ORCA™ API

Reference Manual

Version 1.0.3

CONTENTS

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

LIST OF FIGURES

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

INTRODUCTION

This manual provides directions and examples of how to use the Actuator object to control an Orca™ motor (linear actuator). Procedures for constructing the object, initializing parameters, connecting to the Orca device, and maintaining different modes of communication are discussed. Functions are available to command motors as desired and receive information from the motor without having to interact with the serial communications protocol.

ACTUATOR CLASS OVERVIEW

The Actuator object is used to establish and maintain a connection with an Orca™ motor.

Header:	#include "modbus_lib/client/device_applications/actuator.h"
Inherits:	IrisClientApplication

Public Types

Type	Description
enum	*CommunicationMode* Type of command that can be streamed to the motor.

Name	Value	Description
SleepMode	0x00	Puts motor in "sleep" (electrical brake)
ForceMode	0x02	Motor command writes to ForceControl register
PositionMode	0x04	Motor command writes to PositionControl register

struct — *ConnectionConfig*

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

Variable	Type	Default	Description
server_address	uint8_t	1	The MODBUS specified server address (1 - 247) given to the motor.
req_num_discovery_ping	int	15	The number of successful ping message responses required to move to the next step of the handshake.
max_consec_failed_msgs	int	5	The number of consecutive failed responses that will trigger a disconnect from the motor.
target_baud_rate_bps	uint32_t	625000	Baud rate requested for stream connection once handshake has been completed.
target_delay_us	uint16_t	80	Delay between received message and next outgoing message once a connection has been established.
response_timeout_us	uint32_t	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	*init()* Initializes modbus communication between the Orca Motor and the controller's UART port, using the default baud rate.
void	*set_mode(CommunicationMode mode)* Change the Actuator Object's communication mode, which determines the type of commands being streamed by the motor.
Communication Mode	*get_mode()* Returns the current communication mode of the Actuator Object. Used to determine which type of command is being streamed to the motor.
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	*run_out()* This functions dispatches transmissions for motor frames when connected and dispatches handshake messages when not. It must be called frequently. When using many motors, it may be useful to alternate this call to maximize message throughput.
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	*isr()* Provides access to the device driver's interrupt service routine function for linking to an interrupt handler.
uint16_t	*get_num_successful_msgs()* Returns the number of successfully received messages, that have been sent and responded to without error.
uint16_t	*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.
const char *	*get_name()* Returns the name given to the Actuator object.
int	*channel_number()* Returns the UART (or com port) channel number.
int32_t	*get_force_mN()* Returns the force sensed by the motor, in milliNewtons.
int32_t	*get_position_um()*

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

	Returns the position of the shaft in the motor (distance from the zero position) in micrometers.
uint16_t	*get_power_W()* Returns the amount of power being drawn by the motor, in Watts.
uint8_t	*get_temperature_C()* Returns the temperature of the motor in Celsius.
uint16_t	*get_voltage_mV()* Returns the amount of voltage the motor is receiving, in milliVolts.
uint16_t	*get_errors()* Returns the sum of all error messages currently sent by the motor. See Error Types for detailed descriptions of errors.
uint32_t	*get_serial_number()* Returns the motor's serial number
uint16_t	*get_major_version()* Returns the major version of the firmware on the motor.
uint16_t	*get_release_state()* Returns the release state (minor version) of the firmware on the motor.
uint16_t	*get_revision_number()* Returns the revision number of the firmware on the motor.
uint16_t	*get_build_id()* Returns the build ID of the firmware on the motor.
uint16_t	*get_orca_reg_content(uint16_t offset)* Returns the contents of the input register from the Actuator object's copy of the motor's memory map.
void	*set_force_mN(int32_t force)* Set/adjust the force, in milliNewtons. Written to the motor's Force Control Register when the communication mode is set to Force Mode.
void	*set_position_um(int32_t position)* Set/adjust the position, in micrometers. Written to the motor's Position Control Register, when the communication mode is set to Position Mode.
void	*set_stream_timeout(uint32_t 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	*set_max_force(int32 max_force)* Set the maximum allowable force for motor, in milliNewtons. Associated with the Force Exceeded error.
void	*set_max_temp(uint16_t max_temp)* Set the maximum allowable temperature for the motor, in Celsius. Associated with the Temperature Exceeded error.
void	*set_max_power(uint16_t max_power)* Set the maximum allowable power for the motor, in Watts. Associated with the Power Exceeded error.
void	*set_safety_damping(uint16_t d_gain)*

	Sets the motion damping gain value used when communications are interrupted.
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_pctrl_tune_softstart(uint16_t t_in_ms)* Sets the fade period when changing position controller tune in milliseconds.
void	*tune_position_controller(uint16_t p, uint16_t i, uint16_t d, uint32_t sat)* 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	*read_register(uint16_t register_address)* Request read a register from the motor's memory map; this will update the Actuator object's local copy of the memory map.
void	*write_register(uint16_t register_address, uint16_t reg_data)* Request for a specific register in the motor's memory map to be updated with a given value.

Inherited Functions

Return Type	Name and Description
int	*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 communication with a motor. Allows the handshake sequence to begin and enables transceiver hardware.
void	*disable()* Disable communication with the motor.
void	*disconnect()* Move to disconnected state and reset variables.

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 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 Object

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 communication with the motor will be attempted and can be determined by calling is_enabled().

When disabled, no messages will be sent, and incoming messages will be discarded. Transmitting hardware will be disabled and when possible, placed into a high-impedance state. 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. Transmitting hardware will be enabled and live.

Connection Status

The connection status can be checked by calling is_connected(). The connected state implies that valid communication with the motor has been established and streaming of commands can be accomplished.

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 19200bps.

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 a number of 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.

Object Use in the Disconnected State

When the object is disabled, it will not initiate any new messages. It may finish any pending messages and will parse any responses.

If the Actuator is disconnected, the values returned by the functions for retrieving data will not be relevant or accurate. Avoid calling these functions until is_connected() returns true.

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 command for establishing connection and adjusting these parameters follows the format described in tables 1 and 2.

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. For a table of accepted baud rates, messaging delays, and corresponding stream frequencies see the appendix.

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

Table 1. Manage High-speed Stream Request PDU

Name	# of Bytes	Value
Device Address	1 byte	0x01
Function Code	1 byte	0x41
State Command	2 bytes	0xFF00 (enable and apply parameters) 0x00 00(disable and return to default)
Baud rate	4 bytes	**State Command = 0** Ignored **State Command = 1** Target baud rate
Delay (us)	2 bytes	**State Command = 0** Ignored **State Command = 1** Target Response delay in microseconds

Table 2. Manage High-speed Stream Response PDU

Name	# of Bytes	Value
Device Address	1 byte	0x01
Function Code	1 byte	0x41
State Command	2 bytes	Echo of request
Baud rate	4 bytes	Realized baud rate
Delay (us)	2 bytes	Realized message delay in microseconds

Communication Modes

The Actuator class will stream a series of commands corresponding to the selected mode. The specific command for each mode follows the format described in tables 3 and 4 below.

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

Table 3: Stream Command Request PDU

Name	# of Bytes	Value
Device Address	1 byte	0x01
Function Code	1 byte	0x64
Command Address	1 byte	0x1C - Force Command 0x1E - Position Command All else - Sleep Command
Command Value	4 bytes	**Command Address = 1C** Force Command in milli-Newtons **Command Address = 1E** Position Command in micro-meters **Command Address = all else** Ignored
CRC	2 bytes	CRC-16 (Modbus) Polynomial 0xA001

Table 4: Stream Command Response PDU

Name	# of Bytes	Value
Device Address	1 byte	0x01
Function Code	1 byte	0x64
Position Value (um)	4 bytes	Shaft position in micro-meters
Force Value (mN)	4 bytes	Force realized in milli-Newtons
Power Value (W)	2 bytes	Power consumed in Watts
Temperature Value (C)	1 byte	Temperature value in degrees Celsius
Voltage Value (mV)	2 bytes	Supply Voltage in milli-Volts
Errors	2 bytes	Error register contents
CRC	2 bytes	CRC-16 (Modbus) Polynomial 0xA001

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

The command in the force mode will write to the motor's Force Control Register to adjust the force the motor will seek to achieve.

To remain in force mode, and to adjust the force being sent to the motor, the set_force_mN(int32_t *force*) function must be called repeatedly. If the command is not called often enough, the communication mode will return to sleep mode. The default maximum period between calls to set_force_mN(int32_t *force*) is 0.1 seconds and can be adjusted by calling set_stream_timeout(uint32_t *timeout_us*).

Calling the set_position_um(int32_t *position*) function when in force mode will have no noticeable effect.

Position Mode

The command in the position mode will write to the motor's Position Control Register to adjust the position the motor will seek to achieve.

To adjust the position being sent to the motor, call the set_position_um(int32_t *position*) function.

To remain in position mode, and to adjust the position being sent to the motor, the set_position_um(uint32_t *position*) function must be called repeatedly. If the command is not called often enough, the communication mode will return to sleep mode. The default maximum period between calls to set_position_um(int32_t *position*) is 0.1 seconds and can be adjusted by calling set_stream_timeout(uint32_t *timeout_us*).

Calling the set_force_mN(int32_t *force*) function when in position mode will have no noticeable effect.

The motor uses PID control to achieve the target position, to tune the values used by the motor call the tune_position_controller(uint16_t *p*, uint16_t *i*, uint16_t *d*, uint32_t *sat*) function.

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

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.

The following list of functions inject messages into the stream:

- void read_register(uint16_t *register_address*)
- void write_register(uint16_t *register_address*, uint16_t *reg_data*)
- void zero_position()
- void clear_errors()
- void set_max_force(int32 *max_force*)
- void set_max_temp(uint16_t *max_temp*)
- void set_max_power(uint16_t *max_power*)
- void set_safety_damping(uint16_t *d_gain*)
- void tune_position_controller(uint16_t *p*, uint16_t *i*, uint16_t *d*, uint32_t *sat*)

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.

- int32_t get_force_mN()
- int32_t get_position_um()
- uint16_t get_power_W()
- uint8_t get_temperature_C()
- uint16_t get_voltage_mV()
- uint16_t get_errors()

To determine if there is new data from the motor, call the new_data() function. If the function returns false, it can be assumed that the data has not changed since the last function call.

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().

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

Table 5: Error Type Associations

Error	Mask	Trigger Level Registers	Modules disabled	Cleared By
Configuration Errors	31 (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.

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

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.

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_lib/client/device_applications/actuator.h"


//Constructor(channel, name, cycles per microsecond)
Actuator motor(1, "Actuator 1", (COUNTS_PER_SECOND / 1000000));


Int32_t 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);
            motor.run_in();
            motor.run_out();
      }
}
/**Interrupt handling for Eagle */
//UART # corresponds to Actuator channel in constructor
void uart1_status_isr(void){
```


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

```
      motor.isr();
}


/** Interrupt handling for SuperEagle
* in interrupt.cc file a new case in the
* InterruptSystem2::MyIntcHandler function will need to be implemented
* to service the interrupt on the uarts being used by an actuator and
* call the corresponding actuator's isr function.
*/
void InterruptSystem2::MyIntcHandler(void *){
      . . .
      if (IntrStatus & 1) {
            switch (IntrNumber) {
            . . .
            //UART # corresponds to Actuator channel in constructor
                  case XPAR_AXI_INTC_0_PL_UART1_IP2INTC_IRPT_INTR:
                        motor.isr()
                        break;
            . . .
      }
      . . .
}
```


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

APPENDIX

Acceptable Baud rate and Messaging Delay Ranges

Accepted Baud Rate	Compatible Messaging Delay	Corresponding Stream Frequency
19200	1000 – 0	57 Hz - 66 Hz
192000	1000 – 0	370 Hz - 590 Hz
312500	1000 – 0	480 Hz - 900 Hz
625000	1000 – 0	630 Hz - 1700 Hz
780000	1000 – 0	680 Hz - 2000 Hz
1040000	1000 – 0	730 Hz - 2500 Hz

REVISION HISTORY

Version	Date	Author	Reason
1.0.0	December, 2021	KE, RM, KH	Initial Release
1.0.1	April, 2022	RM	Additional function revision
1.0.2	June, 2022	SW, RM	Additional and updated function description, reordering of information. Changes to heading layer and table compressing
1.0.3	June, 2022	RM	Remove references to private functions/properties formatting, error descriptions Clarify Actuator for object reference motor for device reference