ORCA API

Reference Manual
Version 1.0.1

CONTENTS

LIST OF FIGURES

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 actuators as desired and receive information from the actuator without having to interact with the serial communications protocol.

Actuator Class

The Actuator object is used to establish and maintain a connection with an actuator (aka an Orca™ motor). More…

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

Public Types

enum	CommunicationMode { SleepMode, ForceMode, PositionMode}
struct	IrisClientApplication::ConnectionConfig

Properties

-	orca_reg_contents : uint16_t[ ]	-	success_msg_counter : uint32_t
-	comms_mode : CommunicationMode	-	failed_msg_counter : uint32_t
-	stream_timeout_start : uint32_t	-	force_command : int32_t
-	stream_timeout_cycles : uint32_t	-	position_command : int32_t
-	new_data_flag : bool		

Public Functions

void	set_mode(CommunicationMode *mode*)
Communication Mode	get_mode()
void	set_force_mN(int32_t *force*)
void	set_position_um(int32_t *position*)
bool	new_data()
void	set_stream_timeout(uint32_t *timeout_us*);
void	init()
uint16_t	get_num_successful_msgs()
uint16_t	get_num_failed_msgs()
void	run_out()
void	run_in()
void	isr()
const char *	get_name()
int	channel_number()
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()
uint16_t	get_major_version()
uint16_t	get_release_state()
uint16_t	get_revision_number()
uint16_t	get_build_id()
void	zero_position()
void	clear_errors()
void	set_max_force(int32_t *force_mN*)
void	set_max_temperature(uint16_t *temp_C*)
void	set_max_power(uint16_t *pow_W*)
void	set_safety_damping(uint16_t *d_gain*)
void	tune_position_controller(uint16_t *p*, uint16_t *i*, uint16_t *d*, uint32_t *sat*)
void	read_register(uint16_t *register_address*)
void	write_register(uint16_t *register_address*, uint16_t *reg_data*)
uint16_t	get_orca_reg_content(uint16_t *offset*)

Private Functions

void	synchronize_memory_map()
void	Desynchronize_memory_map()
void	enqueue_motor_frame()
int	get_app_reception_length(uint8_t *fn_code*)
int	motor_command_fn (uint8_t *device_address*, uint8_t *register_address*, uint32_t *register_value*)

Inherited Functions

int	set_connection_config(IrisClientApplication::ConnectionConfig *config*)
bool	is_connected()
bool	is_enabled()
void	enable()
void	disable()
void	disconnect()
void	modbus_handshake()

Detailed Description

The Actuator object is used to establish and maintain a connection with an actuator (aka 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 is 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 actuator 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 actuator 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().

The enabled status of the object determines if communication with the actuator 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 actuator 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 actuator device.

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

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 actuator devices 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 device. 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.

Table 1. Manage High-speed Stream Request PDU

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

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

Device Address	1 byte	0x01
Function Code	1 byte	0x64
Command Address	1 bytes	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

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 bytes	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 actuator 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 actuator, 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() is 0.1 seconds and can be adjusted by calling set_stream_timeout().

Calling the set_position_um() 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 actuator, call the set_position_um(int32_t position) function.

To remain in position mode, and to adjust the position being sent to the actuator, 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() is 0.1 seconds and can be adjusted by calling set_stream_timeout().

Calling the set_force_mN () 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.

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()
- void write_register()
- void zero_position()
- void clear_errors()
- void set_max_force(int32_t force_mN)
- void set_max_temp(uint16_t temp_C)
- void set_max_power(uint32 p)
- 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 actuator 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 actuator device, 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.

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 3 actuator objects (can be adapted for a simple or additional actuators)

```
#include "modbus_lib/client/device_applications/actuator.h"

///(for Eagle)

#include <modbus_lib.h>

#include “client/device_applications/actuator.h"

///

#define COUNTS_PER_SECOND 666666687 /2


Actuator actuator[3]{

                {0, "Actuator 0", (COUNTS_PER_SECOND / 1000000)},

                {1, "Actuator 1", (COUNTS_PER_SECOND / 1000000)},

                {2, "Actuator 2", (COUNTS_PER_SECOND / 1000000)}

                };

Actuator::ConnectionConfig connection_config;

int main(){

     connection_config.server_address           = 1;

     connection_config.req_num_discovery_pings   = 5;

     connection_config.max_consec_failed_msgs    = 5;

     connection_config.target_baud_rate_bps      = 625000;

     connection_config.target_delay_us           = 50;


     for (int i = 0; i <3 ; i++){

          actuator[i].set_connection_config(connection_config);
```

```
                actuator[i].init();
                actuator[i].enable();
                actuator[i].set_mode(Actuator::ForceMode);
        }

        while(1){
                for (int i = 0; i <3 ; i++){
                        actuator[i].run_in();
                        actuator[i].run_out();
                        if(actuator[i].new_data()){
                                //perform calculations with new actuator data
                        }
                }
        }
}
```

(interrupt handling for eagle)

```
void uart0_status_isr(void){
         actuator[0].isr();
}
void uart1_status_isr(void){
         actuator[1].isr();
}
void uart_status_isr(void){
         actuator[2].isr();
}
```

(interrupt handling in super eagle)

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 *)

{

u32 IntrStatus;

int IntrNumber;

/* Get the interrupts that are waiting to be serviced */

IntrStatus = Xil_In32(XPAR_INTC_SINGLE_BASEADDR + XIN_ISR_OFFSET) &
Xil_In32(XPAR_INTC_SINGLE_BASEADDR + XIN_IER_OFFSET);


for (IntrNumber = 0; IntrNumber < XPAR_INTC_MAX_NUM_INTR_INPUTS;
IntrNumber++) {

      if (IntrStatus & 1) {

            switch (IntrNumber) {


            case XPAR_AXI_INTC_0_PL_UART0_IP2INTC_IRPT_INTR:
      actuator[0].isr();

                  break;

            case XPAR_AXI_INTC_0_PL_UART1_IP2INTC_IRPT_INTR:
      actuator[1].isr()

                  break;

            case XPAR_AXI_INTC_0_PL_UART2_IP2INTC_IRPT_INTR:
      actuator[2].isr()

                  break;

            }

            ack( IntrNumber );

            break;

      }

      /* Move      to the next interrupt to check */

      IntrStatus >>= 1;

/* If there are no other bits set indicating that all

 * interrupts have been serviced, then exit the loop
```

```
  */
        if (IntrStatus == 0) {
             break;
        }    // if interrupt is enabled and triggered
}     // for all interrupts
}
```

Member Type Documentation

enum Actuator::CommunicationMode [public]

This enum describes the three communication modes that determine which commands are streamed to the actuator.

Constant	Value	Description
Actuator::SleepMode	0x0	Setting comms_mode to SleepMode results in streaming commands that put the actuator into "sleep" mode.
Actuator::ForceMode	0x2	Setting comms_mode to ForceMode results in streaming commands which write to the actuator's Force Control Register.
Actuator::PositionMode	0x4	Setting comms_mode to PositionMode results in streaming commands to the actuator's Position Control Register.

struct IrisClientApplication::ConnectionConfig [public]

This struct contains parameters used to configure aspects of the handshake and connection maintenance with the actuator device.

Variable	Type	Default Value	Description
server_address	uint8_t	1	The MODBUS specified server address (1 - 247) given to the actuator device.
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 actuator device.
target_baud_rate_bps	uint32_t	625000	The baud rate the handshake will request the server to change to.
target_delay_us	uint16_t	80	The messaging delay the handshake will request the server to change to.

response_timeout_us	uint32_t	8000	The response timeout to be used once a connection has been established and a faster baud rate was negotiated.

These variables must be set and passed to set_connection_config() before enabling the actuator object in order to take effect.

Public Member Function Documentation

void Actuator::set_mode(CommunicationMode *mode*) **[public]**
Change the Actuator Object's communication mode which determines the type of commands being streamed to the actuator.

CommunicationMode Actuator::get_mode() **[public]**
Returns the current communication mode of the Actuator Object. Used to determine which type of command is being streamed to the actuator.

void Actuator::set_force_mN(int32_t *force*) **[public]**
Set/adjust the force, in milli-Newtons, used in the request to write to the actuator's Force Control Register which is sent when the communication mode is set to Force Mode.

void Actuator::set_position_um(int32_t position) **[public]**
Set/adjust the position, in micro-meters, used in the request to write to the actuator's Position Control Register which is sent when the communication mode is set to Position Mode.

bool Actuator::new_data() **[public]**
Determine if new data has been returned by one of the stream or single command responses.

Return true if new data has been written to the local copy of the actuator's register contents since the last call to new_data(), false otherwise.

void Actuator::set_stream_timeout(uint32_t *timeout_us*) **[public]**
Configure the maximum time between calls to set_force_mN() or set_position_um(), when in force or position mode respectively, before timing out and returning to sleep mode.

void Actuator::run_in() **[public]**
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 Actuator::run_out() **[public]**
This function dispatches transmissions for motor frames when connected and dispatches handshake messages when not. It must be externally paced, ie. called at the frequency that transmission should be sent.

int32_t Actuator::get_force_mN() **[public]**
Get the output force, in milli-Newtons, of the motor returned in the latest stream command response.

uint32_t Actuator::get_position_um() [public]
Get the actuator's position, in micro-meters, returned in the latest stream command response.

uint16_t Actuator::get_power_W() [public]
Get the output power, in watts, of the motor returned in the latest stream command response.

uint8_t Actuator::get_temperature_C() [public]
Get the temperature, in degrees Celsius, of the motor returned in the latest stream command response.

uint16_t Actuator::get_voltage_mV() [public]
Get the supply voltage, in millivolts, to the motor returned in the latest stream command response.

uint16_t Actuator::get_errors() [public]
Get the active error count returned in the latest stream command response.

Error Flags:
1 -parameters_invalid
2 -stator_cal_invalid
4 -shaft_cal_invalid
8 -force_cal_invalid
16 -tuning_invalid
32 -force_control_clipping
64 -max_temp_exceeded
128 -max_force_exceeded
256 -max_power_exceeded
512 -low_shaft_quality
1024 -voltage_invalid
2048 -comms_timeout

void Actuator::zero_position() [public]
Send a single command to the actuator device to use its current position as the zeroed position value.

void Actuator::clear_errors() [public]
Send a single command to the actuator device that requests all error counters to be cleared.

void Actuator::set_max_force (int32_t force_mN) [public]
Send a single command to the actuator device that configures the maximum allowable force output.

void Actuator::set_max_temperature (uint16_t temp_C) [public]
Send a single command to the actuator device that configures the maximum allowable actuator temperature.

void Actuator::read_register(uint16_t *reg_address*) [public]
Send a request to read the current contents of a specific register in the actuator device's register pool.

void Actuator::write_register(uint16_t *reg_address*, uint16_t *reg_data*) **[public]**
Send a request to write to one of the actuator device's registers.

int IrisClientApplication::set_connection_config(ConnectionConfig *config*) **[public]**
Error check and apply the handshake and connection configuration parameters as set in *config.*

bool IrisClientApplication::is_connected() **[public]**
Determine whether the client has completed a successful handshake to connect with the actuator device.

Returns true if the connection state has been reached by completing the handshake, false otherwise.

bool IrisClientApplication::is_enabled() **[public]**
Determine if communication with the actuator device is enabled or not.

Returns true if enabled, false if disabled.

void IrisClientApplication::enable() **[public]**
Enable communication with the actuator device. Allows the handshake sequence to begin and enables transceiver hardware.

void IrisClientApplication::disable() **[public]**
Disable communication with the actuator device. Moves into disconnecting state where transceiver hardware will be disabled.

APPENDIX

Acceptable Baud rate and Messaging Delay Ranges According to Device

Device Name

Accepted Baud Rate	Compatible Messaging Delay	Corresponding Stream Frequency
19200	1000 - 10	35 Hz - 60 Hz
192000	1000 - 10	220 Hz - 500 Hz
312500	1000 - 10	280 Hz - 800 Hz
625000	1000 - 10	550 Hz - 1400 Hz
780000	1000 - 10	380 Hz - 1600 Hz
1040000	1000 - 10	400 Hz - 2400 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