IRISSDK ORCA ™ API

User Guide 211201

Version 1.6

This document applied to the following Orca Series motor firmware:

- 6.1.4
- 6.1.5
- 6.1.6
- 6.1.7

Notes for firmware versions earlier than 6.1.6: Force returned uses a legacy unit instead of mN. The conversion rate between mN and legacy units is different for each motor type (1 legacy unit ≈ 10mN for orca 6-24, but 1 legacy unit ≈ 21mN for orca 6-48 and 1lu ≈ 23.3mN for 15-48

Prior to 6.1.6 functions related to kinematic controller are not available

Prior to 6.1.7 function related to haptic controller and different streaming modes are not available

For more recent firmware versions, please download the latest version of this user guide at https://irisdynamics.com/downloads

CONTENTS

Iris Dynamics Ltd. Victoria, British Columbia T +1 (888) 995-7050 F +1 (250) 984-0706 www.irisdynamics.com

Iris Dynamics Ltd. Victoria, British Columbia T +1 (888) 995-7050 F +1 (250) 984-0706 www.irisdynamics.com

REVISION HISTORY

Version	Date	Author	Reason
1.0	December, 2021	KE, RM, KH	Initial Release
1.1	April, 2022	RM	Additional function revision
1.2	June, 2022	SW, RM	Additional and updated function description, reordering of information. Changes to heading layer and table compressing
1.3	June, 2022	RM	Remove references to private functions/properties formatting, error descriptions Clarify Actuator for object reference motor for device reference
1.4	August, 2022	KC	Title update, code section formatting
1.5	March, 2023	RM	Add kinematic mode option. Add additional functions. Change order of functions to match actuator.h,
1.6	April, 2023	AB, RM	Included descriptions of changes from 6.1.5 to 6.1.6 Multiple streaming mode options

ACTUATOR CLASS OVERVIEW

The Actuator object is used to establish and maintain a communication stream with an Orca™ Series linear motor. This object abstracts the Modbus RTU communications to control the motor.

Functions are available to command motors as desired and receive information from the motor without having to interact with the serial communications protocol directly. This object can also be used to manage a high speed stream of Modbus messages.

For a complete list of Orca Series motor available registers and complete details on motor behaviour, see the Orca Series Motor Reference Manual.

Public Types

Type	Description		
enum	*StreamMode* Sets the type of command that will be sent on when high speed stream is enabled.		
	Name	Value	Description
	MotorCommand	0	Sets the high speed stream command to send a motor command function code.
	MotorRead	1	Sets the high speed stream command to send motor read function code
	MotorWrite	2	Sets the high speed stream command to send motor write function code
enum	*MotorMode* Different modes of operations available to the motor, effects the type of motor command that can be streamed to the motor.		
	Name	Value	Description
	SleepMode	1	Puts motor in “sleep” (electrical brake)
	ForceMode	2	Motor command writes to ForceControl register
	PositionMode	3	Motor command writes to PositionControl register
	HapticMode	4	Puts motor into haptic mode
	KinematicMode	5	Puts motor into kinematic)
enum	*HapticEffect* Bits related to haptic effects		
	Name	Value	Description
	ConstF	1 << 0	Constant force effect
	Spring0	1 << 1	First spring effect
	Spring1	1 << 2	Second spring effect
	Spring2	1 << 3	Third spring effect
	Damper	1 << 4	Damper effect
	Inertia	1 << 5	Inertia effect
	Osc0	1 << 6	First oscillator effect (vibration)
	Osc1	1 << 7	Second oscillator effect (vibration)

struct

ConnectionConfig

Contains parameters used to configure aspects of the handshake and connection maintenance with the motor.

Variable	Type	Default	Description
server_address	u8	1	The motor Modbus Server ID
req_num_discovery_ping	s32	15	The number of successful ping message responses required to move to the next step of the handshake.
max_consec_failed_msgs	s32	5	The number of consecutive failed responses to trigger a disconnect
target_baud_rate_bps	u32	625000	Baud rate requested for stream connection once handshake has been completed.
target_delay_us	u16	80	Delay between received message and next outgoing message once a connection has been established.
response_timeout_us	u32	8000	The time to wait for a response to a sent message once a connection has been established.

Public Functions

Return Type	Function Call and Description
void	*set_mode(MotorMode mode)* Change the Actuator Object's motor mode, which determines the type of commands being streamed by the motor when enabled. This command will also inject a message to write to the motor's mode control register.
MotorMode	*get_mode()* Returns the current motor mode of the Actuator Object. Used to determine which type of command is being streamed to the motor.
void	*set_stream_mode(StreamMode mode)* Set the type of high speed stream to be sent on run out once handshake is complete
StreamMode	*get_stream_mode()* Get the current stream type to be sent on run out once handshake is complete
void	*update_write_stream(u8 width, u16 reg_addr, u32 reg_value)* This function will update the values being sent when in motor write stream mode. This function can be continuously called.
void	*update_read_stream(u8 width, u16 reg_addr)* This function will update the values being sent when in motor read stream mode. This function can be continuously called.
void	*set_force_mN(s32 force)* Set/adjust the force, in millinewtons. Written to the motor's Force Control Register when sending motor command stream and mode is set to Force Mode.
void	*set_position_um(s32 position)* Set/adjust the position, in micrometers. Written to the motor's Position Control Register when sending motor command stream and mode is set to Position Mode.
s32	*get_force_mN()* Returns the force sensed by the motor, in milliNewtons.
s32	*get_position_um()* Returns the position of the shaft in the motor (distance from the zero position) in micrometers.
void	*enable_haptic_effects(u16 effects)* Write to the HAPTIC_STATUS register to set which haptic effects will be enabled when the motor is in HapticMode. Bits to enable each effect are defined in the HapticEffect enum.
bool	*new_data()* Determines if new data has been received from the motor since the function was last called. Returns true if new data has been written to the Actuator object's local copy of the motor's memory map, returns false otherwise.
void	*set_stream_timeout(u32 timeout_us)*

	Set the maximum time between calls to set_force or set_position, in force or position mode respectively, before timing out and returning to Sleep Mode.
void	*init()* Initializes modbus communication between the Orca Motor and the controller's communication port, using the default baud rate (19200 Bps).
u16	*get_num_successful_msgs()* Returns the number of successfully received messages, that have been sent and responded to without error.
u16	*get_num_failed_msgs()* Returns the number of failed messages, that have either timed out without a response or have received an incomplete or inaccurate response.
void	*run_out()* When a motor is enabled, this function dispatches transmissions for motor frames when connected and dispatches handshake messages when not. It must be called frequently. When a motor is not enabled this call will send out messages in the queue.
void	*run_in()* This function should be called as frequently as possible; it polls for timeouts and parses responses from the message queue. This function is used to maintain the connection state based on failed messages and parses successful messages.
void	*isr()* Provides access to the device driver's interrupt service routine function for linking to an interrupt handler.
const char *	*get_name()* Returns the name given to the Actuator object.
s32	*channel_number()* Returns the UART (or com port) channel number.
u16	*get_mode_of_operation()* Returns the mode of operation as currently updated in the local memory maps.
u16	*get_power_W()* Returns the amount of power being drawn by the motor, in Watts.
u8	*get_temperature_C()* Returns the temperature of the motor in Celsius.
u16	*get_voltage_mV()* Returns the amount of voltage the motor is receiving, in millivolts.
u16	*get_errors()* Returns the sum of all error messages currently sent by the motor. See Error Types for detailed descriptions of errors.
u32	*get_serial_number()* Returns the motor's serial number
u16	*get_major_version()* Returns the major version of the firmware on the motor.

Iris Dynamics Ltd. Victoria, British Columbia T +1 (888) 995-7050 F +1 (250) 984-0706 www.irisdynamics.com

u16	*get_release_state()* Returns the release state (minor version) of the firmware on the motor.
u16	*get_revision_number()* Returns the revision number of the firmware on the motor.
bool	*version_is_at_least(u8 version, u8 release_state, u8 revision_number)* Returns true if the motor's firmware version is 'at least as recent' as the version designated by the parameters, and false otherwise.
void	*zero_position()* Sends a single command to the motor to use its current position as the zero position.
void	*clear_errors()* Requests that all errors are cleared from the motor. Removes latched errors.
void	*get_latched_errors()* Copies the register holding the codes for latched errors from the motor's memory map into the local memory map. Latched errors are errors that were found but are no longer active.
void	*set_max_force(s32 max_force)* Set the maximum force that the motor will allow before clipping and tripping the Force Exceeded error.
void	*set_max_temp(u16 max_temp)* Set the maximum allowable temperature for the motor, in Celsius. Associated with the Temperature Exceeded error.
void	*set_max_power(u16 max_power)* Set the maximum allowable power for the motor, in Watts. Associated with the Power Exceeded error.
void	*set_pctrl_tune_softstart(u16 t_in_ms)* Sets the fade period when changing position controller tune in milliseconds.
void	*set_safety_damping(u16 d_gain)* Sets the motion damping gain value used when communications are interrupted.
void	*tune_position_controller(u16 p, u16 i, u16 dv, u32 sat, u16 de =0)* Sets the PID tuning values on the motor (proportional, integral, derivative, maximum force). The sat value determines the maximum force the motor will output while trying to reach the target position.
void	*set_kinematic_config(s8 num_motions, s8 trig_period = 0, s8 HW_trig = 0)* Set the number of motions to be configured. Can additionally sets the debounce and enable hardware trigger. Enabling the hardware trigger will disable modbus communications. The default number of motions is 1. (after v6.1.7 num motions will always be 32 and are not adjustable)
void	*set_kinematic_motion(s32 ID, s32 pos, s32 time, u16 delay, s8 type, s8 chain)* Sets the parameters to define a kinematic motion. Defines a single movement. Multiple movements can be defined and chained together.

void	*trigger_kinematic_motion(s32 ID)* Trigger the start of a kinematic motion with the specified ID. Any chained motions will also run.
void	*read_register(u16 register_address)* Request read of a register from the motor's memory map; this will update the Actuator object's local copy of the memory map once a response is received.
void	*read_registers(u16 register_address, u16 num_registers)* Request read of sequential registers from the motor's memory map; this will update the Actuator object's local copy of the memory map once a response is received.
void	*write_register(u16 register_address, u16 reg_data)* Request for a specific register in the motor's memory map to be updated with a given value.
void	*write_registers(u16 register_address, u16 num_registers, u16* reg_data)* Request for multiple registers in the motor's memory map to be updated with a given array of values.
u16	*get_orca_reg_content(u16 offset)* Returns the contents of the input register from the Actuator object's copy of the motor's memory map.

Iris Dynamics Ltd. Victoria, British Columbia T +1 (888) 995-7050 F +1 (250) 984-0706 www.irisdynamics.com

Inherited Functions

Return Type	Name and Description
s32	*set_connection_config(ConnectionConfig config)* Apply the handshake/connection configuration parameters passed in the ConnectionConfig struct. Return 0 if one of the parameters was invalid and default values were used, 1 otherwise.
bool	*is_connected()* Determine whether the device using the Actuator object has completed a successful handshake to connect with the motor.
bool	*is_enabled()* Determine if communication with the motor is enabled or not.
void	*enable()* Enable high speed communication with a motor. Allows the handshake sequence to begin and enables transceiver hardware.
void	*disable()* Disables high speed communication with the motor.
void	*disconnect()* Move to disconnected state of high speed connection and reset connection variables.

Windows Only Functions

Return Type	Name and Description
bool	*set_new_comport(s32 _comport)* Change the comport associated with the motor's modbus connection
void	*disable_comport ()* Close the comport associated with a motor's modbus connection

DETAILED DESCRIPTION

The Actuator object is used to establish and maintain a connection with an Orca motor. In this context, a connection is a consecutive stream of messages to and from the motor in which either a target force, a target position, or a sleep directive is commanded, and a response is received which provides information about the position, force, temperature, power, voltage, and errors.

Timing and framing for the stream are handled automatically by the Actuator class, with the user being simply required to call the class's run_in() and run_out() function regularly.

Current device information such as position, force, temperature, etc., can then be accessed using the provided get functions. Additional functions are available for configuration of the connection or of motor parameters, such as limiting maximum power draw, forces, temperatures, etc.

The purpose of this object is to encapsulate the MODBUS communication protocol, hiding it from the user, and abstracting the concept of an Actuator to allow the user to provide clear directives to an Orca motor.

To construct an instance of this object, pass the channel/port on your device that the corresponding motor will be connected to, as seen in the object use example.

Initializing the Object

Initializing is done by calling the init() function, which should be done before attempting any communication. This will set up the appropriate communication channel (UART or serial) with the appropriate timers and interrupts as required for the device.

Enabling and Disabling the High Speed Stream

The enabled status can be changed at any time by calling enable() and disable().

Any desired changes to the ConnectionConfig structure should be made prior to enabling as it will use the current values during the handshaking sequence when establishing a connection.

The enabled status of the object determines if a high speed communication stream with the motor will be attempted and can be determined by calling is_enabled().

When disabled, only injected commands will be sent. Configurations such as handshake parameters should be changed while the object is disabled.

When enabled, messages will be transmitted and received according to the connection status and communication mode.

Connection Status

The connection status can be checked by calling is_connected(). The connected state implies that valid high speed communication with the motor has been established and messages are being sent as determined by the Stream Mode (Orca firmware v6.1.7 or later, prior firmware will only use motor command messages when streaming).

The Actuator object is in the disconnected state upon initialization and remains in this state until it has completed a successful handshake sequence with the motor device.

When disconnected and enabled, the object sends regular pings to check for a motor device. By default, the baud rate used for pings is 19200 bps.

Iris Dynamics Ltd. Victoria, British Columbia T +1 (888) 995-7050 F +1 (250) 984-0706 www.irisdynamics.com

When connected, the Actuator object maintains a constant stream of messages to and from the motor. The content of this message stream depends on the communication mode, which can be changed by calling set_mode().

The motor will transition from connected to disconnected if several consecutive failed messages are detected. The number of failed messages which constitutes a disconnection can be modified by adjusting the max_consec_failed_msgs variable from the ConnectionConfig struct before calling set_connection_config().

Following a disconnect, the Actuator object will pause communications to allow the server to reset to the default baud rate and messaging delays, then resumes sending pings to attempt to re-establish communications.

Handshake Sequence

The Actuator object will reach the connected state after completing the steps of the handshake sequence. The sequence is managed automatically from the run_out() function and does not need to be invoked by the user. Below is a description of the class's behavior while connecting.

Step 1: Communication Check (Discovery)

Successful communication is established by receiving consecutive successful responses to a ping message which expect an echo response.

The number of required successful consecutive messages can be changed by adjusting the req_num_discovery_pings variable from the ConnectionConfig struct before calling set_connection_config().

Step 2: Register Contents Synchronization (Synchronization)

Next, a series of read register requests will be sent to update relevant sections of the local copy of the motor's register contents.

Step 3: Baud Rate and Messaging Delay Adjustment (Negotiation)

Lastly, a command will be sent to adjust the value of the baud rate and messaging delay registers in the actuator.

The new baud rate, if different than the default 19200bps, must be adjusted using the target_baud_rate_bps variable from the ConnectionConfig Struct and set_connection_config() method described previously. The new delay, if different from the default, must be adjusted, in microseconds, using the target_delay_us variable from the ConnectionConfig Struct and set_connection_config() method described previously.

Stream Modes (Orca v6.1.7 or later)

Motor Command

This stream mode will send messages to the Orca motor that will return information about the motor's status information position, force, power, temperature, voltage, and errors. If in Position or Force modes the target values that are set in set_force_mN() and set_position_um() will be commanded to the motor. Calling the functions to update the force and position will start a timer. In this streaming mode it is required that set_force_mN() and set_position_um() are called faster than the stream_trimeout_cycles which defaults to 100ms and can be adjust using the set_stream_timeout() function.

In other modes no specific commands are sent to the motor other than setting the mode of operation.

Motor Read

In this mode a specified register (either single or double wide) can be read from. The response to this stream will be the value in the in the register as well as the motor's mode of operation, position, force, power, temperature, voltage, and errors.

The register to read from can be updated through the update_read_stream() function.

Motor Write

In this mode a specified register (either single or double wide) can be written to. The response to this stream will be the motor's mode of operation, position, force, power, temperature, voltage, and errors.

The register to write to and the value to write from can be updated through the update_write_stream() function.

Motor Modes

When high stream streams are enabled the Actuator class will stream will be different depending on the selected motor mode.

To check or change the communication mode, use the get_mode() and set_mode() functions respectively.

Sleep Mode

The command sent in this mode puts the motor into "sleep" mode where the motor will only produce an electro-mechanical 'braking' force induced by shorting all its windings. No other force generation will be possible, regenerative braking is disabled, and motor power consumption will be minimized.

Force Mode

When in Force mode the motor will seek to reach its target force. This target will either come from the value set using the set_force_mN(s32 *force*) function when in Motor command stream mode; or the last value written to the FORCE_CMD register when in other streaming modes.

Position Mode

When in Position mode the motor will seek to reach its target position. This target will either come from the value set using the set_position_um(u32 *position*) function when in Motor Command stream mode; or the last value written to the POSITION_CMD register when in other streaming modes.

The motor uses PID control to achieve the target position, to tune the values used by the motor call the tune_position_controller(u16 *p*, u16 *i*, u16 *d*, u32 *sat*) function.

Haptic Mode

(Available with Orca firmware v6.1.7 or late)

The messages sent in motor command stream mode sent in this mode will put the motor into haptic mode without sending any additional commands. The configured effects can be enabled or disabled by using the enable_haptic_effects(uint16_t effects) function. The effects can either be configured can either be configured through the Orca motor's IrisControls GUI or by writing to the registers in the Haptic section of the memory map. If attempting to continuously update an effect parameter this should be done by setting the mode to Haptic mode and using the motor write stream mode to update the specified effect parameter.

Kinematic Mode

(Available with Orca firmware v6.1.6 or later)

The messages sent in motor command stream mode sent in this mode will put the motor into kinematic mode without sending any additional commands. This for injection messages to triggering of kinematic motions while maintaining the stream of returned data. In kinematic mode the position controller will be used to follow the targets specified by the kinematic controller. Motion IDs can either be configured through the Orca motor's IrisControls GUI or using set_kinematic_motion(s32 ID, s32 pos, s32 time, u16 delay, s8 type, s8 chain). Any configured motion can be triggered using the trigger_kinematic_motion(s32 ID) function.

Injecting Other Commands into Stream

Other messages can be injected into the stream of “Stream Command” messages when required. For example, the position could be zeroed, the position controller could be tuned, or individual registers in the memory map could be written to or read from.

The Actuator object will send any messages injected before continuing to send “Stream Command” messages and will do so while respecting the appropriate delays required as configured in ConnectionConfig structure.

The number of single commands that can be injected into the stream within a certain time frame is restricted by the size of the message buffer queue used by the MODBUS client serial layer. To adjust the size of the queue, adjust the NUM_MESSAGES definition in shared\mb_config.h to one of the preset options.

Accessing Retrieved Data

The position, power, force, temperature, voltage, and error data returned by the motor in each of the above modes can be retrieved using the following series of "get" functions.

- u16 get_orca_reg_content(u16 reg_address)
- s32 get_force_mN()
- s32 get_position_um()
- u16 get_power_W()
- u8 get_temperature_C()
- u16 get_voltage_mV()
- u16 get_errors()

Iris Dynamics Ltd. Victoria, British Columbia T +1 (888) 995-7050 F +1 (250) 984-0706 www.irisdynamics.com

Error Types

The motor will generate error codes when a user setting, or a device limit is reached or exceeded. Depending on the error, certain features will not be available until the error is cleared. Motor errors are communicated as a series of bit flags. There may be a combination of multiple errors: For example, error 320 would be a Temperature Exceeded (64) + a Power Exceeded (256). Provided the error does not persist it can be cleared using clear_errors().

Error	Mask	Trigger Level Registers	Modules disabled	Cleared By
Configuration Errors	1 (0x001F)	-	Position, Force	Calibration Routines
Force Clipping	32 (0x0020)	-	-	Automatically
Temperature Exceeded	64 (0x0040)	USER_MAX_TEMP MAX_TEMP	Position, Force, Calibration	Sleep Mode
Force Exceeded	128 (0x0080)	USER_MAX_FORCE	-	Automatically
Power Exceeded	256 (0x0100)	USER_MAX_POWER MAX_POWER	Position, Force	Sleep Mode
Shaft Image Failed	512 (0x0200)	-	Position, Force	Sleep Mode + Insert or Calibrate Shaft
Voltage Invalid	1024 (0x0400)	MIN_VOLTAGE MAX_VOLTAGE	Position, Force, Calibration	Sleep Mode + Providing a valid voltage source
Comms Timeout	2048 (0x0800)	USER_COMMS_TIMEOUT COMMS_TIMEOUT	Position, Force	Sleep Mode

Configuration Errors

These errors indicate calibrations or settings have not been done or have been made invalid.

Force Clipping

Requested force too large. This error has no effect on operation except to inform the user that linear force output has been compromised.

Temperature Exceeded

When the temperature of the stator windings or of the motor driver exceeds the device or user-set maximum.

Force Exceeded

When the measured force output of the motor exceeds the user-set force limit.

Power Exceeded

When the power burned in the stator exceeds the device or user-set maximum value.

If this error is experienced, either the maximum power user setting can be increased, or the maximum force user setting should be decreased. (set_max_power() or set_max_force())

If the position controller (*i.e.*position mode) is causing this error, the saturation level can also be decreased to prevent this error (tune_position_controller()).

Shaft Image Failed

If the shaft image is detected to be invalid, the shaft might not be inserted, it might be an invalid shaft for the device, or the device may require a calibration.

Communications Timeout

When in force or position mode, a steady stream of communications must be successfully received to avoid this error. Users can make the communications timeout shorter than the default setting by writing a non-zero value to the USER_COMMS_TIMEOUT register. This register has units of milliseconds.

Basic Object Use Example

Below is an example program that initializes the Actuator object, sets its connection parameters, and begins requesting a certain force from an initial position. The data returned is then retrieved and stored in a local variable for further analysis.

Below is a basic example of building and using a single Actuator object on channel 1.

```
#include "modbus_client/device_applications/actuator.h"

//Constructor(channel, name, cycles per microsecond)
Actuator motor(5, "Motor 1", 1);

s32 target_force;

int main(){
     motor.init();
     motor.enable();
     motor.set_mode(Actuator::ForceMode);

     while(1){
        update_target_value(); //some function to update the target
        motor.set_force_mN(target_force); //Update the force command sent to motor
        motor.run_in();   //Parse incoming messages
        motor.run_out();  //Send handshake and command messages to the motor
     }
}
```