• *pingData*: buf10: Any binary data payload to send to the device.

Returns:
• *responsePayload*: buf10: The same data that was sent to the device will be returned if all went well.

Example:

```
pingData_value = 0  # Replace with your desired value

responsePayload = motor.ping(pingData=pingData_value)
print(f"responsePayload: {responsePayload}")
```

Vibrate

Cause the motor to start to vary the voltage quickly and therefore to vibrate (or stop).

Parameters:
• *vibrationLevel*: u8: Vibration level (0 = turn off, 1 = turn on).

Example:

```
vibrationLevel_value = 0  # Replace with your desired value

motor.vibrate(vibrationLevel=vibrationLevel_value)
```

Identify

Identify your motor by sending this command. The motor's green LED will flash rapidly for 3 seconds.

Example:

```
motor.identify()
```

Motion Control

Trapezoid move

Move immediately to the given position using the currently set speed (the speed is set by a separate command)

Parameters:
• *displacement*: i32: The displacement to travel. Can be positive or negative.
• *duration*: u32: The time over which to do the move.

Example:

```
displacement_value = 0  # Replace with your desired value
duration_value = 0  # Replace with your desired value

motor.trapezoid_move(
    displacement=displacement_value,
    duration=duration_value
)
```

Go to position

Move to this new given position in the amount of time specified. Acceleration and deceleration will be applied to make the move smooth.

Parameters:
• *position*: i32: New absolute position value.
• *duration*: u32: Time allowed for executing the move.

Example:

```
position_value = 0  # Replace with your desired value
duration_value = 0  # Replace with your desired value

motor.go_to_position(position=position_value, duration=duration_value)
```

Homing

Homing (or in other words, move until a crash and then stop immediately)

Parameters:
• *maxDistance*: i32: The maximum distance to move (if a crash does not occur). This can be positive or negative. the sign determines the direction of movement.
• *maxDuration*: u32: The maximum time to allow for homing. Make sure to give enough time for the motor to cover the maximum distance or the motor may move too fast or throw a fatal error.

Example:

```
maxDistance_value = 0  # Replace with your desired value
maxDuration_value = 0  # Replace with your desired value

motor.homing(
    maxDistance=maxDistance_value,
    maxDuration=maxDuration_value
)
```

Go to closed loop

Go to closed loop position control mode

Example:

```
motor.go_to_closed_loop()
```

Move with acceleration

Rotates the motor with the specified acceleration

Parameters:
• *acceleration*: i32: The acceleration (the unit is microsteps per time step per time step * 2^24).
• *timeSteps*: u32: The number of time steps to apply this acceleration. Use command 18 to get the frequency of the time steps. After this many time steps, the acceleration will go to zero and velocity will be maintained.

Example:

```
acceleration_value = 0  # Replace with your desired value
timeSteps_value = 0  # Replace with your desired value

motor.move_with_acceleration(
    acceleration=acceleration_value,
    timeSteps=timeSteps_value
)
```

Move with velocity

Rotates the motor with the specified velocity.

Parameters:
• *velocity*: i32: The velocity (the unit is microsteps per time step * 2^20).
• *duration*: u32: The time to maintain this velocity.

Example:

```
velocity_value = 0  # Replace with your desired value
duration_value = 0  # Replace with your desired value

motor.move_with_velocity(
    velocity=velocity_value,
    duration=duration_value
)
```

Multimove

The multimove command allows you to compose multiple moves one after another. Please note that when the queue becomes empty after all the moves are executed and the motor is not at a standstill then a fatal error will be triggered.

Parameters:
• *moveCount*: u8: Specify how many moves are being communicated in this one shot.
• *moveTypes*: u32: Each bit specifies if the move is a (bit = 0) MOVE_WITH_ACCELERATION_COMMAND or a (bit = 1) MOVE_WITH_VELOCITY_COMMAND.
• *moveList*: list_2d: A 2D list in Python format (list of lists). Each item in the list is of type [i32, u32] representing a series of move commands. Each move command specifies the acceleration to move at or the velocity to instantly change to (according to the bits above) and the number of time steps over which this command is to be executed. For example: '[[100, 30000], [-200, 60000]]'. There is a limit of 32 move commands that can be listed in this one multi-move command. Each of the moves takes up one queue spot, so make sure there is enough space in the queue to store all of the commands.

Example:

```
moveCount_value = 0  # Replace with your desired value
moveTypes_value = 0  # Replace with your desired value
moveList_value = 0  # Replace with your desired value

motor.multimove(
    moveCount=moveCount_value,
    moveTypes=moveTypes_value,
    moveList=moveList_value
)
```

Other

Capture hall sensor data

Start sending hall sensor data (work in progress; don't send this command)

Parameters:
• *captureType*: u8: Indicates the type of data to capture. Currently 1 to 3 are valid.
• *nPointsToRead*: u32: Number of points to read back from the device
• *channelsToCaptureBitmask*: u8: Channels to capture bitmask. The first three bits are valid, which will turn on (0) or turn off (0) that hall sensor channel
• *timeStepsPerSample*: u16: Aquire a sample every this number of time steps. Time steps happen at the update frequency, which can be read with the Get product specs command
• *nSamplesToSum*: u16: Number of samples to sum together to make one point to transmit back
• *divisionFactor*: u16: Division factor to apply to the sum of the samples to scale it down before transmitting it so that it fits into the returned data type, which is a 16-bit number per each hall sensor

Returns:
• *data*: general_data: The data of the hall sensors after suming and averaging

Example:

```
captureType_value = 0  # Replace with your desired value
nPointsToRead_value = 0  # Replace with your desired value
channelsToCaptureBitmask_value = 0  # Replace with your desired value
timeStepsPerSample_value = 0  # Replace with your desired value
nSamplesToSum_value = 0  # Replace with your desired value
divisionFactor_value = 0  # Replace with your desired value

data = motor.capture_hall_sensor_data(
    captureType=captureType_value,
    nPointsToRead=nPointsToRead_value,
    channelsToCaptureBitmask=channelsToCaptureBitmask_value,
    timeStepsPerSample=timeStepsPerSample_value,
    nSamplesToSum=nSamplesToSum_value,
    divisionFactor=divisionFactor_value
)
print(f"data: {data}")
```

Read multipurpose buffer

Read whatever is in the multipurpose buffer (the buffer is used for data generated during calibration, going to closed loop mode, and when capturing hall sensor data)

Returns:
• *bufferData*: general_data: The data in the buffer (the format and length of the data depends on what was put in the buffer)

Example:

```
bufferData = motor.read_multipurpose_buffer()
print(f"bufferData: {bufferData}")
```

Status & Monitoring

Get current time

Gets the current absolute time

Returns:
• *currentTime*: u48: The current absolute time

Example:

```
currentTime = motor.get_current_time()
print(f"currentTime: {currentTime}")
```

Get n queued items

Get the number of items currently in the movement queue (if this gets too large, don't queue any more movement commands)

Returns:
• *queueSize*: u8: The number of items in the movement queue. This command will return between 0 and 32. If less than 32, you can add more items to the queue to continue the movements in order without stopping.

Example:

```
queueSize = motor.get_n_queued_items()
print(f"queueSize: {queueSize}")
```

Get hall sensor position

Get the position as measured by the hall sensors (this should be the actual position of the motor and if everything is ok then it will be about the same as the desired position)

Returns:
• *hallSensorPosition*: i64: The current position as determined by the hall sensors

Example:

```
hallSensorPosition = motor.get_hall_sensor_position()
print(f"hallSensorPosition: {hallSensorPosition}")
```

Get status

Gets the status of the motor

Returns:
• *statusFlags*: u16: A series of flags which are 1 bit each
• *fatalErrorCode*: u8: The fatal error code. If 0 then there is no fatal error. Once a fatal error happens, the motor becomes disabled and cannot do much anymore until reset. You can press the reset button on the motor or you can execute the System reset command to get out of the fatal error state.

Example:

```
statusFlags, fatalErrorCode = motor.get_status()
print(f"statusFlags: {statusFlags}")
print(f"fatalErrorCode: {fatalErrorCode}")
```

Control hall sensor statistics

Turn on or off the gathering of statistics for the hall sensors and reset the statistics

Parameters:
• *control*: u8: 0 = turn off statistics gathering, 1 = reset statistics and turn on gathering.

Example:

```
control_value = 0  # Replace with your desired value

motor.control_hall_sensor_statistics(control=control_value)
```

Get hall sensor statistics

Read back the statistics gathered from the hall sensors. Useful for checking the hall sensor health and noise in the system.

Returns:
• *maxHall1*: u16: The maximum value of hall sensor 1 encoutered since the last statistics reset.
• *maxHall2*: u16: The maximum value of hall sensor 2 encoutered since the last statistics reset.
• *maxHall3*: u16: The maximum value of hall sensor 3 encoutered since the last statistics reset.
• *minHall1*: u16: The minimum value of hall sensor 1 encoutered since the last statistics reset.
• *minHall2*: u16: The minimum value of hall sensor 2 encoutered since the last statistics reset.
• *minHall3*: u16: The minimum value of hall sensor 3 encoutered since the last statistics reset.
• *sumHall1*: u64: The sum of hall sensor 1 values collected since the last statistics reset.
• *sumHall2*: u64: The sum of hall sensor 2 values collected since the last statistics reset.
• *sumHall3*: u64: The sum of hall sensor 3 values collected since the last statistics reset.
• *measurementCount*: u32: The number of times the hall sensors were measured since the last statistics reset.

Example:

```
(
    maxHall1,
    maxHall2,
    maxHall3,
    minHall1,
    minHall2,
    minHall3,
    sumHall1,
    sumHall2,
    sumHall3,
    measurementCount
) = motor.get_hall_sensor_statistics()
print(f"maxHall1: {maxHall1}")
print(f"maxHall2: {maxHall2}")
print(f"maxHall3: {maxHall3}")
print(f"minHall1: {minHall1}")
print(f"minHall2: {minHall2}")
print(f"minHall3: {minHall3}")
print(f"sumHall1: {sumHall1}")
print(f"sumHall2: {sumHall2}")
print(f"sumHall3: {sumHall3}")
print(f"measurementCount: {measurementCount}")
```

Get position

Get the current desired position (which may differ a bit from the actual position as measured by the hall sensors)

Returns:
• *position*: i64: The current desired position

Example:

```
position = motor.get_position()
print(f"position: {position}")
```

Get comprehensive position

Get the desired motor position, hall sensor position, and external encoder position all in one shot

Returns:
• *commandedPosition*: i64: The commanded position (which may differ from actual)
• *hallSensorPosition*: i64: The hall sensor position (or you could say the actual measured position)
• *externalEncoderPosition*: i32: The external encoder position. This needs special hardware attached to the motor to work

Example:

```
(
    commandedPosition,
    hallSensorPosition,
    externalEncoderPosition
) = motor.get_comprehensive_position()
print(f"commandedPosition: {commandedPosition}")
print(f"hallSensorPosition: {hallSensorPosition}")
print(f"externalEncoderPosition: {externalEncoderPosition}")
```

Get supply voltage

Get the measured voltage of the power supply.

Returns:
• *supplyVoltage*: u16: The voltage. Divide this number by 10 to get the actual voltage in volts.

Example:

```
supplyVoltage = motor.get_supply_voltage()
print(f"supplyVoltage: {supplyVoltage}")
```

Get max PID error

Get the minimum and maximum error value ovserved in the PID control loop since the last read.

Returns:
• *minPidError*: i32: The minimum PID error value.
• *maxPidError*: i32: The maximum PID error value.

Example:

```
minPidError, maxPidError = motor.get_max_pid_error()
print(f"minPidError: {minPidError}")
print(f"maxPidError: {maxPidError}")
```

Get temperature

Get the measured temperature of the motor.

Returns:
• *temperature*: i16: The temperature in degrees celcius. The accuracy is about +/- 3 degrees celcius and is measured at the motor driver PCB.

Example:

```
temperature = motor.get_temperature()
print(f"temperature: {temperature}")
```

Get debug values

Get debug values including motor control parameters, profiler times, hall sensor data, and other diagnostic information.

Returns:
• *maxAcceleration*: i64: Maximum acceleration setting
• *maxVelocity*: i64: Maximum velocity setting
• *currentVelocity*: i64: Current velocity
• *measuredVelocity*: i32: Measured velocity
• *nTimeSteps*: u32: Number of time steps left in the current move
• *debugValue1*: i64: Debug value 1
• *debugValue2*: i64: Debug value 2
• *debugValue3*: i64: Debug value 3
• *debugValue4*: i64: Debug value 4
• *allMotorControlCalculationsProfilerTime*: u16: All motor control calculations profiler time
• *allMotorControlCalculationsProfilerMaxTime*: u16: All motor control calculations profiler maximum time
• *getSensorPositionProfilerTime*: u16: Get sensor position profiler time
• *getSensorPositionProfilerMaxTime*: u16: Get sensor position profiler maximum time
• *computeVelocityProfilerTime*: u16: Compute velocity profiler time
• *computeVelocityProfilerMaxTime*: u16: Compute velocity profiler maximum time
• *motorMovementCalculationsProfilerTime*: u16: Motor movement calculations profiler time
• *motorMovementCalculationsProfilerMaxTime*: u16: Motor movement calculations profiler maximum time
• *motorPhaseCalculationsProfilerTime*: u16: Motor phase calculations profiler time
• *motorPhaseCalculationsProfilerMaxTime*: u16: Motor phase calculations profiler maximum time
• *motorControlLoopPeriodProfilerTime*: u16: Motor control loop period profiler time
• *motorControlLoopPeriodProfilerMaxTime*: u16: Motor control loop period profiler maximum time
• *hallSensor1Voltage*: u16: Hall sensor 1 voltage
• *hallSensor2Voltage*: u16: Hall sensor 2 voltage
• *hallSensor3Voltage*: u16: Hall sensor 3 voltage
• *commutationPositionOffset*: u32: Commutation position offset
• *motorPhasesReversed*: u8: Motor phases reversed flag
• *maxHallPositionDelta*: i32: Maximum hall position delta
• *minHallPositionDelta*: i32: Minimum hall position delta
• *averageHallPositionDelta*: i32: Average hall position delta
• *motorPwmVoltage*: u8: Motor PWM voltage

Example:

```
(
    maxAcceleration,
    maxVelocity,
    currentVelocity,
    measuredVelocity,
    nTimeSteps,
    debugValue1,
    debugValue2,
    debugValue3,
    debugValue4,
    allMotorControlCalculationsProfilerTime,
    allMotorControlCalculationsProfilerMaxTime,
    getSensorPositionProfilerTime,
    getSensorPositionProfilerMaxTime,
    computeVelocityProfilerTime,
    computeVelocityProfilerMaxTime,
    motorMovementCalculationsProfilerTime,
    motorMovementCalculationsProfilerMaxTime,
    motorPhaseCalculationsProfilerTime,
    motorPhaseCalculationsProfilerMaxTime,
    motorControlLoopPeriodProfilerTime,
    motorControlLoopPeriodProfilerMaxTime,
    hallSensor1Voltage,
    hallSensor2Voltage,
    hallSensor3Voltage,
    commutationPositionOffset,
    motorPhasesReversed,
    maxHallPositionDelta,
    minHallPositionDelta,
    averageHallPositionDelta,
    motorPwmVoltage
) = motor.get_debug_values()
print(f"maxAcceleration: {maxAcceleration}")
print(f"maxVelocity: {maxVelocity}")
print(f"currentVelocity: {currentVelocity}")
print(f"measuredVelocity: {measuredVelocity}")
print(f"nTimeSteps: {nTimeSteps}")
print(f"debugValue1: {debugValue1}")
print(f"debugValue2: {debugValue2}")
print(f"debugValue3: {debugValue3}")
print(f"debugValue4: {debugValue4}")
print(f"allMotorControlCalculationsProfilerTime: {allMotorControlCalculationsProfilerTime}")
print(f"allMotorControlCalculationsProfilerMaxTime: {allMotorControlCalculationsProfilerMaxTime}")
print(f"getSensorPositionProfilerTime: {getSensorPositionProfilerTime}")
print(f"getSensorPositionProfilerMaxTime: {getSensorPositionProfilerMaxTime}")
print(f"computeVelocityProfilerTime: {computeVelocityProfilerTime}")
print(f"computeVelocityProfilerMaxTime: {computeVelocityProfilerMaxTime}")
print(f"motorMovementCalculationsProfilerTime: {motorMovementCalculationsProfilerTime}")
print(f"motorMovementCalculationsProfilerMaxTime: {motorMovementCalculationsProfilerMaxTime}")
print(f"motorPhaseCalculationsProfilerTime: {motorPhaseCalculationsProfilerTime}")
print(f"motorPhaseCalculationsProfilerMaxTime: {motorPhaseCalculationsProfilerMaxTime}")
print(f"motorControlLoopPeriodProfilerTime: {motorControlLoopPeriodProfilerTime}")
print(f"motorControlLoopPeriodProfilerMaxTime: {motorControlLoopPeriodProfilerMaxTime}")
print(f"hallSensor1Voltage: {hallSensor1Voltage}")
print(f"hallSensor2Voltage: {hallSensor2Voltage}")
print(f"hallSensor3Voltage: {hallSensor3Voltage}")
print(f"commutationPositionOffset: {commutationPositionOffset}")
print(f"motorPhasesReversed: {motorPhasesReversed}")
print(f"maxHallPositionDelta: {maxHallPositionDelta}")
print(f"minHallPositionDelta: {minHallPositionDelta}")
print(f"averageHallPositionDelta: {averageHallPositionDelta}")
print(f"motorPwmVoltage: {motorPwmVoltage}")
```

Get communication statistics

Get and optionally reset the CRC32 error counter

Parameters:
• *resetCounter*: u8: Reset flag (1 to reset the counter after reading, 0 to just read)

Returns:
• *crc32ErrorCount*: u32: Number of CRC32 errors detected
• *packetDecodeErrorCount*: u32: Number of packet decode errors detected
• *firstBitErrorCount*: u32: Number of times that the first bit in the first byte of a packet was not 1 as expected
• *framingErrorCount*: u32: Number of framing errors detected during reception from the RS485 interface
• *overrunErrorCount*: u32: Number of overrun errors detected during reception from the RS485 interface
• *noiseErrorCount*: u32: Number of noise errors detected during reception from the RS485 interface

Example:

```
resetCounter_value = 0  # Replace with your desired value

(
    crc32ErrorCount,
    packetDecodeErrorCount,
    firstBitErrorCount,
    framingErrorCount,
    overrunErrorCount,
    noiseErrorCount
) = motor.get_communication_statistics(
    resetCounter=resetCounter_value
)
print(f"crc32ErrorCount: {crc32ErrorCount}")
print(f"packetDecodeErrorCount: {packetDecodeErrorCount}")
print(f"firstBitErrorCount: {firstBitErrorCount}")
print(f"framingErrorCount: {framingErrorCount}")
print(f"overrunErrorCount: {overrunErrorCount}")
print(f"noiseErrorCount: {noiseErrorCount}")
```

Unit Conversions

The servomotor library supports multiple unit systems for convenience.

Position Units:
• encoder_counts - Raw encoder counts (default)
• shaft_rotations - Rotations of the motor shaft
• degrees - Degrees of rotation
• radians - Radians of rotation

Velocity Units:
• counts_per_second - Encoder counts per second (default)
• rotations_per_second - Rotations per second
• rpm - Revolutions per minute
• degrees_per_second - Degrees per second
• radians_per_second - Radians per second

Acceleration Units:
• counts_per_second_squared - Encoder counts per second² (default)
• rotations_per_second_squared - Rotations per second²
• rpm_per_second - RPM per second
• degrees_per_second_squared - Degrees per second²
• radians_per_second_squared - Radians per second²

Time Units:
• seconds - Time in seconds
• milliseconds - Time in milliseconds
• microseconds - Time in microseconds

Setting Units:

You can set the units for a motor instance during initialization or at runtime:

```
# During initialization
motor = servomotor.M3(
    alias='motor1',
    position_unit='degrees',
    velocity_unit='rpm',
    acceleration_unit='rpm_per_second'
)

# At runtime
motor.set_position_unit('radians')
motor.set_velocity_unit('rotations_per_second')
```

Error Handling

The servomotor has comprehensive error detection and handling. If an error condition is detected then a fatal error condition will result. When this happens, the motor will immediately disable itself and the red LED will flash. The flashing LED will indicate the error code. You can count the pulses. You can also retrieve the error using the "Get Status" command. Once you know the error code, you can look it up in the section below to understand the root reason. The servomotor will not respond to most commands when it is in a fatal error state, only "Get Status" and "System Reset". You will need to reset the servomotor to get it back into a functional state. With careful programming, a fatal error should not be triggered. Nearly in all cases, if a fatal error occurs, it is for a good reason and most likely you will need to improve the way you try to use the motor.

Error Codes

This section lists all possible error codes that can be returned by the servomotor.

Error 1: ERROR_TIME_WENT_BACKWARDS

Short Description: time went backwards

Description: The system detected that time appeared to move backwards, which indicates a problem with the microsecond clock

Possible Causes:
- Timer overflow
- Hardware clock malfunction
- Software bug in time tracking

Solutions:
- Reset the device
- Check for timer configuration issues
- Verify timer interrupt priorities

Error 2: ERROR_FLASH_UNLOCK_FAIL

Short Description: flash unlock fail

Description: Failed to unlock the flash memory for writing

Possible Causes:
- Flash memory is locked
- Incorrect unlock sequence
- Hardware protection enabled

Solutions:
- Reset the device
- Check flash memory protection settings
- Verify flash unlock sequence in code

Error 3: ERROR_FLASH_WRITE_FAIL

Short Description: flash write fail

Description: Failed to write data to flash memory

Possible Causes:

• Flash memory write protection active
• Write operation timeout
• Invalid flash address
• Insufficient power during write

Solutions:

• Check flash write protection settings
• Verify flash write timing
• Ensure stable power supply
• Check flash address calculations

Error 4: ERROR_TOO_MANY_BYTES

Short Description: too many bytes

Description: Data size exceeds buffer or memory limits

Possible Causes:

• Command data too large
• Buffer overflow attempt
• Incorrect data length calculation

Solutions:

• Reduce data size
• Split data into smaller chunks
• Check data length calculations
• Verify buffer size constraints

Error 5: ERROR_COMMAND_OVERFLOW

Short Description: command overflow

Description: Command buffer overflow detected

Possible Causes:

• Too many commands sent too quickly
• Command buffer full
• Command processing too slow

Solutions:

• Reduce command frequency
• Wait for command completion
• Implement command flow control
• Check command processing speed

Error 6: ERROR_COMMAND_TOO_LONG

Short Description: command too long

Description: Individual command exceeds maximum allowed length

Possible Causes:

• Command data exceeds limit
• Malformed command
• Protocol violation

Solutions:

• Reduce command length
• Split into multiple commands
• Check command format
• Verify protocol compliance

Error 7: ERROR_NOT_IN_OPEN_LOOP

Short Description: not in open loop

Description: An operation that requires open loop control mode was attempted while the motor was in a different control mode

Possible Causes:

• Attempting calibration while not in open loop mode
• Incorrect mode transition sequence

Solutions:

• Switch to open loop mode before calibration
• Reset motor control mode to open loop
• Check mode transition logic in code

Error 8: ERROR_QUEUE_NOT_EMPTY

Short Description: queue not empty

Description: An operation that requires an empty movement queue was attempted while the queue still had pending movements

Possible Causes:

• Starting calibration with pending movements
• Starting homing with pending movements
• Attempting mode change with active movements

Solutions:

• Wait for all movements to complete
• Clear the movement queue
• Call clear_the_queue_and_stop() before operation

Error 9: ERROR_HALL_SENSOR

Short Description: hall sensor error

Description: Hall sensor readings are invalid or out of expected range

Possible Causes:

- Hall sensor malfunction
- Sensor connection issues
- Magnetic interference
- Sensor power issues

Solutions:

- Check hall sensor connections
- Verify sensor power supply
- Shield from magnetic interference
- Replace faulty sensors

Error 10: ERROR_CALIBRATION_OVERFLOW

Short Description: calibration overflow

Description: Calibration data buffer overflow during motor calibration

Possible Causes:

- Too many calibration points collected
- Calibration buffer size exceeded
- Incorrect calibration sequence

Solutions:

- Check calibration data buffer size
- Verify calibration point collection logic
- Reset and retry calibration

Error 11: ERROR_NOT_ENOUGH_MINIMA_OR_MAXIMA

Short Description: not enough minima or maxima

Description: Insufficient number of peaks detected during hall sensor calibration

Possible Causes:

- Poor sensor signals
- Incorrect motor movement during calibration
- Sensor sensitivity issues
- Mechanical problems

Solutions:

- Check hall sensor signal quality
- Verify motor movement during calibration
- Adjust sensor sensitivity
- Check for mechanical binding

Error 12: ERROR_VIBRATION_FOUR_STEP

Short Description: vibration four step

Description: Error in four-step vibration sequence during motor testing

Possible Causes:

• Incorrect vibration sequence
• Motor movement obstruction
• Control timing issues

Solutions:

• Check vibration sequence timing
• Verify motor freedom of movement
• Check control loop timing

Error 13: ERROR_NOT_IN_CLOSED_LOOP

Short Description: not in closed loop

Description: An operation that requires closed loop control mode was attempted while in a different mode

Possible Causes:

• Attempting position control in wrong mode
• Incorrect mode transition
• Mode change during operation

Solutions:

• Switch to closed loop mode first
• Check mode transition sequence
• Verify control mode logic

Error 14: ERROR_OVERVOLTAGE

Short Description: overvoltage

Description: Motor supply voltage exceeds maximum allowed limit

Possible Causes:

• Power supply voltage too high
• Regenerative braking voltage spike
• Voltage measurement error

Solutions:

• Check power supply voltage
• Add regenerative braking resistor
• Verify voltage measurements
• Implement voltage limiting

Error 15: ERROR_ACCEL_TOO_HIGH

Short Description: accel too high

Description: The requested acceleration exceeds the maximum allowed acceleration limit

Possible Causes:

- Movement command with excessive acceleration
- Acceleration parameter too high
- Incorrect acceleration calculation

Solutions:

- Reduce the requested acceleration
- Check acceleration calculations
- Verify max_acceleration setting

Error 16: ERROR_VEL_TOO_HIGH

Short Description: vel too high

Description: The requested velocity exceeds the maximum allowed velocity limit

Possible Causes:

- Movement command with excessive velocity
- Velocity parameter too high
- Acceleration leading to excessive velocity

Solutions:

- Reduce the requested velocity
- Check velocity calculations
- Verify max_velocity setting

Error 17: ERROR_QUEUE_IS_FULL

Short Description: queue is full

Description: Movement queue is full and cannot accept more commands

Possible Causes:

- Too many queued movements
- Commands sent too quickly
- Queue not being processed

Solutions:

- Wait for queue space
- Reduce command frequency
- Check queue processing
- Clear unnecessary movements

Error 18: ERROR_RUN_OUT_OF_QUEUE_ITEMS

Short Description: run out of queue items

Description: Movement queue unexpectedly empty during operation

Possible Causes:

• Movement sequence incomplete
• Queue processing error
• Timing synchronization issue

Solutions:

• Check movement sequence completeness
• Verify queue processing logic
• Ensure proper movement timing

Error 19: ERROR_MOTOR_BUSY

Short Description: motor busy

Description: An operation was attempted while the motor was busy with another operation like calibration or homing

Possible Causes:

• Attempting new movement during calibration
• Starting operation while homing active
• Multiple concurrent motor operations

Solutions:

• Wait for current operation to complete
• Check motor_busy flag before operations
• Implement proper operation sequencing

Error 20: ERROR_CAPTURE_PAYLOAD_TOO_BIG

Short Description: too much capture data

Description: The protocol used to communication over RS485 supports a total packet size that cannot be larger than 65535 bytes. The maximum payload size is just a bit less than this (about 65530 bytes). We have requested to capture hall sensor data that is larger than this maximum payload size.

Possible Causes:

• We called the Capture hall sensor data command and gave it innapropriate parameters. The two relevant parameters are: (1) Number of points to read back from the device and (2) Channels to capture bitmask. The number of bits set in this second parameter multipled by two bytes per value multiplied by this first parameter determines the payload size.

Solutions:

• Decrease the number of points to read back or disable reading back some of the channels. For instance, setting a bistmask to 1 will read back just one channel and that will cost two bytes per point.

Error 21: ERROR_CAPTURE_OVERFLOW

Short Description: capture overflow

Description: The data could not be transmitted back over RS485 fast enough to keep up with the rate of capturing the data

Possible Causes:

• We called the Capture hall sensor data command and gave it innapropriate parameters

Solutions:

• You can do one of the following: (1) Disable some of the channels that you wish to capture by setting bits in the bitmask parameter to 0, or (2) You can capture less often by increasing the time steps per capture parameter, or (3) average more sameples together by increasing the number of samples to sum parameter

Error 22: ERROR_CURRENT_SENSOR_FAILED

Short Description: current sensor failed

Description: The motor current sensor readings are outside expected baseline range

Possible Causes:

• Current sensor hardware failure
• Incorrect sensor calibration
• Wiring issues
• Power supply problems

Solutions:

• Check current sensor connections
• Verify power supply voltage
• Calibrate current sensor
• Check for hardware damage

Error 23: ERROR_MAX_PWM_VOLTAGE_TOO_HIGH

Short Description: max pwm voltage too high

Description: Configured maximum PWM voltage exceeds hardware limits

Possible Causes:

• Incorrect PWM voltage setting
• Configuration error
• Hardware limitation violation

Solutions:

• Reduce max PWM voltage setting
• Check voltage configurations
• Verify hardware specifications

Error 24: ERROR_MULTIMOVE_MORE_THAN_32_MOVES

Short Description: multi-move more than 32 moves

Description: Multi-move sequence exceeds maximum allowed number of moves

Possible Causes:

- Too many moves in sequence
- Movement splitting error
- Command sequence too long

Solutions:

- Reduce number of moves
- Split into multiple sequences
- Optimize movement planning

Error 25: ERROR_SAFETY_LIMIT_EXCEEDED

Short Description: safety limit exceeded

Description: The motor position has exceeded the configured safety limits

Possible Causes:

- Movement command beyond safety bounds
- Incorrect safety limit configuration
- Motor overshooting target position

Solutions:

- Adjust movement within safety limits
- Check safety limit settings
- Verify position calculations
- Implement proper deceleration

Error 26: ERROR_TURN_POINT_OUT_OF_SAFETY_ZONE

Short Description: turn point out of safety zone

Description: Calculated movement turn point exceeds safety zone limits

Possible Causes:

- Movement trajectory error
- Safety zone misconfiguration
- Trajectory calculation error

Solutions:

- Adjust movement trajectory
- Check safety zone settings
- Verify turn point calculations

Error 27: ERROR_PREDICTED_POSITION_OUT_OF_SAFETY_ZONE

Short Description: predicted position out of safety zone

Description: Movement trajectory would exceed safety zone limits

Possible Causes:

- Movement command too large
- Safety zone too restrictive
- Position prediction error

Solutions:

- Reduce movement distance
- Check safety zone limits
- Verify position predictions

Error 28: ERROR_PREDICTED_VELOCITY_TOO_HIGH

Short Description: predicted velocity too high

Description: Movement would result in velocity exceeding limits

Possible Causes:

- Acceleration too high
- Movement distance too large
- Velocity limit too low

Solutions:

- Reduce acceleration
- Increase movement time
- Check velocity limits

Error 29: ERROR_DEBUG1

Short Description: debug1

Description: Debug error used for development testing

Possible Causes:

- Development testing
- Debug condition met

Solutions:

- Check debug conditions
- Contact developer

Error 30: ERROR_CONTROL_LOOP_TOOK_TOO_LONG

Short Description: control loop took too long

Description: The motor control loop calculations exceeded the maximum allowed time

Possible Causes:

- Complex calculations taking too long
- Interrupt conflicts
- System overload

Solutions:

- Optimize control loop code
- Check interrupt priorities
- Reduce control loop complexity
- Verify timer configurations

Error 31: ERROR_INDEX_OUT_OF_RANGE

Short Description: index out of range

Description: Array or buffer index exceeds valid range

Possible Causes:

• Buffer overflow attempt
• Index calculation error
• Memory access violation

Solutions:

• Check index calculations
• Verify buffer sizes
• Add range checking

Error 32: ERROR_CANT_PULSE_WHEN_INTERVALS_ACTIVE

Short Description: can't pulse when intervals are active

Description: Pulse command attempted while interval timing active

Possible Causes:

• Timing conflict
• Incorrect pulse sequencing
• Mode conflict

Solutions:

• Wait for intervals to complete
• Check timing sequence
• Verify mode compatibility

Error 33: ERROR_INVALID_RUN_MODE

Short Description: invalid run mode

Description: Attempted to set invalid motor run mode

Possible Causes:

• Invalid mode selection
• Mode transition error
• Configuration error

Solutions:

• Check mode selection
• Verify mode transitions
• Use valid run modes

Error 34: ERROR_PARAMETER_OUT_OF_RANGE

Short Description: parameter out of range

Description: Configuration parameter exceeds valid range

Possible Causes:

- Invalid parameter value
- Configuration error
- Calculation error

Solutions:

- Check parameter values
- Verify configuration
- Use valid parameter ranges

Error 35: ERROR_DISABLE_MOSFETS_FIRST

Short Description: disable MOSFETs first

Description: Operation requires MOSFETs to be disabled first

Possible Causes:

- MOSFETs still enabled
- Incorrect operation sequence
- Safety check failure

Solutions:

- Disable MOSFETs before operation
- Check operation sequence
- Verify MOSFET control

Error 36: ERROR_FRAMING

Short Description: framing error

Description: Communication framing error detected

Possible Causes:

- Serial communication error
- Incorrect baud rate
- Signal integrity issues

Solutions:

- Check communication settings
- Verify baud rate
- Check signal integrity

Error 37: ERROR_OVERRUN

Short Description: overrun error

Description: Communication buffer overrun detected

Possible Causes:

• Data received too quickly
• Buffer processing too slow
• Flow control issues

Solutions:

• Implement flow control
• Increase buffer size
• Optimize data processing

Error 38: ERROR_NOISE

Short Description: noise error

Description: Communication noise detected

Possible Causes:

• Electrical interference
• Poor signal quality
• Wiring issues

Solutions:

• Check wiring shielding
• Reduce interference
• Improve signal quality

Error 39: ERROR_GO_TO_CLOSED_LOOP_FAILED

Short Description: go to closed loop failed

Description: Failed to transition from open loop to closed loop control mode

Possible Causes:

• Poor signal quality during transition
• Incorrect phase detection
• Motor position instability

Solutions:

• Check hall sensor signals
• Verify motor phase sequence
• Ensure stable motor position before transition
• Calibrate motor before transition

Error 40: ERROR_OVERHEAT

Short Description: overheat

Description: Motor or controller temperature too high

Possible Causes:

• Excessive motor current
• Insufficient cooling
• High ambient temperature
• Mechanical binding

Solutions:

• Reduce motor load
• Improve cooling
• Check for mechanical issues
• Allow system to cool

Error 41: ERROR_TEST_MODE_ACTIVE

Short Description: test mode active

Description: Operation not allowed while test mode is active

Possible Causes:

• Test mode enabled
• Mode conflict
• Incorrect operation sequence

Solutions:

• Disable test mode
• Complete test sequence
• Reset controller

Error 42: ERROR_POSITION_DISCREPANCY

Short Description: position discrepancy

Description: Mismatch between expected and actual position after movement

Possible Causes:

• Position tracking error
• Movement calculation error
• Mechanical issues

Solutions:

• Check position calculations
• Verify movement execution
• Calibrate system

Error 43: ERROR_OVERCURRENT

Short Description: overcurrent

Description: Motor current exceeds maximum allowed limit

Possible Causes:

- Excessive motor load
- Short circuit
- Motor binding
- Incorrect current limit

Solutions:

- Reduce motor load
- Check for shorts
- Verify current limits
- Check mechanical system

Error 44: ERROR_PWM_TOO_HIGH

Short Description: PWM too high

Description: The calculated PWM duty cycle exceeds the maximum allowed value

Possible Causes:

- PID controller output too high
- Incorrect PWM calculations
- Motor load too high

Solutions:

- Adjust PID parameters
- Check PWM calculation logic
- Verify motor load conditions
- Reduce movement demands

Error 45: ERROR_POSITION_DEVIATION_TOO_LARGE

Short Description: position deviation too large

Description: The difference between commanded position and actual position exceeds allowed limits

Possible Causes:

- Mechanical obstruction
- Motor stall
- Insufficient torque
- Incorrect PID tuning

Solutions:

- Check for mechanical blockages
- Verify motor power
- Adjust PID parameters
- Reduce acceleration/velocity demands

Error 46: ERROR_MOVE_TOO_FAR

Short Description: move too far

Description: Requested movement distance exceeds system limits

Possible Causes:

• Movement command too large
• Position calculation overflow
• Invalid target position

Solutions:

• Reduce movement distance
• Split into smaller moves
• Check position calculations

Error 47: ERROR_HALL_POSITION_DELTA_TOO_LARGE

Short Description: hall position delta too large

Description: The change in hall sensor position between updates is larger than expected

Possible Causes:

• Hall sensor noise
• Sensor malfunction
• Motor moving too fast
• Incorrect sensor readings

Solutions:

• Check hall sensor connections
• Verify sensor signal quality
• Reduce motor speed
• Check for electromagnetic interference

Error 48: ERROR_INVALID_FIRST_BYTE

Short Description: invalid first byte format

Description: The received first byte does not have the least significant bit set to 1

Possible Causes:

• Incorrect first byte encoding of the communication packet
• The baud rate mismatches
• Communication protocol mismatch
• Data corruption during transmission
• Sender not compliant with protocol requirements

Solutions:

• Ensure all first bytes have LSB set to 1
• Verify baud rates are the same on both ends
• Update sender software to comply with protocol
• Check for transmission errors
• Verify first byte encoding logic

Error 49: ERROR_CAPTURE_BAD_PARAMETERS

Short Description: capture bad parameters

Description: Some invalid parameters were given when invoking the Capture hall sensor data command

Possible Causes:

• A capture type other than 1, 2, or 3 was given
• The number of points to be captures was given as 0, which implies that you do not want any data
• The number of time steps per sample was given as 0, which implies an infinite sampling rate
• The number of samples to sum together was given as 0, which implies that you do not want any data
• The division factor was given as 0, which implies that you want to divide by 0, which implies that you do not remember high school math
• The bitmask that specifies the channels you want to capture was not valid. You either chose no channels or you chose channels that don't exits.

Solutions:

• Give 1, 2, or 3 as the capture type
• Give a value that is 1 or higher
• Give a value that is 1 or higher
• Give a value that is 1 or higher
• Give a value that is 1 or higher
• Normally, 7 specifies that all three channels should be captured. If you want to capture just one channel, choose either 1 (first channel only), 2 (second channel only), or 4 (third channel only).

Error 50: ERROR_BAD_ALIAS

Short Description: bad alias

Description: You tried to set the device to one of the reserved aliases such as 254, 253, or 252

Possible Causes:

• You tried to set the device to one of the reserved aliases such as 254, 253, or 252, which is not allowed. These aliases have special purposes.

Solutions:

• When using the Set device alias command, make sure to use an allowed alias, which is 0 to 251. You can also set the alias to 255, which is the broadcast alias, which is similar like having no alias set.

Error 51: ERROR_COMMAND_SIZE_WRONG

Short Description: command size wrong

Description: The payload sent with the command does not match the expected size of the input parameters

Possible Causes:

• You have sent a command that does not have the expected parameters or sizes of parameters
• Data was corrupted somehow during transmission over RS485 and you do not have CRC32 enabled to ensure integrity
• You have not used the communication protocol correctly. For instance, you did not encode the size of the packet correctly.

Solutions:

• Make sure to send the correct parameters with all the correct size, Look at motor_commands.json for the details about every command and its parameters
• Ensure good connections and grounding on the communication line and use CRC32 to ensure packet integrity
• Check the documentation about the protocol and make sure that the packet you are sending is correctly formatted and that the packet size is right

Error 52: ERROR_INVALID_FLASH_PAGE

Short Description: invalid flash page

Description: You are sending a firmware packet and the specified firmware page to be written is not valid and out of range

Possible Causes:

• You have sent a firmware upgrade packet and the specified firmware page is not one that can be written. Maybe the firmware size that you are trying to write is too big?
• The packet is encoded wrong

Solutions:

• Make sure to send the correct firmware page numbers and ensure the firmware is not too big. You can look at the FIRST_FIRMWARE_PAGE_NUMBER and LAST_FIRMWARE_PAGE_NUMBER settings in settings.h
• Study the protocol and make sure that everything is encoded right

Error 53: ERROR_INVALID_TEST_MODE

Short Description: invalid test mode

Description: You tried to trigger a certain test mode but it is not a valid one

Possible Causes:

• You are a user and doing something you should not and generally don't understand what you are getting yourself into
• You are a developer and you did not study the source code enough yet to use the test modes successfully

Solutions:

• Done call the Test mode command
• Keep studying the source code until you know how this works