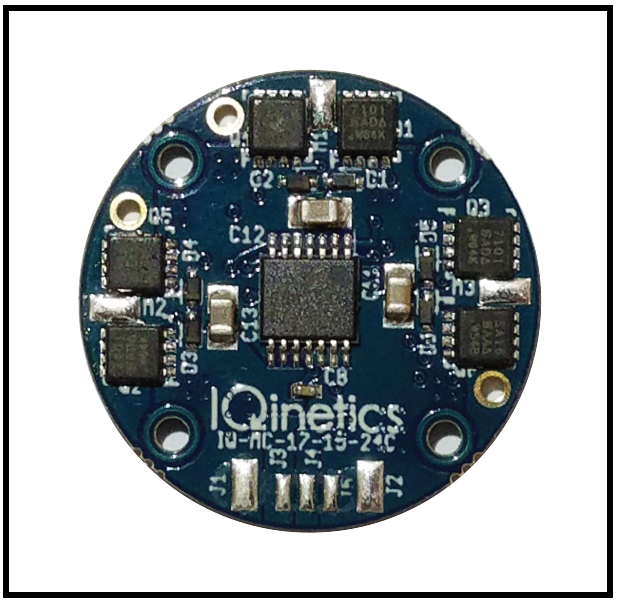
Three Phase Brushless Motor Controller/Driver 17 V 15 A

1 Features

- Controls one three-phase brushless motor
- Supply voltage 5-17 V DC[1]
- Output current 15 A[1]
- Sensored vector control with space vector PWM
- ±28000 RPM Max
- Precise to 0.022°
- Position control with tunable PID
- Velocity control with tunable PID + 2nd order polynomial feed forward
- Torque control[2]
- Current control[2]
- Voltage control
- Coast and brake modes
- Regenerative braking
- Active freewheeling
- Built-in current limit[2]
- Serial (UART), I²C[3], 1-2ms PWM[3], OneShot125[4], OneShot42[4], MultiShot[4]
- OPTION: Anticogging torque ripple reduction



2 Applications

- UAV propeller, reversible/3D capable
- Gimbals
- 3D printers
- Robotics
- Haptic devices

3 Description

The IQ-MC-17-15-24C-H is a sensored, three phase brushless motor controller. It utilizes a high resolution magnetic rotary position sensor and a powerful ARM Cortex-M4 processor. The IQ-MC-17-15-24C-H commutes motors smoothly at any speed between 0 and 28,000 RPM in either direction. The sensored vector control with space vector PWM is up to twice as efficient as sensorless electronic speed controllers that use block commutation. Furthermore, this controller can output 4.8% more shaft power compared to block commutation controllers in voltage limited applications. The controller is capable of communicating via UART (3.3 V logic level serial), I²C, 1-2 ms PWM, and OneShot125 PWM.

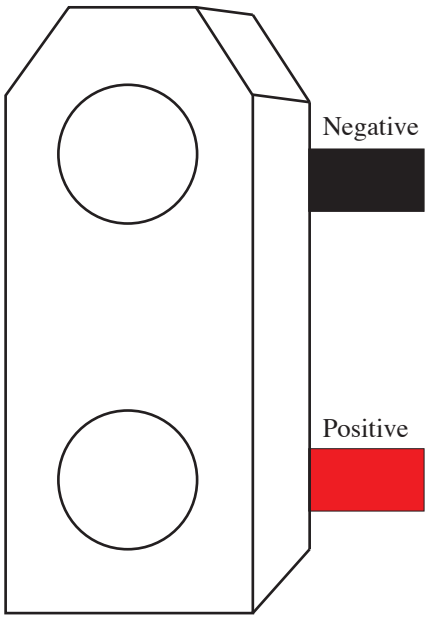
With the Anticogging option the torque ripple from cogging torque is effectively reduced by up to 88%.

[1]Subject to change
[2]Estimated, ±20%
[3]In development
[4]Planned

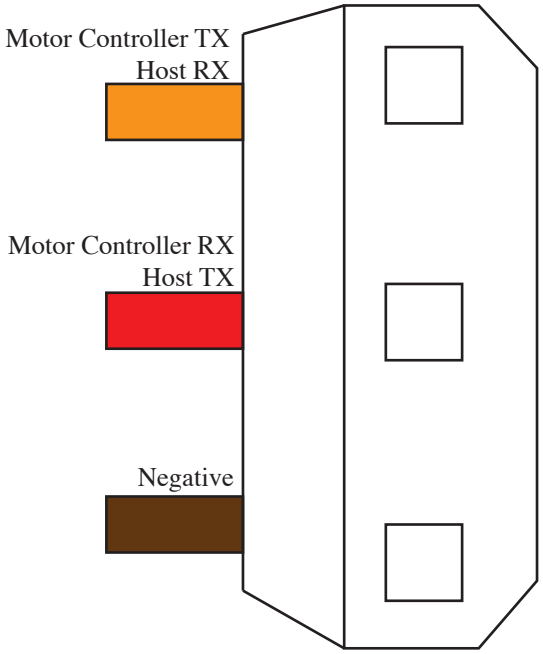
4 Absolute Maximum Ratings

Description	Symbol	Min	Max	Unit	Notes
Supply Voltage	V_{CC}	0	17	V	Designed for 12.6 V = 3S LiPo [5]
Continuous Stall Current	I_S	-	11.1	A	
Continuous Rotating Current	I_{SR}	-	15	A	
Pulsed Stall Current	I_{SP}	-	60	A	
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 IQ-MC-17-15-24C-Hvia an XT-30 connector and 6cm 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 12in 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. Brown is ground, red is motor controller RX (host TX), orange is motor controller TX (host RX).

[5] Usable on 4S LiPo systems as long as extreme care is taken to ensure there are no no voltage spikes at the motor controller. A Zener diode with a Zener voltage below 17 V is recommended for power supply and other applications that cannot absorb energy from motor regeneration. We recommend a 1N5351B for 12 V systems that may experience light regeneration.

6 Messaging

6.1 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.

6.2 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.

6.2.1 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 complex motor controllers
#include <complex_motor_control_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 a complex motor control object
ComplexMotorControlClient motor_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 absolute angle request message in the outbound com queue
  motor_client.obs_absolute_angle_.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
    motor_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(motor_client.obs_absolute_angle_.IsFresh()) {
    // Check if position is above pi
    if (motor_client.obs_absolute_angle_.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));
  }
}
```

6.3 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.

6.4 Buffered Initialized Encoder

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

6.4.1 C++

To use Buffered Initialized Encoder in C++, include buffered_Initialized_Encoder.hpp. This allows the creation of a BufferedInitializedEncoderClient 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 BufferedInitializedEncoderClient is:

```
#include "generic_interface.hpp"
#include "buffered_initialized_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
BufferedInitializedEncoderClient 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();
}
```

6.4.2 Matlab

To use Buffered Initialized Encoder in Matlab, all IQinetics communication code must be included in your path. This allows the creation of a BufferedInitializedEncoderClient 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 BufferedInitializedEncoderClient is:

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

% Make a Buffered Initialized Encoder object with obj_id 0
encoder = BufferedInitializedEncoderClient('com',com);

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

6.5 Buffered Voltage Monitor

The Buffered Voltage Monitor samples and filters the input voltage to the motor controller.

Table 1: Type ID 53: Buffered Initialized 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

6.5.1 C++

To use Buffered Voltage Monitor in C++, include buffered_voltage_monitor.hpp. This allows the creation of a BufferedVoltageMonitorClient 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 BufferedVoltageMonitorClient is:

```
#include "generic_interface.hpp"
#include "buffered_voltage_monitor_client.hpp"

float volts_filtered;

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

// Make a Buffered Voltage Monitor object with obj_id 0
BufferedVoltageMonitorClient volt_monitor(0);

// Use the Buffered Voltage Monitor object
volt_monitor.volts_.get(com);
volt_monitor.filter_fc_.set(com,100);

// Insert code for interfacing with hardware here

// Read response
volts_filtered = volt_monitor.volts_.get_reply();
}
```

6.5.2 Matlab

To use Buffered Voltage Monitor in Matlab, all IQinetics communication code must be included in your path. This allows the creation of a BufferedVoltageMonitorClient 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 BufferedVoltageMonitorClient is:

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

% Make a Buffered Voltage Monitor object with obj_id 0
volt_monitor = BufferedVoltageMonitorClient('com',com);
```

```
% Use the Buffered Voltage Monitor object
volts_filtered = volt_monitor.get('volts');
volt_monitor.set('filter_fc',100);
```

Table 2: Type ID 42: Buffered Voltage Monitor

Sub ID	Short Name	Data Type	Unit	Note
0	volts_raw	float	V	Unfiltered voltage
2	volts	float	V	Filtered Voltage
11	volts_gain	float		
16	filter_fs	uint32	Hz	Filter sample frequency
17	filter_fc	uint32	Hz	Filter cutoff frequency

6.6 System Control

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

6.6.1 C++

To use System Control in C++, include system_control_client.hpp. This allows the creation of a SystemControlClient 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 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();
}
```

6.6.2 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 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 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 3: 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

6.7 Complex Motor Control

Complex Motor Control is a multi-mode motor motion controller. It exposes 3 closed loop controllers–angle, velocity, current[6]. It also provides access to a number of open loop control modes.

6.7.1 C++

To use Complex Motor Control in C++, include complex_motor_control.hpp. This allows the creation of a ComplexMotorControlClient 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 ComplexMotorControlClient is:

```
#include "generic_interface.hpp"
#include "complex_motor_control_client.hpp"

float abs_angle;

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

[6]Closed loop available on some models, open loop on the remaining models

```
// Make a Complex Motor Control object with obj_id 0
ComplexMotorControlClient motor_control(0);

// Use the Complex Motor Control object
motor_control.obs_absolute_angle_.get(com);
motor_control.cmd_angle_.set(com,3.14f);

// Insert code for interfacing with hardware here

// Read response
abs_angle = motor_control.obs_absolute_angle_.get_reply();
}
```

6.7.2 Matlab

To use Complex Motor Control in Matlab, all IQinetics communication code must be included in your path. This allows the creation of a ComplexMotorControlClient 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 ComplexMotorControlClient is:

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

% Make a Complex Motor Control object with obj_id 0
motor_control = ComplexMotorControlClient('com',com);

% Use the System Control object
abs_angle = motor_control.get('obs_absolute_angle');
motor_control.set('cmd_angle',3.14);
```


Table 4: Type ID 51: Complex Motor Control

Sub ID	Short Name	Data Type	Unit	Note
0	cmd_mode	uint8	enum	0 = phase pwm, 1 = phase volts, 2 = spin pwm, 3 = spin volts, 4 = brake, 5 = coast, 6 = calibrate, 7 = velocity, 8 = angle, 9 = spin amps
1	cmd_phase_pwm	float	pwm	-1 to 1
2	cmd_phase_volts	float	V	-supply voltage to supply voltage
3	cmd_spin_pwm	float	pwm	-1 to 1
4	cmd_spin_volts	float	V	-supply voltage to supply voltage
5	cmd_brake			
6	cmd_coast			
7	cmd_calibrate	float	pwm	
8	cmd_velocity	float	rad/s	Sets target speed for velocity controller
9	cmd_angle	float	rad	Sets target angle for position controller
10	drive_pwm	float	pwm	The pwm currently used
11	drive_volts	float	V	The voltage currently used
12	mech_lead_angle	float	rad	
13	obs_supply_volts	float	V	Observed supply voltage
14	obs_supply_amps	float	V	Observed supply amperage
15	obs_angle	float	V	Observed angle using zero_angle
16	obs_absolute_angle	float	V	Observed angle ignoring zero_angle
17	obs_velocity	float	V	Observed velocity
18	motor_pole_pairs	uint16		
19	motor_emf_shape	uint8		
20	motor_Kv	float	RPM/V	
21	motor_R_ohm	float	ohm	
22	motor_I_max	float	A	Software current limit
23	permute_wires	uint8	bool	
24	calibration_angle	float	rad	
25	lead_time	float	s	
26	commutation_hz	uint32	Hz	
27	control_hz	uint32	Hz	
28	phase_angle	float	rad	
29	calibration_time	float	s	
30	velocity_filter_fc	uint32	Hz	Cutoff frequency for velocity filter
31	velocity_filter_value	float	rad/s	Observed velocity with filter
32	velocity_Kp	float	V/(rad/s)	Velocity control proportional gain
33	velocity_Ki	float	V/(rad)	Velocity control integral gain
34	velocity_Kd	float	V/(rad/s^2)	Velocity control derivative gain
35	velocity_ff0	float	V	Velocity control constant feed forward
36	velocity_ff1	float	V/(rad/s)	Velocity control linear feed forward
37	velocity_ff2	float	V/(rad/s)^2	Velocity control quadratic feed forward
38	angle_Kp	float	V/(rad)	Position control proportional gain
39	angle_Ki	float	V/(rad*s)	Position control integral gain
40	angle_Kd	float	V/(rad/s)	Position control derivative gain
42	est_motor_amps	float	A	Estimated motor amperage
43	est_motor_torque	float	Nm	Estimated motor torque
44	obs_motor_amps	float	A	Observed motor amperage
45	ctrl_spin_amps	float	A	-motor_I_max to motor_I_max
46	ctrl_spin_torque	float	Nm	
47	amps_Kp	float	V/A	Current control proportional gain
48	amps_Ki	float	V/(A*s)	Current control integral gain
49	amps_Kd	float	V/(A/s)	Current control derivative gain
50	volts_limit	float	V	Maximum regen voltage