IQ2306 Speed Module

1 Features

- Up to 20% efficiency boost in hover applications
- 4.8% more shaft power than same sized motors
- "Thrust" controller that allows flight controller to be motor and propeller agnostic
- Velocity controller with PID and 2nd order feed forward
- Voltage controller
- PWM controller
- Coast and brake modes
- No minimum speed
- Immediate reversibility (3D mode)
- Backdrivable
- Regenerative braking
- Active freewheeling
- Current limiter
- Serial (UART) w/ access to control parameters
- 1-2ms PWM
- Oneshot (42, 125)
- MultiShot
- DShot (150-1200) (autodetect)

2 Applications

- Drones
- Fans
- Wheeled vehicles
- Displays

3 Description

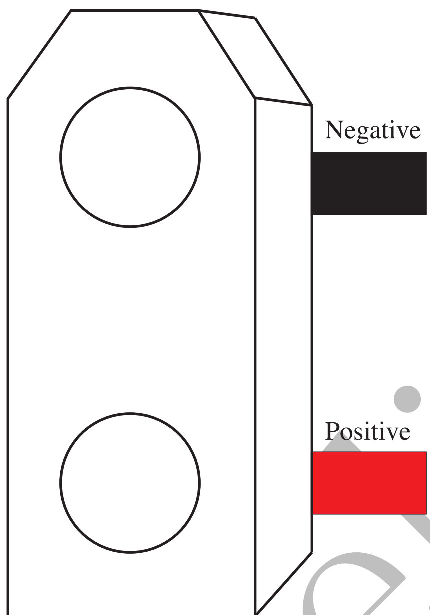
The IQ2306 Speed Module in an integrated motor and controller with a wide range of velocity based applications. It has an open and closed loop controller designed primarily to drive propeller loads. Its performance is comparable to or better than other 2306 sized motors and can operate at any speed between -30,000 and 30,000 RPM thanks to its sensored control.

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.

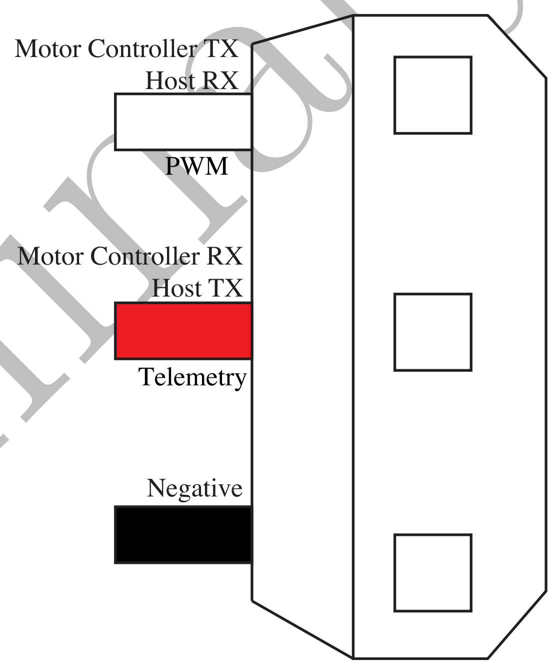
4 Electrical Specifications

Description	Symbol	Min	Max	Unit	Notes
Supply Voltage	V_{CC}	6	17	V	Designed for 3S-4S LiPo, use with caution on empty 2S
Continuous Stall Current	I_S	-	20	A	Motor current
Continuous Rotating Current	I_{SR}	-	30	A	Motor current
Pulsed Current	I_{SP}	-	60	A	Motor current
Digital Logic Voltage	V_L	-0.3	7.3	V	3.3 V system, 5 V tolerant

5 Electrical Interface



(a) XT-30 Power Connector



(b) JR Servo Communication Connector

5.1 Supply Wiring

Power is transmitted to the IQ2306 Speed Module via an XT-30 connector and 15cm of 18 AWG wires. The motor controller has the male connector, while the supply has the female connector. Both genders of the connector have positive and negative markings. The wires are soldered directly to the motor controller PCB for lower resistance. Please refrain from unnecessary tugging.

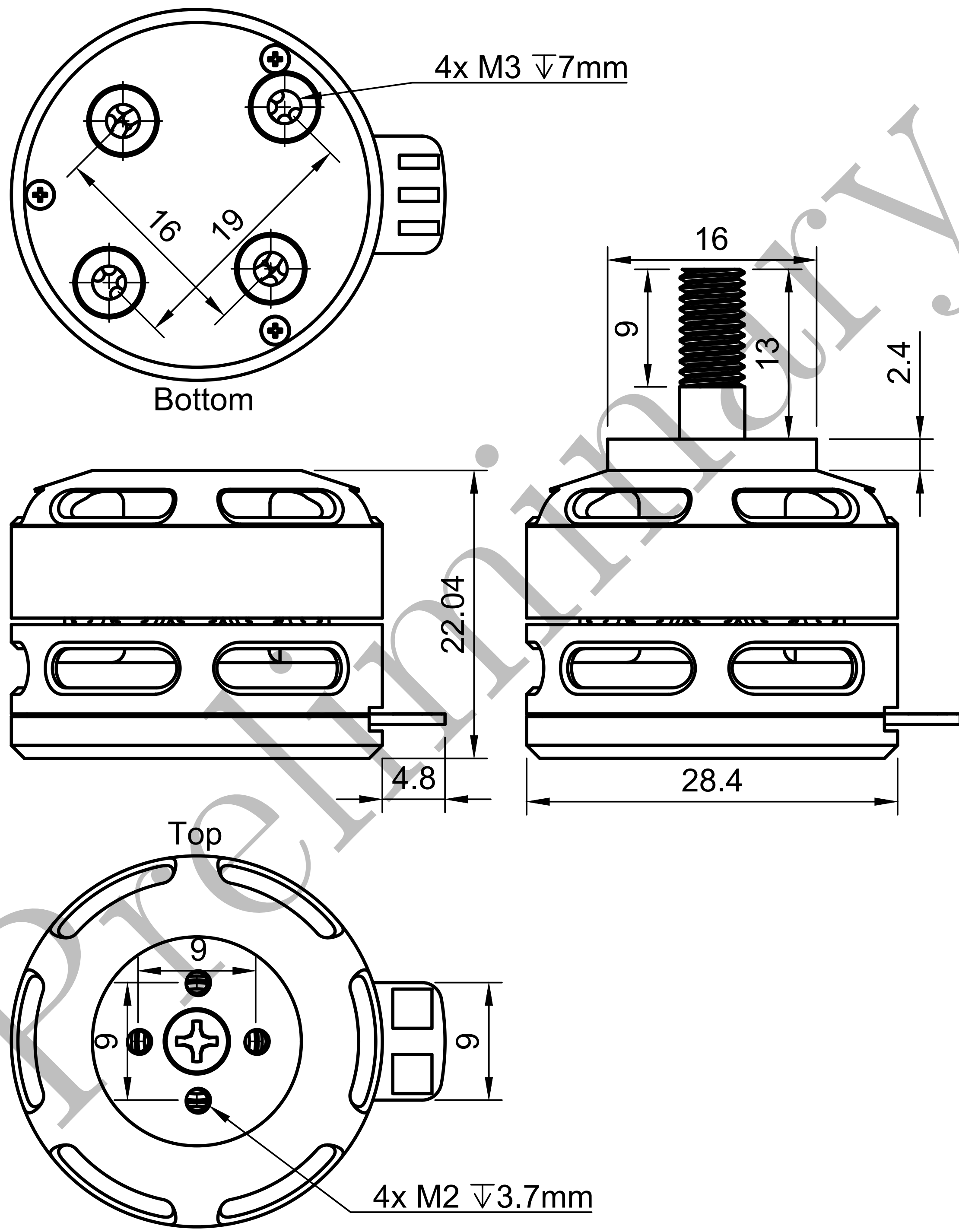
5.2 Communication Wiring

The standard communication connector is a JR type servo connector with 15cm of wire. These connectors have 0.1in spacing and can be inserted into standard perfboard and breadboards with a 3x1 0.1in male-to-male header. Black is ground, red is motor controller RX (host TX), white is motor controller TX (host RX).

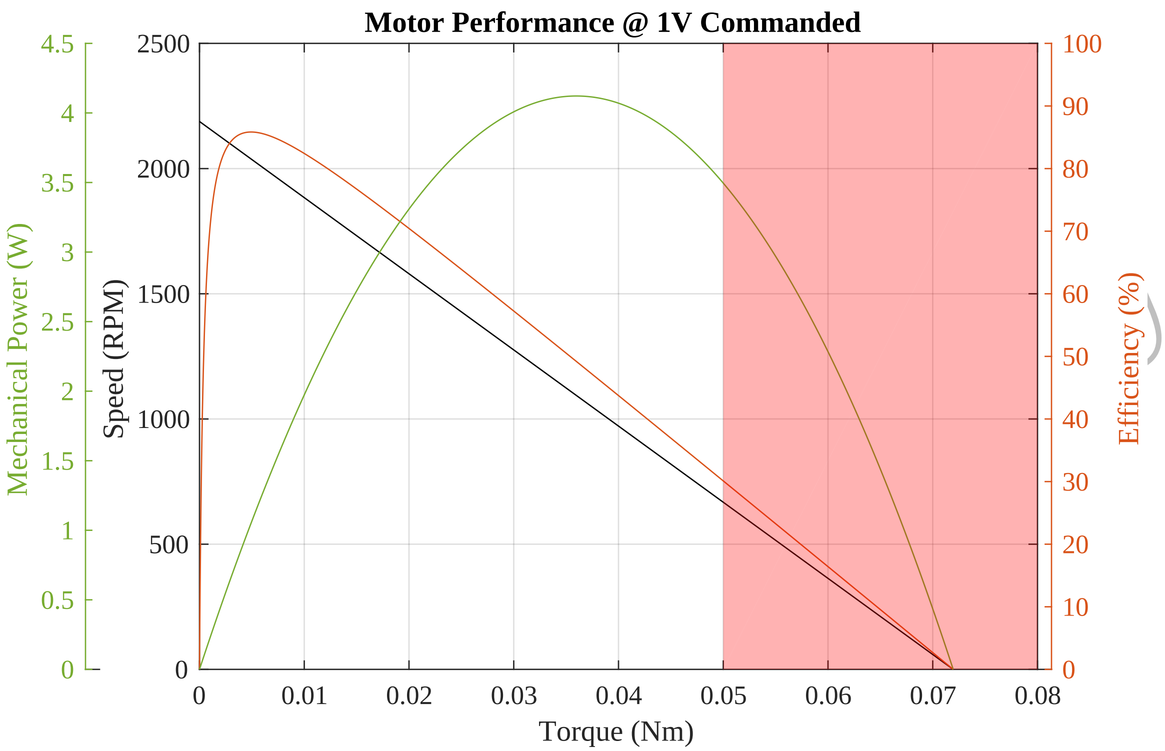
6 Motor Specifications

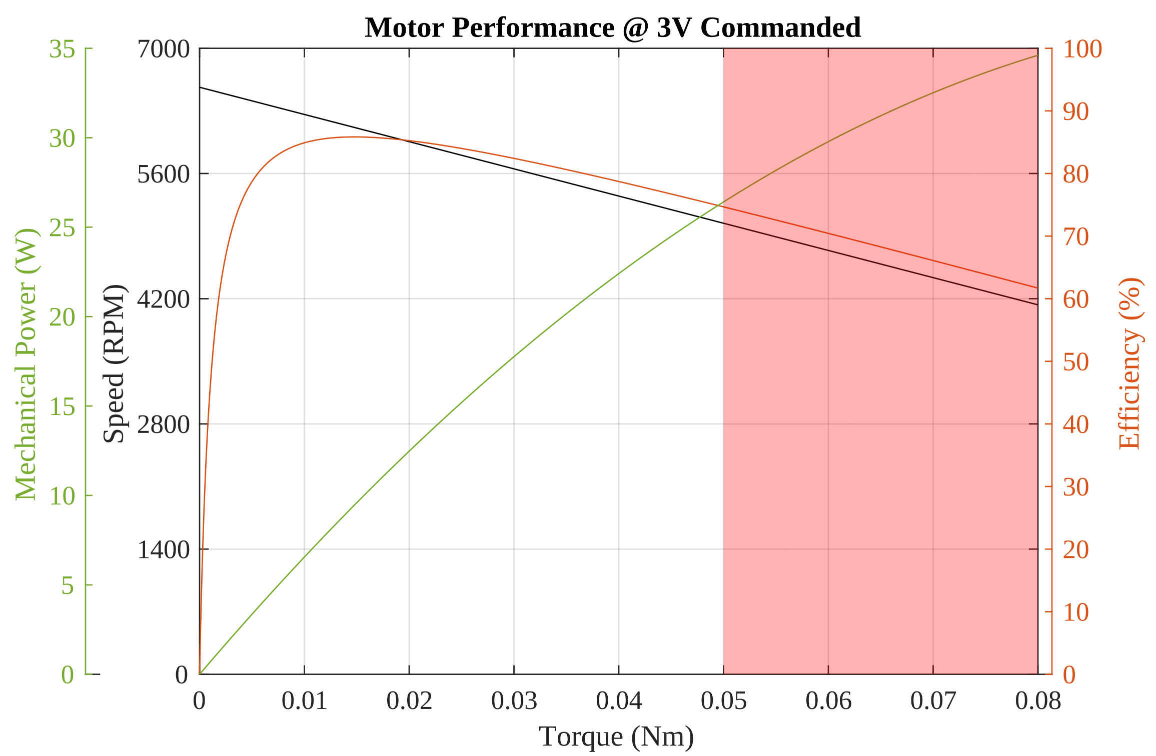
Description	Symbol	Value	Unit	Notes
Speed Constant	K_v	2200	RPM/V	
Torque/EMF Constant	K_t/K_e	0.0043	$\mathrm{N\,m\,A^{-1}}$	
Resistance	R	0.06	Ω	
Mass	m	45.7	g	
Torque	τ_c	50	N mm	Continuous, in moving air
Torque	τ_b	80	N mm	Burst, 3s, in moving air
No Load Speed	ω_0	2344	$\mathrm{rad\,s^{-1}}$	$@V_{CC} = 10\,\mathrm{V}$
No Load Current	I_0	0.9	A	$@V_{CC} = 10\,\mathrm{V}$

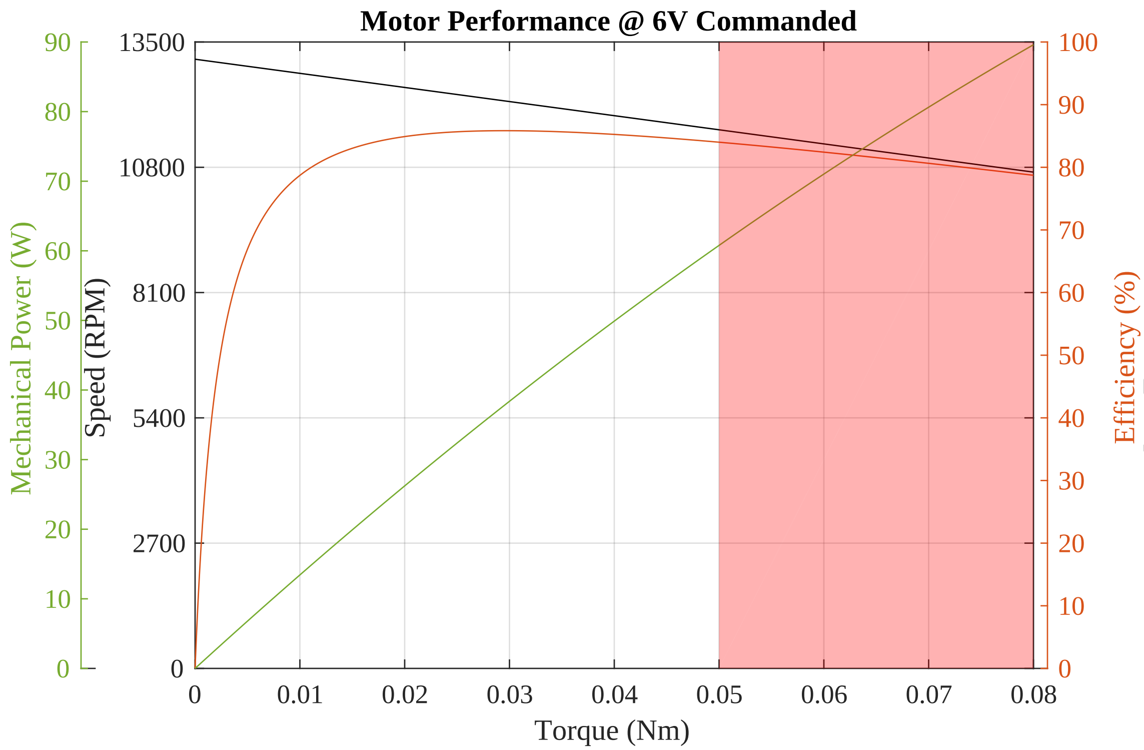
7 Mechanical Interface

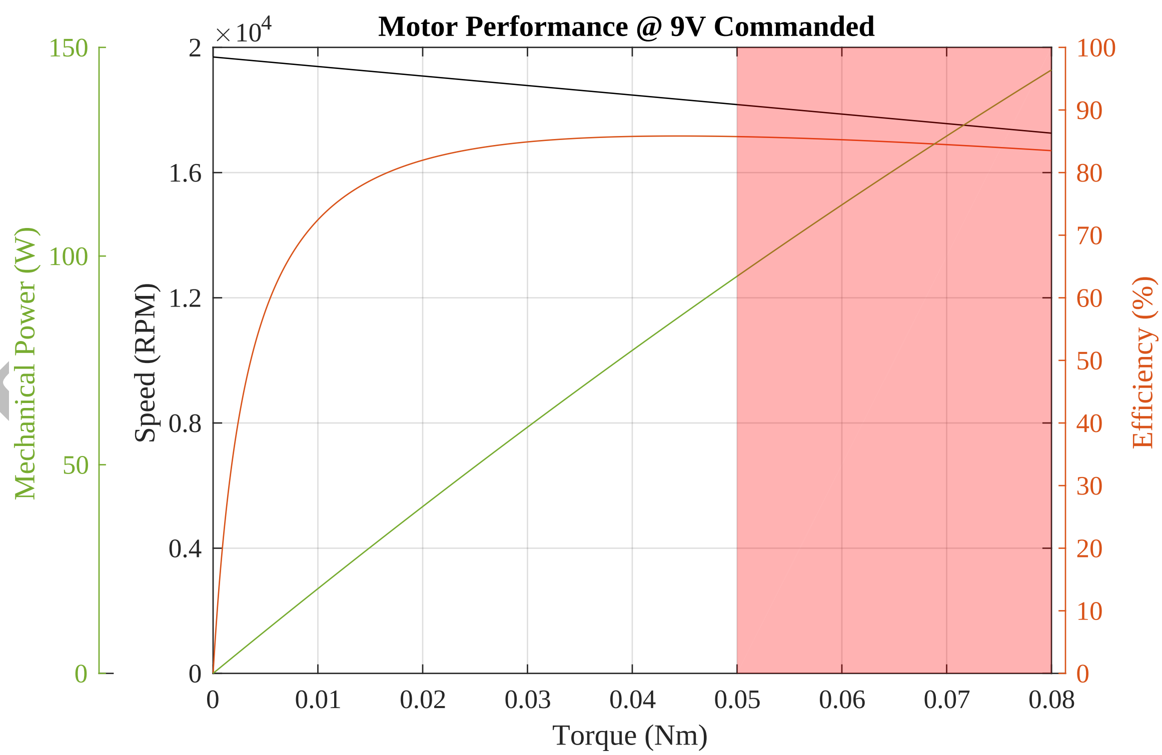


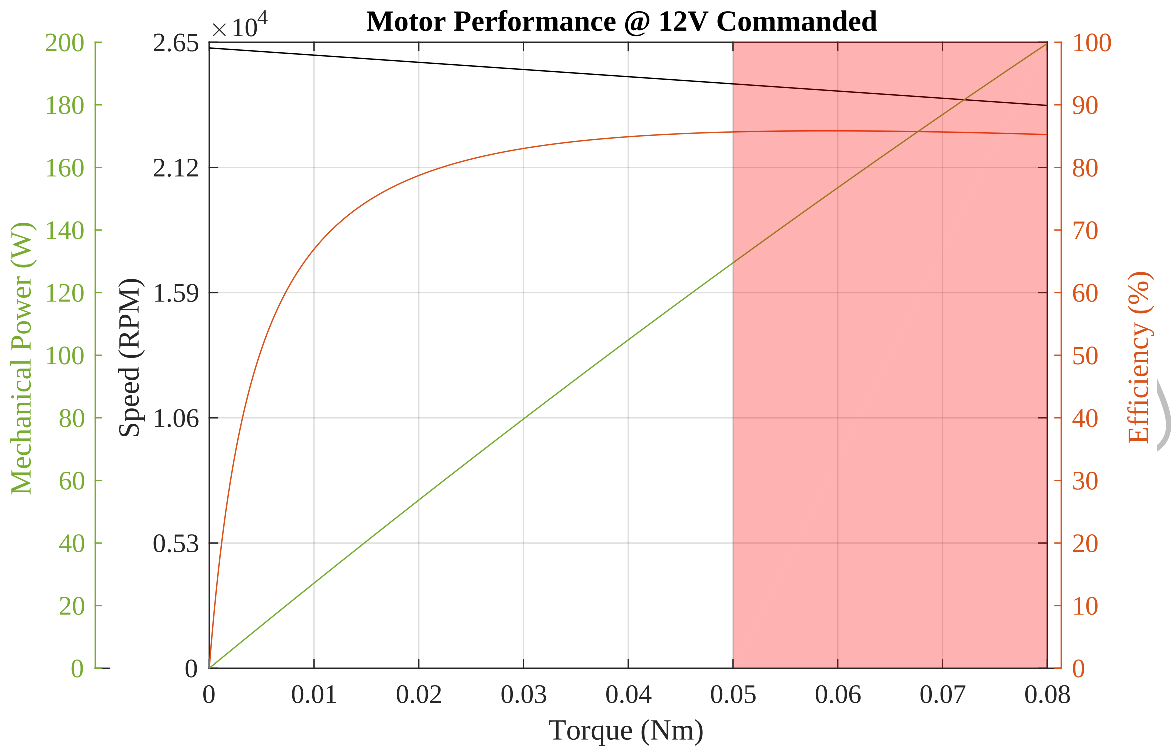
8 Motor Performance

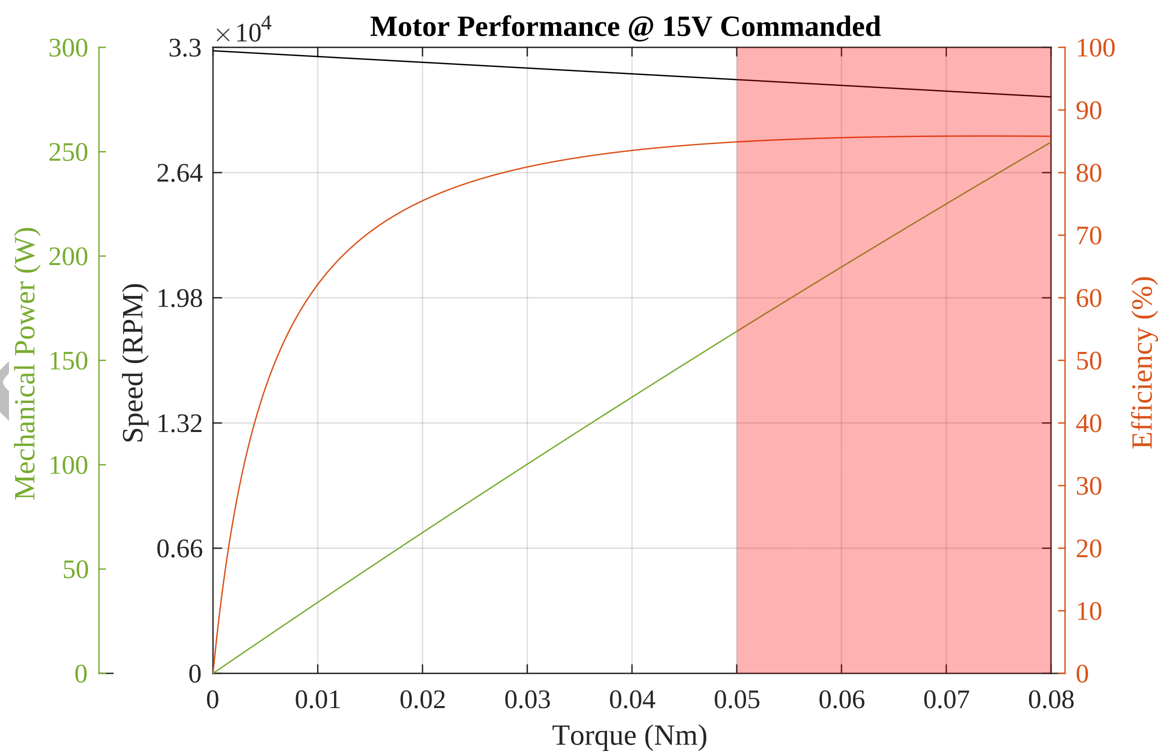





Motor Performance @ 6V Commanded
Mechanical Power (W)
0
10
20
30
40
50
60
70
80
90
Speed (RPM)
0
2700
5400
8100
10800
13500
Efficiency (%)
0
10
20
30
40
50
60
70
80
90
100
Torque (Nm)
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08


Motor Performance @ 9V Commanded
Mechanical Power (W)
0
50
100
150
Speed (RPM)
$\times 10^4$
0
0.4
0.8
1.2
1.6
2
Efficiency (%)
0
10
20
30
40
50
60
70
80
90
100
Torque (Nm)
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08


Motor Performance @ 12V Commanded
×10^4
Mechanical Power (W)
200
180
160
140
120
100
80
60
40
20
0
Speed (RPM)
2.65
2.12
1.59
1.06
0.53
0
Efficiency (%)
100
90
80
70
60
50
40
30
20
10
0
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
Torque (Nm)


Motor Performance @ 15V Commanded
×10^4
Mechanical Power (W)
300
250
200
150
100
50
0
Speed (RPM)
3.3
2.64
1.98
1.32
0.66
0
Efficiency (%)
100
90
80
70
60
50
40
30
20
10
0
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
Torque (Nm)

9 UART Messaging

9.1 Introduction

IQinetics uses a fully featured, serial based protocol for communicating with motor controllers. This communication protocol is broken into classes of related functionality. As such, IQinetics supplies libraries for communicating with the motor controllers in the object-oriented languages C++ (C++11 standard) and Matlab.

9.2 C++ Libraries

The C++ libraries 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()
```

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

9.2.1 Arduino

These C++ libraries are compatible with Arduino. To use them, copy all files in IQinetics_cpp/inc and IQinetics_cpp/src into a single folder. See the instructions on this page to install that folder as a library: https://www.arduino.cc/en/Guide/Libraries. Usage is identical to the C++ documentation.

Below is a complete example Arduino sketch:

```
/*
*  IQinetics serial communication example.
*
*  Turns off the LED when the motor's position is under pi.
*  Turns on the LED when the motor's position is over pi.
*
*  The circuit:
*    LED attached from pin 13 to ground
*    Arduino RX is directly connected to motor TX
*    Arduino TX is directly connected to motor RX
*
*  Created 2016/12/28 by Matthew Piccoli
*
*  This example code is in the public domain.
*/

// Includes required for communication
// Message forming interface
#include "generic_interface.hpp"
// Client that speaks to the encoder
#include "encoder_client.hpp"

// LED pin
const int kLedPin =  13;

// This buffer is for passing around messages.
// We use one buffer here to save space.
uint8_t communication_buffer[256];
// Stores length of message to send or receive
uint8_t communication_length;

// Time in milliseconds since we received a packet
unsigned long communication_time_last;

// Make a communication interface object
GenericInterface com;
// Make an Encoder object with obj_id 0
EncoderClient encoder_client(0);

void setup() {
  // Initialize the Serial peripheral
  Serial1.begin(115200);
  // Initialize the LED pin as an output:
  pinMode(kLedPin, OUTPUT);

  // Initialize communication time
  communication_time_last = millis();
}
```



```
void loop() {

  // Puts an radian request message in the outbound com queue
  encoder_client.rad_.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);
  }

  // wait a bit so as not to send massive amounts of data
  delay(100);

  // Reads however many bytes are currently available
  communication_length = Serial1.readBytes(communication_buffer, Serial1.available());
  // Puts the recently read bytes into com's 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))
  {
    // Remember time of received packet
    communication_time_last = millis();

    // Share that packet with all client objects
    encoder_client.ReadMsg(com,rx_data,rx_length);

    // Once we're done with the message packet, drop it
    com.DropPacket();
  }

  // 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.
  if(encoder_client.rad_.IsFresh()) {
    // Check if position is above pi
    if (encoder_client.rad_.get_reply() > 3.14f) {
      // turn LED on:
      digitalWrite(kLedPin, HIGH);
    }
    else {
      // turn LED off:
      digitalWrite(kLedPin, LOW);
    }
  }

  // If we haven't heard from the motor in 250 milliseconds
```

```
  if(millis() - communication_time_last > 250)
  {
    // Toggle the LED
    // Should flash at 5 hz thanks to the delay(100) above
    digitalWrite(kLedPin, !digitalRead(kLedPin));
  }
}
```

9.3 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.

9.4 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.

9.4.1 C++

To use Propeller Motor Controller in C++, include propeller_motor_control_client.hpp. This allows the creation of a PropellerMotorControlClient object. See Table 1 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 thrust = 1.0f; // N

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_thrust_.set(com, thrust);

// Insert code for interfacing with hardware here

}
```

9.4.2 Matlab

To use Propeller Motor Controller in Matlab, all IQinetics communication code must be included in your path. This allows the creation of a PropellerMotorControlClient object. See Table 1 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_thrust',1.0);
```

9.5 Safe Brushless Drive

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

Table 1: 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	$T = k_t\omega$ thrust constant in positive direction
14	propeller_kt_neg	float	N/(rad/s)^2	$T = k_t\omega$ 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

9.5.1 C++

To use Safe Brushless Drive in C++, include safe_brushless_drive.hpp. This allows the creation of a SafeBrushlessDriveClient 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 SafeBrushlessDriveClient is:

```
#include "generic_interface.hpp"
#include "safe_brushless_drive_client.hpp"

float torque;

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

// Make a Safe Brushless Drive object with obj_id 0
SafeBrushlessDriveClient drive;

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

// Insert code for interfacing with hardware here

// Read response
torque = drive.est_motor_torque_.get_reply();
}
```


9.5.2 Matlab

To use Safe Brushless Drive in Matlab, all IQinetics communication code must be included in your path. This allows the creation of a SafeBrushlessDriveClient 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 SafeBrushlessDriveClient is:

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

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

% Use the SafeBrushlessDriveClient object
torque = drive.get('est_motor_torque');
drive.set('drive_volts',1);
```

Table 2: Type ID 50: Safe 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] (D = 0)
4	drive_spin_volts	float	V	Spins motor with this voltage (D = 0)
5	drive_brake			
6	drive_coast			
7	drive_angle_offset	float	rad	
8	drive_pwm	float	pwm	Applies this throttle [-1, 1] to motor (DQ unchanged)
9	drive_volts	float	V	Applies this voltage to motor (DQ unchanged)
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

9.6 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 is translated. Inputs can be mapped to PWM control, voltage control, velocity control, and thrust control. Furthermore, inputs can be mapped linearly or can be square rooted, which more closely maps the output thrust of a propeller linearly with the input PWM. Finally, values can be interpreted as signed/unsigned and clockwise/counter clockwise.

9.6.1 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 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 EscPropellerInputParserClient is:

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

float thrust_max = 5.0f; // N

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.thrust_max_.set(com, thrust_max);

// Insert code for interfacing with hardware here

}
```

9.6.2 Matlab

To use ESC Propeller Input Parser in Matlab, all IQinetics communication code must be included in your path. This allows the creation of a EscPropellerInputParserClient 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 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);
```

9.7 Encoder

The Encoder object reads motor positions from the encoder then estimates and filters the motor velocity.

Table 3: 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 PWM value
2	map	uint8	enum	0 = linear, 1 = sqrt
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, input PWM scaled to [-supply, supply] or [0, supply]
5	velocity_max	float	rad/s	Maximum angular velocity command, input PWM scaled to [-velocity_max, velocity_max] or [0, velocity_max]
6	thrust_max	float	N	Maximum thrust command (requires kt values), input PWM scaled to [-thrust_max, thrust_max] or [0, thrust_max]

9.7.1 C++

To use the Encoder in C++, include encoder_client.hpp. This allows the creation of an EncoderClient 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 EncoderClient is:

```
#include "generic_interface.hpp"
#include "encoder_client.hpp"

float velocity_filtered;

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

// Make a Buffered Voltage Monitor object with obj_id 0
EncoderClient encoder(0);

// Use the Buffered Initialized Encoder object
encoder.velocity_.get(com);
encoder.filter_fc_.set(com,100);

// Insert code for interfacing with hardware here

// Read response
velocity_filtered = encoder.velocity_.get_reply();
}
```

9.7.2 Matlab

To use the Encoder object in Matlab, all IQinetics communication code must be included in your path. This allows the creation of an EncoderClient 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 EncoderClient is:

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

```
% Make an Encoder object with obj_id 0
encoder = EncoderClient('com',com);

% Use the Encoder object
velocity_filtered = encoder.get('velocity');
encoder.set('filter_fc',100);
```

Table 4: Type ID 53: Encoder

Sub ID	Short Name	Data Type	Unit	Note
0	zero_angle	float	rad	Angle from absolute to incremental
1	velocity_filter fs	uint32	Hz	Filter sample frequency
2	velocity_filter fc	uint32	Hz	Filter cutoff frequency
3	rev	uint32	Rev32	Position in UQ32 format, 0 to 1, with zero_angle
4	absolute_rev	uint32	Rev32	Position in UQ32 format, 0 to 1, without zero_angle
5	rad	float	rad	Position with zero_angle
6	absolute_rad	float	rad	Position without zero_angle
7	velocity	float	rad/s	Filtered velocity

9.8 System Control

System Control allows the user to perform low level tasks on the motor controller's microcontroller and gather basic information.

9.8.1 C++

To use System Control in C++, include system_control_client.hpp. This allows the creation of a SystemControlClient 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 SystemControlClient is:

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

uint16_t mem_size;

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.mem_size_.get(com);
  system_control.reboot_program_.set(com);

  // Insert code for interfacing with hardware here

  // mem_size = system_control.mem_size_.get_reply();
}
```


9.8.2 Matlab

To use System Control in Matlab, all IQinetics communication code must be included in your path. This allows the creation of a SystemControlClient 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 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
system_control.get('mem_size');
system_control.set('reboot_program');
```

Table 5: 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

9.9 Serial Interface

The Serial client allows the user to change settings related to the serial communication interface, such as baud rate.

9.9.1 C++

To use Serial Interface in C++, include serial_interface_client.hpp. This allows the creation of a SerialInterfaceClient 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 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();
}
```

9.9.2 Matlab

To use Serial Interface in Matlab, all IQinetics communication code must be included in your path. This allows the creation of a SerialInterfaceClient 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 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
serial_interface.get('baud_rate');
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 save the new baud rate
com = MessageInterface('COM18',9600);
serial_interface = SerialInterfaceClient('com',com);
serial_interface.save('baud_rate');
```

Table 6: 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.