TECHNICAL NOTE 260609

Maximizing Streaming Speeds of ORCA™ Motors

Revision 260609
irisdynamics.com | +1 (888) 995 -7050

Maximizing Streaming Speeds of ORCA™ Motors

Definitions

- **Control Loop:** The ORCA motor uses closed-loop control (also known as feedback control), which acts in a continuous cycle of measuring, deciding, and adjusting the system to achieve the desired state [1, 2]. An ORCA motor uses a control loop which operates up to 3 kHz, meaning there are approximately 0.33 ms between updates, as 1 Hz indicates a cycle per second. This is the rate at which a complete set of new data is available for position, force, and other values.

- **Baud Rate**: Represents the total frequency of symbols sent per second over a communication channel [3]. For ORCA motors, increasing the baud rate (typically up to 1 Mbps) helps to reduce data transmission latency. Learn more here: *TN250613: Achieving High-Speed Serial Communication.*

- **Software Execution Loop Rate:** The speed at which the software interfacing with the ORCA motor runs its complete processing loop, including issuing read or write commands. Learn more here: *Low-Latency and Real-Time Response in Smart Linear Actuators.*

- **Data Streaming:** The continuous real-time transmission of data [4]. ORCA motors provide low-latency and high-bandwidth access to force, position, voltage, and error state, among other metrics.

- **Data Logging:** The systematic capture, storage, and presentation of datasets. This may involve a standalone, wireless, or PC-based data logger. ORCA motors offer integrated data logging capabilities through built-in position and force sensing, with high-speed communication rates of up to 2.25 kHz. Learn more here: Integrated Data Acquisition & Logging in Smart Linear Motors for Industry 4.0.

Which Platforms Offer the Highest Software Execution Loop Rates?

Typical performance that can be expected using various forms of software is outlined in the following table:

Achievable Rate	Host Platform / Software Environment	Interface / Protocol Requirements
> 2 kHz	Embedded Device	Native UART via UART-to-RS422 converter or UART-RS485
Up to 1 kHz	orcaSDK (C++) pyorcasdk (Python)	Modbus RTU RS422 USB-to-Serial (with 1 ms latency)
	PLCs via ORCA-EIP	EtherNet/IP
~ 200 Hz	PLCs	Modbus RTU (RS422 or RS485 Full-Duplex Configuration)
~ 100 Hz	MATLAB	Modbus RTU RS422 USB-to-Serial (with 1 ms latency)
	LabVIEW	

How Can I Maximize Data Streaming and Software Execution Rates?

Update the Latency Timer in Windows

If using a serial port connected through USB, the latency timer defaults to 16 ms in Windows. This can be reduced to as low as 1 ms.

1. Search for **Device Manager**.
2. Expand the **Ports (Serial & LPT)** drop-down menu.
3. Determine the Serial port being used for Modbus communications. (If you are unsure, unplugging and replugging the cable is an easy way to identify it).
4. Right click the associated Serial port, and select **Properties**.

5. Select the **Port Settings** tab.
6. Click **Advanced**, select **Latency Timer (msec)** and set it to 1 ms.
7. Click **OK** to save the changes.
8. Restarting your computer may be required for the changes to take effect.

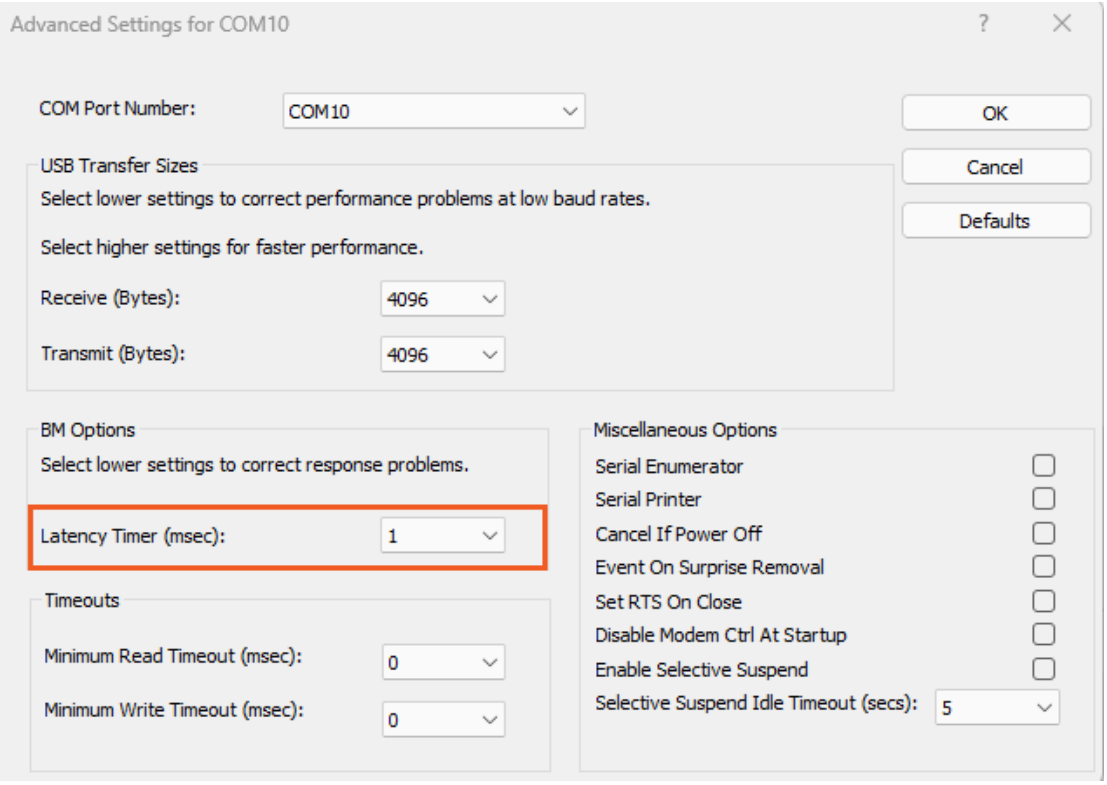

Fig. 1. Setting the Latency Timer to 1 (msec) on Windows.

Update the Latency Timer In Linux

IrisControls does not yet offer Linux support. Therefore only the RS422 Modbus RTU Serial port is relevant for the steps below:

To adjust the latency timer for high-speed Modbus RTU communications:

1. Verify Serial Device Enumeration

Check if the system detects the FTDI adapter by running:

```shell
ls -l /dev/ttyUSB*
sudo dmesg | grep -i ftdi
```

Expected Output: The serial port(s) should be listed, along with lines confirming the FTDI USB Serial converter is attached (e.g. to ttyUSB0). The terminal will output the applicable serial ports listed with its read and write permissions, and the current date and time. (*Note*: the following results occur when working with an ORCA USB):

2. Create a Custom udev Rule

To automatically set the latency timer to 1ms whenever the device is plugged in, find your device's vendor and product ID:

```shell
udevadm info -a -n /dev/ttyUSB0 | grep -E 'idVendor|idProduct'
```

Identify the first pair of values:

```
                    :~$ ls -l /dev/ttyUSB*
crw-rw---- 1 root dialout 188, 0 Jun  9 15:31 /dev/ttyUSB0
crw-rw---- 1 root dialout 188, 1 Jun  9 15:31 /dev/ttyUSB1
                    :~$ sudo dmesg | grep -i ftdi
[sudo] password for preludeinc:
[   76.396982] usb 2-1: Manufacturer: FTDI
[   76.430679] usbcore: registered new interface driver ftdi_sio
[   76.431628] usbserial: USB Serial support registered for FTDI USB Serial Device
[   76.431747] ftdi_sio 2-1:1.0: FTDI USB Serial Device converter detected
[   76.440708] usb 2-1: FTDI USB Serial Device converter now attached to ttyUSB0
[   76.440751] ftdi_sio 2-1:1.1: FTDI USB Serial Device converter detected
[   76.450073] usb 2-1: FTDI USB Serial Device converter now attached to ttyUSB1
                     ~$ udevadm info -a -n /dev/ttyUSB0 | grep -E 'idVendor | idProduct'
                     ~$ udevadm info -a -n /dev/ttyUSB0 | grep -E 'idVendor|idProduct'

    ATTRS{idProduct}=="6010"
    ATTRS{idVendor}=="0403"
    ATTRS{idProduct}=="0002"
    ATTRS{idVendor}=="1d6b"
```

Fig. 2. The output for the steps up until this point including the idProduct and idVendor of the FTDI adapter. Use the first pair of values for the following steps.

Create a new udev rule file:

```
Shell
nano /etc/udev/rules.d/99-orca.rules
```

Paste the following configuration into the file, ensuring the IDs match your device:

```
Shell
# Set FTDI Latency timer
DRIVER=="ftdi_sio", \
ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6010"
ATTR{latency_timer}="1"
```

3. Apply and Verify the Rules

Reload the udev logic to recognize the new file and trigger the changes:

```
Shell
sudo udevadm control --reload-rules && sudo udevadm trigger
```

Verify that the low-latency configuration was successfully applied to the device:

```
Shell
cat /sys/bus/usb-serial/devices/ttyUSB0/latency_timer
```

If the steps above were successful, the terminal should return a value of 1, as shown below.

```
                    ~$ cat /sys/bus/usb-serial/devices/ttyUSB0/latency_timer
1
```

Fig. 3. Indication that low-latency settings have been configured successfully.

Testing Streaming Speeds

Streaming speeds can be tested by running the test script shown in Figure 4:

Note: pyorcasdk will need to be installed as a prerequisite on a Linux system or virtual

machine.

```python
from pyorcasdk import Actuator

motor = Actuator("ORCA")

motor.open_serial_port("/dev/ttyUSB0", 1000000)

while True:
    print(f'Current Position: {motor.get_position_um().value}        ')
```

Fig. 4. pyorcasdk code to test if high-speed streaming has been configured successfully.

Expected Output:
The position should update rapidly with a new value. Try putting light pressure on the shaft of the ORCA motor to watch the position value update with the applied motion.

If you have entered the incorrect Serial port value, the position will repeatedly return a value of zero.

If you have access to a Windows environment, the Modbus rate can also be verified in IrisControls. If the settings have been configured correctly, you should see a Modbus rate of approximately 1 kHz, as shown in Figure 5:

Fig. 5. Modbus Rate in IrisControls after configuring high-speed streaming using low latency and increased baud rate.

Improving Code Performance

Use Streaming Commands over Blocking Calls

Minimize the use of blocking read or write calls wherever possible. Instead, favour streaming commands, which provide more data per message than blocking read and write calls.

Figure 6 illustrates the difference between these two approaches:

Streaming vs Blocking Calls

```
# Asynchronous: Pulls the latest position data from the data stream
streamed_pos = motor.get_stream_data().position

# Synchronous/Blocking: Requests position data from a specific register
POS_CMD = 30  # Register for actuator position in micrometers (lower two bytes)

# Reading the wide register provides data from both position registers (30 -31)
blocking_pos = motor.read_wide_register_blocking(POS_CMD).value
```

Fig. 6. Comparison between a streaming and blocking call in the pyorcasdk library.

Use of time.sleep() or Delays

The function time.sleep() acts as a blocking call, meaning that the program entirely pauses for its duration if it is passed a value larger than zero [5].

Using time.sleep() inside a high-frequency execution loop (such as in Position Mode) can trigger a Communications Timeout error because of the strict timing demands of high-rate communication.

If a sleep interval is required for your application, try decreasing the sleep duration or moving the delay outside of the frequently executing loop.

References

[1] https://www.sciencedirect.com/topics/engineering/closed-loop-control
[2] Closed Loop Control
[3] https://dipsingh.github.io/Back-to-basics-communication/
[4] https://www.splunk.com/en_us/blog/learn/data-streaming.html
[5] https://www.digitalocean.com/community/tutorials/python-time-sleep