IQ2306 Communication Manual - Speed Firmware

1 Contents

Contents

2 UART Messaging

2.1 Introduction

IQ uses a fully featured, serial based protocol for communicating with motor controllers. This communication protocol is broken into classes of related functionality. As such, IQ supplies libraries for communicating with the motor controllers in various object-oriented languages.

2.1.1 Arduino

The Arduino libraries allow simple get/set/save access to all parameters on the modules, which includes the various control methods to make the modules spin. Please see the official Arduino library installation guide at `https://www.arduino.cc/en/guide/libraries`. Using the Library Manager is preferred. Search for "IQ Module Communication". A .zip of the library can also be found at `http://iq-control.com/libraries`.

To use the IQ Module Communication library, you must include the library header file (done automatically by Arduino if you use their Sketch → Include Library → IQ Module Communication), by typing

```
#include <iq_module_communicaiton.hpp>
```

Next make an IqSerial object. You have two options. Use the default constructor to use the default Serail (also called Serial0) on Arduino by typing

```
IqSerial ser;
```

or select the Serial peripheral of your choice by passing in the Serial object by typing

```
IqSerial ser(Serial2);
```

Next, create a client object to communicate with the motor. Here we'll talk with the Brushless Drive Client by typing

```
BrushlessDriveClient mot(0);
```

0 indicates the Module ID. See the System Control Client documentation for more details.

Now we can setup the Arduino and communication. Initialize the IqSerial object by calling begin. The default begin function will set the baud rate to 115200.

```
void setup() {
  // Initialize the IqSerial object
  ser.begin();

  // Initialize Serial (for displaying information on the terminal)
  Serial.begin(115200);
}
```

If you have changed the motor module's baud rate, pass in the non-default baud rate into the begin function.

```
void setup() {
  // Initialize the IqSerial object
  ser.begin(9600);

  // Initialize Serial (for displaying information on the terminal)
  Serial.begin(115200);
}
```

Finally, use the client to talk to the motor. Here we'll repeatedly send it a low level spin command.

```
void loop() {
  static float velocity = 0.0f;

  // Send a voltage command of 0.1 volts
  ser.set(mot.drive_spin_volts_,0.1f);

  // Get the velocity.  If a response was received...
  if(ser.get(mot.obs_velocity_,velocity))
  {
  // Display the reported velocity
  Serial.println(velocity);
  }
}
```

The get, set, an save functions take in a client_object.sub_object as the first parameter.

In get, the second parameter is a reference to a value. The get function will put the response from the motor in this value variable. The get function returns a bool, which indicate if it received a response from the motor.

The set function takes in the value to be sent as the second parameter. This value must be the datatype expected by the client_object.sub_object.

The save function requires just the client_object.sub_object.

For a complete list of client_object.sub_object and their datatypes, please see the documentation for the client classes.

2.2 C++ Libraries

The C++ libraries (C++11 standard) contain all of the required code to form and decode communication packets. They also contain tools for buffering packets until ready for transmission on your hardware and for storing received packets until parsing.

Each communication client object is capable of forming packets to send get, set, and save messages to a motor controller. This is done in the library with a sub-object for each piece of data that can be get, set, and saved. Thus, to form a get message, use

```
client_object.sub_object.get(CommunicationInterface &com)
```

To form a set message with a value of data type T, use


```
client_object.sub_object.set(CommunicationInterface &com, T value)
```

To form a set message with no value, use

```
client_object.sub_object.set(CommunicationInterface &com)
```

Finally, to form a save message, use

```
client_object.sub_object.save(CommunicationInterface &com)
```

These commands form serialized get/set/save packets and store them into a CommunicationInterface object. We supply a hardware agnostic CommunicationInterface called GenericInterface. Once packets are stored in the GenericInterface object, the user must remove the bytes with the class method

```
interface_object.GetTxBytes(uint8_t* data_out, uint8_t& length_out)
```

and send the bytes in `data_out` over the hardware serial.

Similarly, when bytes are received over the hardware serial they must be transferred into the GenericInterface using the class method

```
interface_object.SetRxBytes(uint8_t* data_in, uint16_t length_in)
```

Once transferred, a packet can be peeked using

```
int8_t interface_object.PeekPacket(uint8_t **packet, uint8_t *length)
```

which will return 1 if there is a packet, 0 if not. If there is a packet, this packet must be passed to the client objects using

```
client_object.ReadMsg(CommunicationInterface& com, uint8_t* rx_data, uint8_t rx_length)
```

Once passed to all objects, drop the packet using `interface_object.DropPacket()`.

You can check for newly received data with

```
client_object.sub_object.IsFresh()
```

To retrieve the most recent data, regardless of its freshness, use

```
data = client_object.sub_object.get_reply()
```

The below is a complete example of a program using the Arduino programming environment. This example is to demonstrate how to use the clients, the GenericInterface class, and the transfer of data between the classes and the Arduino Serial class. Please note that IQ's dedicated Arduino libraries streamline the data transfer process, thus, actual Arduino programming is simpler than the below example. Please see the Arduino documentation if you intend on using the Arduino programming environment.

```
/*
*  IQ Motion Control spin and report demo.
*
*  This code will command a motor to spin at various voltages and
*  simultaniously report the motor's position and velocity over USB
*
*
*  The circuit:
*    Serial1 RX is directly connected to motor TX (Red)
*    Serial1 TX is directly connected to motor RX (White)
*
*  Created 2018/10/8 by Matthew Piccoli
*
*  This example code is in the public domain.
*/
```

```
// USER SETABLE VALUES HERE---------------------------------------------------
// Voltage step size
const float kVoltageStep = 0.01f;
// Max voltage
const float kVoltageMax = 0.25f;
// END USER SETABLE VALUES-----------------------------------------------------

// Includes required for communication
// Message forming interface
#include <generic_interface.hpp>
// Clients that speaks to module's objects
#include <brushless_drive_client.hpp>

// Make a communication interface object
GenericInterface com;
// Make a objects that talk to the module
BrushlessDriveClient mot(0);

void setup() {
  // Initialize USB communicaiton
  Serial.begin(115200);
  Serial.print("Program starting");
  Serial.println();

  // Initialize the Serial peripheral for motor controller
  Serial1.begin(115200);
}

void loop() {
  static float voltage_to_set = 0.0f;
  static float voltage_sign = 1.0f;

  // Update voltage command
  if(abs(voltage_to_set) >= kVoltageMax)
  {
    voltage_sign = -1*voltage_sign;
  }
  voltage_to_set += kVoltageStep*voltage_sign;

  SendMessages(voltage_to_set);
  ReceiveMessages();
  DoSomethingWithMessages();

  delay(100);
}

void SendMessages(float voltage_command)
{
  // This buffer is for passing around messages.
  uint8_t communication_buffer[64];
  // Stores length of message to send or receive
  uint8_t communication_length;
```



```
  // Generate the set message
  mot.drive_spin_volts_.set(com, voltage_command);

  // Generate the get message
  mot.obs_angle_.get(com);
  mot.obs_velocity_.get(com);

  // Grab outbound messages in the com queue, store into buffer
  // If it transferred something to communication_buffer...
  if(com.GetTxBytes(communication_buffer,communication_length))
  {
    // Use Arduino serial hardware to send messages
    Serial1.write(communication_buffer,communication_length);
  }

  Serial.print("Setting voltage: ");
  Serial.print(voltage_command);
  Serial.println();
}

void ReceiveMessages()
{
  // This buffer is for passing around messages.
  uint8_t communication_buffer[64];
  // Stores length of message to send or receive
  uint8_t communication_length;

  // Reads however many bytes are currently available
  communication_length = Serial1.readBytes(communication_buffer, Serial1.available());
  // Puts the recently read bytes into coms receive queue
  com.SetRxBytes(communication_buffer,communication_length);
  uint8_t *rx_data; // temporary pointer to received type+data bytes
  uint8_t rx_length; // number of received type+data bytes
  // while we have message packets to parse
  while(com.PeekPacket(&rx_data,&rx_length))
  {
    // Share that packet with all client objects
    mot.ReadMsg(com,rx_data,rx_length);
    // Once were done with the message packet, drop it
    com.DropPacket();
  }
}

void DoSomethingWithMessages()
{
  // Check if we have any fresh data
  // Checking for fresh data is not required, it simply
  // lets you know if you received a message that you
  // have not yet read.

  // Check for a new angle message
  if(mot.obs_angle_.IsFresh()) {
    Serial.print("Angle: ");
    Serial.print(mot.obs_angle_.get_reply());
```

```
    Serial.println();
  }

  // Check for a new velocity message
  if(mot.obs_velocity_.IsFresh()) {
    Serial.print("Velocity: ");
    Serial.print(mot.obs_velocity_.get_reply());
    Serial.println();
  }
}
```

2.3 Matlab Libraries

The Matlab libraries contain everything required to open a serial port, send and receive messages on that serial port, and parse the results. First, create a MessageInterface, which opens a serial port and is responsible for the transmission and reception of messages, by typing

```
com = MessageInterface('COM_PORT',115200);
```

Replace the 'COM_PORT' string with the port string for your serial device (FTDI or similar). In Windows, this string has the form 'COM1', 'COM2', etc. In a Unix based OS, this string has the form '/dev/ttyUSB0' or similar and depends on the device. The default serial baud rate for the motor controller is 115200.

To communicate to the motor controller, create a client object using

```
client_object = ClientClass('com',com);
```

Then, send and receive messages using this object via the `get`, `set`, and `save` member functions.

```
value = client_object.get('short_name');
```

sends a get request to the motor controller and waits for its response. The responded value is returned.

```
client_object.set('short_name', value); % with value
client_object.set('short_name'); % without value
```

sends a set message. If the message requires a value, the value is stored in the motor controller's RAM.

```
client_object.save('short_name');
```

sends a save message, which store's the current RAM value into non-volatile memory. These functions are blocking and perform all necessary tasks for messaging.

All clients have added member functions `list`, `get_all`, `set_all`, `set_verify`, and `save_all`.

```
client_object.list()
```

displays all possible short names, their data types, and their units.

```
data_all = client_object.get_all()
```

performs a `get` on all messages in `list` and stores it in `data_all`.

```
client_object.set_all(data_all);
data.short_name1 = 0;
data.short_name2 = 1;
client_object.set_all(data);
```

will send set messages for all fieldnames in `data`.

```
client_object.set_verify('short_name', value);
```

performs the same function as `set`, but also performs a get to verify transmission. It will retry up to 10 times if transmission fails.

```
client_object.save_all()
```

saves all values currently in the motor controller's RAM into non-volatile memory.

For a complete example of usage, please see the documentation for the client classes.

2.4 Message Packetization

2.4.1 General Packets

Packets are structured such that a valid packet is identifiable even when surrounded by random bytes. The start byte (dec 85, hex 0x55, bin 0b01010101) indicates the start of a new packet. This particular start byte is simple to identify using an oscilloscope by its alternating 01 pattern. The 'length' byte indicates length of the 'payload' segment, which in turn points to the location of the two CRC bytes. The 'type' byte indicates the source and destination client for this packet. 'type' is required and comes immediately before 'payload'. The 'payload' segment is a variable length section, little-endian, with a practical limit of 59 bytes due to the common 64 byte buffer size for many serial protocols. Finally, a little-endian CRC-16-CCITT cyclic redundancy check on 'length'+'type'+'payload' is appended. See Table 1 a visual representation.

2.4.2 Standard Packets

A common use for communication is to get values, set values, save values, and reply to a get request. The General Packet is used to make a Standard Packet capable of performing these four functions. The 'payload' segment of the General Packet is broken into three new segments: subType, obj/access, and 'data'. The 'subType' byte refers to the specific value within the 'type' that is being get/set/saved. The 'obj' and 'access' are packed into a single byte, where 'obj', using the high 6 bits, refers to the Module ID (settable in the System Control class) and the 'access', the lowest 2 bits, indicate the message intention. 'access' can be one of four values: "get" = 0, "set" = 1, "save" = 2, "reply" = 3. "get" access requests a "reply" of 'subType'. "set" access places –data– into the 'subType'. "save" access saves the 'subType' value in flash. "reply" is the response from a "get", usually with a value loaded into –data–.

Table 1: Message Packetization

General	0x55	length	type	–payload–			crcL	crcH
Standard	0x55	length	type	subType	obj/access	–data–	crcL	crcH

2.5 Propeller Motor Control

The Propeller Motor Controller is an open and closed loop controller designed to drive propeller loads. If given thrust coefficients, this controller can be commanded in units of thrust, seamlessly accepting values from flight controllers in their native units. An added benefit is the decoupling of flight controller gains from motor choice, propeller choice, battery level, and more. Thrust commands are fed into a PID velocity controller with a second order polynomial feed forward. This sits on top of a voltage controller, which compensates for varying input voltages. Finally, the core is a raw PWM controller. Any of the above controllers can be used by the user.

2.5.1 Arduino

To use Propeller Motor Controller in Arduino, ensure iq_module_communication.hpp is included. This allows the creation of a PropellerMotorControlClient object. See Table 2 for available messages. All message objects use the Short Name with a trailing underscore. All messages use the standard Get/Set/Save functions.

A minimal working example for the PropellerMotorControlMessaging is:

```
#include <iq_module_communicaiton.hpp>

IqSerial ser(Serial2);
PropellerMotorControlClient prop(0);

void setup() {
  ser.begin();
}
```

```
void loop() {
  ser.set(prop.ctrl_velocity_,50.0f);
}
```

2.5.2 C++

To use Propeller Motor Controller in C++, include propeller_motor_control_client.hpp. This allows the creation of a PropellerMotorControlClient object. See Table 2 for available messages. All message objects use the Short Name with a trailing underscore. All messages use the standard Get/Set/Save functions.

A minimal working example for the PropellerMotorControlClient is:

```
#include "generic_interface.hpp"
#include "propeller_motor_control_client.hpp"

float velocity = 100.0f; // rad/s

void main()
{
// Make a communication interface object
GenericInterface com;

// Make a Propeller Motor Controller object with obj_id 0
PropellerMotorControlClient prop(0);

// Use the Propeller Motor Controller object
prop.ctrl_velocity_.set(com, velocity);

// Insert code for interfacing with hardware here

}
```

2.5.3 Matlab

To use Propeller Motor Controller in Matlab, all IQ communication code must be included in your path. This allows the creation of a PropellerMotorControlClient object. See Table 2 for available messages. All message strings use the Short Names. All messages use the standard Get/Set/Save functions.

A minimal working example for the PropellerMotorControlClient is:

```
% Make a communication interface object
com = MessageInterface('COM18',115200);

% Make a PropellerMotorControlClient object with obj_id 0
prop = PropellerMotorControlClient('com',com);

% Use the PropellerMotorControlClient object
prop.set('ctrl_velocity',100.0);
```


2.5.4 Message Table

Table 2: Type ID 52: Propeller Motor Controller

Sub ID	Short Name	Data Type	Unit	Note
0	ctrl_mode	int8	enum	-1 = no change, 0 = brake, 1 = coast, 2 = pwm, 3 = volts, 4 = velocity, 5 = thrust
1	ctrl_brake			Shorts motor leads, slows motor down dissipating energy in motor
2	ctrl_coast			Disables all drive circuitry
3	ctrl_pwm	float	PWM	[-1, 1] fraction of input voltage
4	ctrl_volts	float	V	[-supply, supply] Voltage to apply to motor
5	ctrl_velocity	float	rad/s	Angular velocity command
6	ctrl_thrust	float	N	Thrust command (requires kt values)
7	velocity_kp	float	V/(rad/s)	Proportional gain
8	velocity_ki	float	V/(rad)	Integral gain
9	velocity_kd	float	V/(rad/s^2)	Derivative gain
10	velocity_ff0	float	V	Feed forward 0th order term
11	velocity_ff1	float	V/(rad/s)	Feed forward 1st order term
12	velocity_ff2	float	V/(rad/s)^2	Feed forward 2nd order term
13	propeller_kt_pos	float	N/(rad/s)^2	$T = k_t\omega^2$ thrust constant in positive direction
14	propeller_kt_neg	float	N/(rad/s)^2	$T = k_t\omega^2$ thrust constant in negative direction
15	timeout	float	s	The controller must receive a message within this time otherwise it is set to coast mode
16	input_filter_fc	uint32	Hz	Low pass cutoff frequency for input commands

2.6 Brushless Drive

Brushless Drive is the low level driver of the motor's phase voltage.

2.6.1 Arduino

To use Brushless Drive in Arduino, ensure iq_module_communication.hpp is included. This allows the creation of a BrushlessDriveClient object. See Table 3 for available messages. All message objects use the Short Name with a trailing underscore. All messages use the standard Get/Set/Save functions.

A minimal working example for the BrushlessDriveClient is:

```
#include <iq_module_communicaiton.hpp>

IqSerial ser(Serial2);
BrushlessDriveClient mot(0);

void setup() {
  ser.begin();
}

void loop() {
  ser.set(mot.drive_spin_volts_,0.1f);
}
```

2.6.2 C++

To use Brushless Drive in C++, include brushless_drive_client.hpp. This allows the creation of a BrushlessDriveClient object. See Table 3 for available messages. All message objects use the Short Name with a trailing underscore. All messages use the standard Get/Set/Save functions.

A minimal working example for the BrushlessDriveClient is:

```
#include "generic_interface.hpp"
#include "brushless_drive_client.hpp"

float velocity;

void main()
{
// Make a communication interface object
GenericInterface com;

// Make a Brushless Drive object with obj_id 0
BrushlessDriveClient drive;

// Use the Brushless Drive object
drive.obs_velocity_.get(com);
drive.drive_volts_.set(com,1.0f);

// Insert code for interfacing with hardware here

// Read response
velocity = drive.obs_velocity_.get_reply();
}
```

2.6.3 Matlab

To use Brushless Drive in Matlab, all IQ communication code must be included in your path. This allows the creation of a BrushlessDriveClient object. See Table 3 for available messages. All message strings use the Short Names. All messages use the standard Get/Set/Save functions.

A minimal working example for the BrushlessDriveClient is:

```
% Make a communication interface object
com = MessageInterface('COM18',115200);

% Make a BrushlessDriveClient object with obj_id 0
drive = BrushlessDriveClient('com',com);

% Use the BrushlessDriveClient object
velocity = drive.get('obs_velocity');
drive.set('drive_volts',1);
```


2.6.4 Message Table

Table 3: Type ID 50: Brushless Drive

Sub ID	Short Name	Data Type	Unit	Note
0	drive_mode	uint8		0 = phase_pwm, 1 = phase_volts, 2 = spin_pwm, 3 = spin_volts, 4 = brake, 5 = coast
1	drive_phase_pwm	float	pwm	
2	drive_phase_volts	float	V	
3	drive_spin_pwm	float	pwm	Spins motor with this throttle [-1, 1]
4	drive_spin_volts	float	V	Spins motor with this voltage
5	drive_brake			
6	drive_coast			
7	drive_angle_offset	float	rad	
8	drive_pwm	float	pwm	The applied pwm after all computation and limiting [-1, 1]
9	drive_volts	float	V	The applied pwm after all computation and limiting
10	mech_lead_angle	float	rad	
11	obs_supply_volts	float	V	Observed supply voltage
12	obs_angle	float	rad	Observed motor angle
13	obs_velocity	float	rad/s	Observed motor velocity
14	motor_pole_pairs	uint16		Number of motor pole pairs (magnets/2)
15	motor_emf_shape	uint8		
16	permute_wires	uint8	bool	
17	calibration_angle	float	rad	
18	lead_time	float	s	
19	commutation_hz	uint32	Hz	
20	phase_angle	float	rad	
32	motor_Kv	float	RPM/V	Motor's voltage constant
33	motor_R_ohm	float	ohm	Motor's resistance
34	motor_I_max	float	A	Max allowable motor current
35	volts_limit	float	V	Max regen voltage
36	est_motor_amp	float	A	Estimated motor amps
37	est_motor_torque	float	Nm	Estimated motor torque
38	motor_redline_start	float	rad/s	Speed at which motor begins to derate
39	motor_redline_end	float	rad/s	Speed at which the motor is fully derated
40	motor_l	float	H	Cross inductance
41	derate	int32	PU fix16	Amount of derating. No derate = 65536, full derate = 0
42	i_soft_start	float	A	Current at which motor begins to derate
43	i_soft_end	float	A	Current at which the motor is fully derated

2.7 Anticogging

Anticogging is the process of electronically canceling out cogging torque of a motor. Each motor is loaded with its unique cog information. This class allows enabling and disabling the anticogging process. Though this class can also manipulate the cog information it is not recommended to manipulate or erase this data as it is unrecoverable.

2.7.1 Arduino

To use the Anticogging in Arduino, ensure iq_module_communication.hpp is included. This allows the creation of a AnticoggingClient object. See Table 4 for available messages. All message objects use the Short Name with a trailing underscore. All messages use the standard Get/Set/Save functions.

A minimal working example for the AnticoggingClient is:

```
#include <iq_module_communicaiton.hpp>

IqSerial ser(Serial2);
BrushlessDriveClient mot(0);
AnticoggingClient cog(0);

void setup() {
  ser.begin();
  ser.set(mot.drive_spin_volts_,0.0f);
}

void loop() {
  // Spin the motor with your hand and feel Anticogging turning on and off
  ser.set(cog.is_enabled_,(uint8_t)1);
  delay(2000);
  ser.set(cog.is_enabled_,(uint8_t)0);
  delay(2000);
}
```

2.7.2 C++

To use the Anticogging client in C++, include anticogging_client.hpp. This allows the creation of a AnticoggingClient object. See Table 4 for available messages. All message objects use the Short Name with a trailing underscore. All messages use the standard Get/Set/Save functions.

A minimal working example for the AnticoggingClient is:

```
#include "generic_interface.hpp"
#include "anticogging_client.hpp"

void main()
{
// Make a communication interface object
GenericInterface com;

// Make a ESC Propeller Input Parser object with obj_id 0
AnticoggingClient cog(0);

// Use the ESC Propeller Input Parser object
cog.is_enabled_.set(com, 1);

// Insert code for interfacing with hardware here

}
```

2.7.3 Matlab

To use the Anticogging client in Matlab, all IQ communication code must be included in your path. This allows the creation of a AnticoggingClient object. See Table 4 for available messages. All message strings use the Short Names. All messages use the standard Get/Set/Save functions.

A minimal working example for the AnticoggingClient is:

```
% Make a communication interface object
com = MessageInterface('COM18',115200);

% Make an AnticoggingClient object with obj_id 0
cog = AnticoggingClient('com',com);

% Use the AnticoggingClient object
cog.set('is_enabled',1);
```

2.7.4 Message Table

Table 4: Type ID 71: Anticogging

Sub ID	Short Name	Data Type	Unit	Note
0	table_size	uint16	-	
1	is_data_valid	uint8	bool	Indicates if the cog information is valid. is_enabled must be called first to check the cog information.
2	is_enabled	uint8	bool	Indicates if anticogging is running. This will stay 0/false if the is_data_valid field is 0/false.
3	erase	-	-	Erases the cog information. This is not recommended.
4	left_shift	uint8	$*2^x$	Anticog multiplier. Modification is not recommended.

2.8 Buzzer Control

The Buzzer Control handles all beeps and songs played by the motor. The controls mimic standard MIDI commands allowing simple translation from MIDI to Buzzer Control commands. The volume_max parameter controls the absolute volume across all notes, measured in volts. To play a note on the buzzer, set the frequency by sending a 'hz' command, set a relative volume by sending a 'volume' command, set a note length by sending a 'duration' command, and finally put the controller in note mode by sending 'ctrl_note'.

2.8.1 Arduino

To use the Buzzer Control in Arduino, ensure iq_module_communication.hpp is included. This allows the creation of a BuzzerControlClient object. See Table 5 for available messages. All message objects use the Short Name with a trailing underscore. All messages use the standard Get/Set/Save functions.

A minimal working example for the BuzzerControlClient is:

```
#include <iq_module_communicaiton.hpp>

IqSerial ser(Serial2);
BuzzerControlClient buz(0);

void setup() {
  ser.begin();
}

void loop() {
  ser.set(buz.hz_,(uint16_t)1000);
  ser.set(buz.volume_,(uint8_t)127);
  ser.set(buz.duration_,(uint16_t)500);
  ser.set(buz.ctrl_note_);
  delay(1000);
}
```

2.8.2 C++

To use the Buzzer Control in C++, include buzzer_control_client.hpp. This allows the creation of a BuzzerControlClient object. See Table 5 for available messages. All message objects use the Short Name with a trailing underscore. All messages use the standard Get/Set/Save functions.

A minimal working example for the BuzzerControlClient is:

```
#include "generic_interface.hpp"
#include "buzzer_control_client.hpp"

void main()
{
// Make a communication interface object
GenericInterface com;

// Make a Buzzer Control object with obj_id 0
BuzzerControlClient buz(0);

// Use the Buzzer Control object
buz.hz_.set(com, 440); // A4
buz.volume_.set(com, 127); // Max volume
buz.duration_.set(com, 500); // 500ms
buz.ctrl_note_.set(com); // Note mode

// Insert code for interfacing with hardware here

}
```

2.8.3 Matlab

To use the Buzzer Contol in Matlab, all IQ communication code must be included in your path. This allows the creation of a BuzzerControlClient object. See Table 5 for available messages. All message strings use the Short Names. All messages use the standard Get/Set/Save functions.

A minimal working example for the BuzzerControlClient is:

```
% Make a communication interface object
com = MessageInterface('COM18',115200);

% Make a BuzzerControlClient object with obj_id 0
buz = BuzzerControlClient('com',com);

% Use the BuzzerControlClient object
buz.set('hz',440); % A4
buz.set('volume',127); % Max volume
buz.set('duration',500); % 500ms
buz.set('ctrl_note');
```


2.8.4 Message Table

Table 5: Type ID 61: Buzzer Control

Sub ID	Short Name	Data Type	Unit	Note
0	ctrl_mode	uint8	enum	no_change = -1, brake=0, coast=1, note=2, song=3
1	ctrl_brake	-	-	
2	ctrl_coast	-	-	
3	ctrl_note	-	-	Must have sent a 'hz' and 'volume' first
4	volume_max	float	V	Uses this voltage command for maximum volume
5	hz	uint16	hz	Frequency of the note
6	volume	uint8	0-127	Individual note volume as fraction of 127
7	duration	uint16	ms	Note length. Assumed max (65535 ms) if not sent.

2.9 ESC Propeller Input Parser

The ESC Propeller Input Parser is an interface between the Propeller Motor Controller and the PWM based inputs like 1-2ms, OneShot, MultiShot, and DShot. This parser allows the user to control how ratiomatic values from the PWM input are translated. Inputs can be mapped to PWM control, voltage control, velocity control, and thrust control. Values can be interpreted as signed/unsigned and clockwise/counter clockwise.

2.9.1 Arduino

To use ESC Propeller Input Parser in Arduino, ensure iq_module_communication.hpp is included. This allows the creation of a EscPropellerInputParserClient object. See Table 6 for available messages. All message objects use the Short Name with a trailing underscore. All messages use the standard Get/Set/Save functions.

A minimal working example for the EscPropellerInputParserClient is:

```
#include <iq_module_communicaiton.hpp>

IqSerial ser(Serial2);
EscPropellerInputParserClient esc(0);

void setup() {
  ser.begin();

  // Initialize Serial (for displaying information on the terminal)
  Serial.begin(115200);
}

void loop() {
  float vel_max = 0;
  ser.get(esc.velocity_max_,vel_max);
  Serial.println(vel_max);
}
```

2.9.2 C++

To use ESC Propeller Input Parser in C++, include esc_propeller_input_parser_client.hpp. This allows the creation of a EscPropellerInputParserClient object. See Table 6 for available messages. All message objects use the Short Name with a trailing underscore. All messages use the standard Get/Set/Save functions.

A minimal working example for the EscPropellerInputParserClient is:

```
#include "generic_interface.hpp"
#include "esc_propeller_input_parser_client.hpp"

float velocity_max = 1000.0f; // rad/s

void main()
{
// Make a communication interface object
GenericInterface com;

// Make a ESC Propeller Input Parser object with obj_id 0
EscPropellerInputParserClient esc(0);

// Use the ESC Propeller Input Parser object
esc.velocity_max_.set(com, velocity_max);

// Insert code for interfacing with hardware here

}
```

2.9.3 Matlab

To use ESC Propeller Input Parser in Matlab, all IQ communication code must be included in your path. This allows the creation of a EscPropellerInputParserClient object. See Table 6 for available messages. All message strings use the Short Names. All messages use the standard Get/Set/Save functions.

A minimal working example for the EscPropellerInputParserClient is:

```
% Make a communication interface object
com = MessageInterface('COM18',115200);

% Make a EscPropellerInputParserClient object with obj_id 0
esc = EscPropellerInputParserClient('com',com);

% Use the EscPropellerInputParserClient object
esc.set('thrust_max',5.0);
```

2.9.4 Message Table

Table 6: Type ID 60: ESC Propeller Input Parser

Sub ID	Short Name	Data Type	Unit	Note
0	mode	uint8	enum	0 = PWM, 1 = Voltage, 2 = Velocity, 3 = Thrust
1	raw_value	float	PU	Input value [0, 1]
3	sign	uint8	enum	0 = unconfigured, 1 = signed positive, 2 = signed negative, 3 = unsigned positive, 4 = unsigned negative
4	volts_max	float	V	Maximum voltage to apply to motor, raw_value scaled to [-volts_max, volts_max] or [0, volts_max]
5	velocity_max	float	rad/s	Maximum angular velocity command, raw_value scaled to [-velocity_max, velocity_max] or [0, velocity_max]
6	thrust_max	float	N	Maximum thrust command (requires kt values), raw_value scaled to [-thrust_max, thrust_max] or [0, thrust_max]

2.10 Hobby Input

The Hobby Input gives the module the ability to read in a variety of hobby communication protocols. Supported protocols are standard 1-2ms PWM, OneShot125, OneShot42, MultiShot, and DShot (150 - 1200). The protocols are autodetected by default, but can be set to accept a single specific protocol. The values read by the Hobby Input are fed into a Parser object, such as the Servo Parser or the ESC Parser.

2.10.1 Arduino

To use Hobby Input in Arduino, ensure iq_module_communication.hpp is included. This allows the creation of a HobbyInputClient object. See Table 7 for available messages. All message objects use the Short Name with a trailing underscore. All messages use the standard Get/Set/Save functions.

A minimal working example for the HobbyInputClient is:

```
#include <iq_module_communicaiton.hpp>

IqSerial ser(Serial2);
HobbyInputClient hin(0);

void setup() {
  ser.begin();

  ser.set(hin.allowed_protocols_,(uint8_t)6);
  ser.save(hin.allowed_protocols_);
}

void loop() {
}
```

2.10.2 C++

To use Hobby Input in C++, include hobby_input_client.hpp. This allows the creation of a HobbyInputClient object. See Table 7 for available messages. All message objects use the Short Name with a trailing underscore. All messages use the standard Get/Set/Save functions.

A minimal working example for the HobbyInputClient is:

```
#include "generic_interface.hpp"
#include "hobby_input_client.hpp"

void main()
{
// Make a communication interface object
GenericInterface com;

// Make a Hobby Input object with obj_id 0
HobbyInputClient hin(0);

// Use the Hobby Input object
hin.allowed_protocols_.set(com, 3); // Set the protocol to OneShot42
hin.allowed_protocols_.save(com);

// Insert code for interfacing with hardware here

}
```

2.10.3 Matlab

To use Hobby Input in Matlab, all IQ communication code must be included in your path. This allows the creation of a Hobby Input object. See Table 7 for available messages. All message strings use the Short Names. All messages use the standard Get/Set/Save functions.

A minimal working example for the HobbyInputClient is:

```
% Make a communication interface object
com = MessageInterface('COM18',115200);

% Make a HobbyInputClient object with obj_id 0
hin = HobbyInputClient('com',com);

% Use the EscPropellerInputParserClient object
hin.set('allowed_protocols',3);
hin.save('allowed_protocols');
```

2.10.4 Message Table

Table 7: Type ID 76: Hobby Input

Sub ID	Short Name	Data Type	Unit	Note
0	allowed_protocols	uint8	enum	Standard PWM = 1, OneShot125 = 2, OneShot42 = 3, MultiShot = 4, DShot150 = 5, DShot300 = 6, DShot600 = 7, DShot1200 = 8
1	protocol	uint8	enum	

2.11 Serial Interface

The Serial client allows the user to change settings related to the serial communication interface, namely the baud rate. The set function of the baud rate behaves as both a set then a save. This allows the user to set and save using the initial baud rate, rather than having to disconnect and reconnect using the new baud rate in order to send a save. For this reason, the standard save function for the baud rate is disabled.

2.11.1 Arduino

To use Serial Interface in Arduino, ensure iq_module_communication.hpp is included. This allows the creation of a SerialInterfaceClient object. See Table 8 for available messages. All message objects use the Short Name with a trailing underscore. All messages use the standard Get/Set/Save functions.

A minimal working example for the SerialInterfaceClient is:

```
#include <iq_module_communicaiton.hpp>

IqSerial ser(Serial2);
SerialInterfaceClient sic(0);

void setup() {
  ser.begin();

  ser.set(sic.baud_rate_,(uint32_t)9600);
}

void loop() {
}
```

To start from 9600 baud and reset the baud back to the default 115200, use:


```
#include <iq_module_communicaiton.hpp>

IqSerial ser(Serial2);
SerialInterfaceClient sic(0);

void setup() {
  ser.begin(9600);

  ser.set(sic.baud_rate_,(uint32_t)115200);
}

void loop() {
}
```

2.11.2 C++

To use Serial Interface in C++, include serial_interface_client.hpp. This allows the creation of a SerialInterfaceClient object. See Table 8 for available messages. All message objects use the Short Name with a trailing underscore.

A minimal working example for the SerialInterfaceClient is:

```
#include "generic_interface.hpp"
#include "serial_interface_client.hpp"

uint32_t baud_rate;

void main()
{
  // Make a communication interface object
  GenericInterface com;

  // Make a Serial Interface object with obj_id 0
  SerialInterfaceClient serial_interface(0);

  // Use the Serial Interface object
  serial_interface.baud_rate_.get(com);

  // Insert code for interfacing with hardware here

  // baud_rate = serial_interface.baud_rate_.get_reply();
}
```

2.11.3 Matlab

To use Serial Interface in Matlab, all IQ communication code must be included in your path. This allows the creation of a SerialInterfaceClient object. See Table 8 for available messages. All message strings use the Short Names. All messages use the standard Get/Set/Save functions.

A minimal working example for the SerialInterfaceClient is:

```
% Make a communication interface object
com = MessageInterface('COM18',115200);

% Make a Serial Interface object with obj_id 0
serial_interface = SerialInterfaceClient('com',com);
```

```
% Use the Serial Interface object
old_baud = serial_interface.get('baud_rate'); // should be 115200
serial_interface.set('baud_rate', 9600);

% Note: the baud rate is now 9600.  This com object uses 115200
% Make a new com object at 9600 baud to continue communication
com = MessageInterface('COM18',9600);
serial_interface = SerialInterfaceClient('com',com);
new_baud = serial_interface.get('baud_rate'); // should be 9600
```

2.11.4 Message Table

Table 8: Type ID 16: Serial Interface

Sub ID	Short Name	Data Type	Unit	Note
0	baud_rate	uint32	hz	Reliable up to 115200. Unreliable but functional up to 2mbps. Set also performs a save.

2.12 Power Monitor

The Power Monitor measures the power coming into the module. It reports input voltage and current as well as calculates power and energy consumed. A built in low pass filter with adjustable cutoff frequency smooths these values.

2.12.1 Arduino

To use Power Monitor in Arduino, ensure iq_module_communication.hpp is included. This allows the creation of a PowerMonitorClient object. See Table 9 for available messages. All message objects use the Short Name with a trailing underscore. All messages use the standard Get/Set/Save functions.

A minimal working example for the PowerMonitorClient is:

```
#include <iq_module_communicaiton.hpp>

IqSerial ser(Serial2);
PowerMonitorClient pwr(0);

void setup() {
  ser.begin();

  // Initialize Serial (for displaying information on the terminal)
  Serial.begin(115200);
}

void loop() {
  float voltage = 0;
  if(ser.get(pwr.volts_,voltage))
    Serial.println(voltage);
}
```

2.12.2 C++

To use Power Monitor in C++, include power_monitor_client.hpp. This allows the creation of a PowerMonitorClient object. See Table 9 for available messages. All message objects use the Short Name with a trailing underscore. All messages use the standard Get/Set/Save functions.

A minimal working example for the PowerMonitorClient is:


```
#include "generic_interface.hpp"
#include "power_monitor_client.hpp"

float volts;

void main()
{
// Make a communication interface object
GenericInterface com;

// Make a Power Monitor object with obj_id 0
PowerMontitorClient pwr(0);

// Use the Power Monitor object
pwr.volts_.get(com);

// Insert code for interfacing with hardware here

// Read response
volts = pwr.volts_.get_reply();
}
```

2.12.3 Matlab

To use Power Monitor in Matlab, all IQ communication code must be included in your path. This allows the creation of a PowerMonitorClient object. See Table 9 for available messages. All message strings use the Short Names. All messages use the standard Get/Set/Save functions.

A minimal working example for the PowerMonitorClient is:

```
% Make a communication interface object
com = MessageInterface('COM18',115200);

% Make a PowerMontitorClient object with obj_id 0
pwr = PowerMontitorClient('com',com);

% Use the PowerMontitorClient object
volts = pwr.get('volts');
```

2.12.4 Message Table

Table 9: Type ID 69: Power Monitor

Sub ID	Short Name	Data Type	Unit	Note
0	volts	float	V	Input voltage to module
1	amps	float	A	Input amperage to module
2	watts	float	W	Input wattage to module
3	joules	float	J	Total energy consumed by module
4	reset_joules	-	-	Sets joules to zero
5	filter_fs	uint32	Hz	Low pass filter sample frequency
6	filter_fc	uint32	Hz	Low pass filter cutoff frequency
7	volts_raw	uint16	bit	
8	amps_raw	uint16	bit	
9	volts_gain	float	V/bit	
10	amps_gain	float	A/bit	
11	amps_bias	float	bit	

2.13 Temperature Monitor Microcontroller

The Temperature Monitor Microcontroller reads, filters, and reports the microcontroller's internal temperature. The temperature is used to derate the motor if the temperature rises into dangerous levels. The filter's cutoff frequency and the temperature limits can be adjusted, though this is not recommended.

2.13.1 Arduino

To use the Temperature Monitor Microcontroller in Arduino, ensure iq_module_communication.hpp is included. This allows the creation of a TemperatureMonitorUcClient object. See Table 10 for available messages. All message objects use the Short Name with a trailing underscore. All messages use the standard Get/Set/Save functions.

A minimal working example for the TemperatureMonitorUcClient is:

```
#include <iq_module_communicaiton.hpp>

IqSerial ser(Serial2);
TemperatureMonitorUcClient tmp(0);

void setup() {
  ser.begin();

  // Initialize Serial (for displaying information on the terminal)
  Serial.begin(115200);
}

void loop() {
  float temperature = 0.0f;

  if(ser.get(tmp.uc_temp_,temperature))
    Serial.println(temperature);
}
```

2.13.2 C++

To use the Temperature Monitor Microcontroller client in C++, include temperature_monitor_uc_client.hpp. This allows the creation of a TemperatureMonitorUcClient object. See Table 10 for available messages. All

message objects use the Short Name with a trailing underscore. All messages use the standard Get/Set/Save functions.

A minimal working example for the TemperatureMonitorUcClient is:

```
#include "generic_interface.hpp"
#include "temperature_monitor_uc_client.hpp"

float uc_temp;

void main()
{
// Make a communication interface object
GenericInterface com;

// Make a Temperature Monitor Microcontroller object with obj_id 0
TemperatureMonitorUcClient tuc(0);

// Use the Temperature Monitor Microcontroller object
tuc.uc_temp_.get(com);

// Insert code for interfacing with hardware here

// Read response
uc_temp = tuc.uc_temp_.get_reply();
}
```

2.13.3 Matlab

To use the Temperature Monitor Microcontroller client in Matlab, all IQ communication code must be included in your path. This allows the creation of a TemperatureMonitorUcClient object. See Table 10 for available messages. All message strings use the Short Names. All messages use the standard Get/Set/Save functions.

A minimal working example for the TemperatureMonitorUcClient is:

```
% Make a communication interface object
com = MessageInterface('COM18',115200);

% Make an TemperatureMonitorUcClient object with obj_id 0
tuc = TemperatureMonitorUcClient('com',com);

% Use the TemperatureMonitorUcClient object
uc_temp = tuc.get('uc_temp');
```

2.13.4 Message Table

Table 10: Type ID 73: Temperature Monitor Microcontroller

Sub ID	Short Name	Data Type	Unit	Note
0	uc_temp	float	degC	Temperature of the microcontroller
1	filter_fs	uint32	Hz	Low pass filter sample frequency
2	filter_fc	uint32	Hz	Low pass filter cutoff frequency
3	otw	float	degC	Over temperature warning. Derating of the motor begins at this temperature.
4	otlo	float	degC	Over temperature lock out. Derating of the motor end at this temperature, where the motor is fully disabled.
5	derate	float	PU	Amount of derating applied to motor [0 1]

2.14 Temperature Estimator

The Temperature Estimator uses a conduction thermal model to estimate the temperature of components not directly sensed. In the motor modules the Temperature Estimator estimates the motor coil temperature. The temperature is used to derate the motor if the temperature rises into dangerous levels. The temperature limits can be adjusted, though this is not recommended.

2.14.1 Arduino

To use the Temperature Estimator in Arduino, ensure iq_module_communication.hpp is included. This allows the creation of a TemperatureEstimatorClient object. See Table 11 for available messages. All message objects use the Short Name with a trailing underscore. All messages use the standard Get/Set/Save functions.

A minimal working example for the TemperatureEstimatorClient is:

```
#include <iq_module_communicaiton.hpp>

IqSerial ser(Serial2);
TemperatureMonitorUcClient tmp(0);

void setup() {
ser.begin();

// Initialize Serial (for displaying information on the terminal)
Serial.begin(115200);
}

void loop() {
float temperature = 0.0f;

if(ser.get(tmp.uc_temp_,temperature))
Serial.println(temperature);
}
```

2.14.2 C++

To use the Temperature Estimator client in C++, include temperature_estimator_client.hpp. This allows the creation of a TemperatureEstimatorClient object. See Table 11 for available messages. All message objects use the Short Name with a trailing underscore. All messages use the standard Get/Set/Save functions.

A minimal working example for the TemperatureEstimatorClient is:


```
#include <iq_module_communicaiton.hpp>

IqSerial ser(2);
TemperatureEstimatorClient tes(0);

void setup() {
  ser.begin();

  // Initialize Serial (for displaying information on the terminal)
  Serial.begin(115200);
}

void loop() {
  float temperature = 0.0f;

  if(ser.get(tes.temp_,temperature))
  Serial.println(temperature);
}
```

2.14.3 Matlab

To use the Temperature Estimator client in Matlab, all IQ communication code must be included in your path. This allows the creation of a TemperatureEstimatorClient object. See Table 11 for available messages. All message strings use the Short Names. All messages use the standard Get/Set/Save functions.

A minimal working example for the TemperatureEstimatorClient is:

```
% Make a communication interface object
com = MessageInterface('COM18',115200);

% Make an TemperatureEstimatorClient object with obj_id 0
tes = TemperatureEstimatorClient('com',com);

% Use the TemperatureEstimatorClient object
coil_temp = tes.get('temp');
```

2.14.4 Message Table

Table 11: Type ID 77: Temperature Estimator

Sub ID	Short Name	Data Type	Unit	Note
0	temp	float	degC	Temperature of the motor coils
1	otw	float	degC	Over temperature warning. Derating of the motor begins at this temperature.
2	otlo	float	degC	Over temperature lock out. Derating of the motor end at this temperature, where the motor is fully disabled.
3	thermal_resistance	float	K/W	Model thermal resistance
4	thermal_capacitance	float	J/K	Model thermal capacitance
5	derate	Fix16	PU	Amount of derating applied to motor [0 65536] where 65536 is normal operation

2.15 System Control

System Control allows the user to perform low level tasks on the motor controller's microcontroller and gather basic information. The motor's uptime can be read and set, allowing for flexible timing and synchronizing. System Control also has a Module ID parameter, which allows motors to be bussed on a single serial line yet addressed uniquely. System Control is unique since its ID is always 0 even when the Module ID has been changed.

2.15.1 Arduino

To use System Control in Arduino, ensure iq_module_communication.hpp is included. This allows the creation of a SystemControlClient object. See Table 12 for available messages. All message objects use the Short Name with a trailing underscore. All messages use the standard Get/Set/Save functions.

A minimal working example for the SystemControlClient is:

```
#include <iq_module_communicaiton.hpp>

IqSerial ser(Serial2);
SystemControlClient sys(0);

void setup() {
  ser.begin();

  // Initialize Serial (for displaying information on the terminal)
  Serial.begin(115200);
}

void loop() {
  float sys_time = 0.0f;

  if(ser.get(sys.time_,sys_time))
  Serial.println(sys_time);
}
```

2.15.2 C++

To use System Control in C++, include system_control_client.hpp. This allows the creation of a System-ControlClient object. See Table 12 for available messages. All message objects use the Short Name with a trailing underscore. All messages use the standard Get/Set/Save functions.

A minimal working example for the SystemControlClient is:

```
#include "generic_interface.hpp"
#include "system_control_client.hpp"

float time;

void main()
{
  // Make a communication interface object
  GenericInterface com;

  // Make a System Control object with obj_id 0
  // System Control objects are always obj_id 0
  SystemControlClient system_control(0);

  // Use the System Control object
```



```
  system_control.time_.get(com);

  // Insert code for interfacing with hardware here

  // time = system_control.time_.get_reply();
}
```

2.15.3 Matlab

To use System Control in Matlab, all IQ communication code must be included in your path. This allows the creation of a SystemControlClient object. See Table 12 for available messages. All message strings use the Short Names. All messages use the standard Get/Set/Save functions.

A minimal working example for the SystemControlClient is:

```
% Make a communication interface object
com = MessageInterface('COM18',115200);

% Make a System Control object with obj_id 0
% System Control objects are always obj_id 0
system_control = SystemControlClient('com',com);

% Use the System Control object
time = system_control.get('time');
```

2.15.4 Message Table

Table 12: Type ID 5: System Control

Sub ID	Short Name	Data Type	Unit	Note
0	reboot_program			Reboots the motor controller with saved values
1	reboot_boot_loader			Reboots into the boot loader
2	dev_id	uint16		
3	rev_id	uint16		
4	uid1	uint32		
5	uid2	uint32		
6	uid3	uint32		
7	mem_size	uint16	Kb	
8	build_year	uint16	year	
9	build_month	uint8	mon	
10	build_day	uint8	day	
11	build_hour	uint8	hour	
12	build_minute	uint8	min	
13	build_second	uint8	s	
14	module_id	uint8	id	The ID used for all obj_id on this module
15	time	float	s	Internal clock time. If unchanged through software this is uptime
16	firmware_version	uint32	ver	
17	hardware_version	uint32	ver	
18	electronics_version	uint32	ver	
19	firmware_valid	uint8	bool	