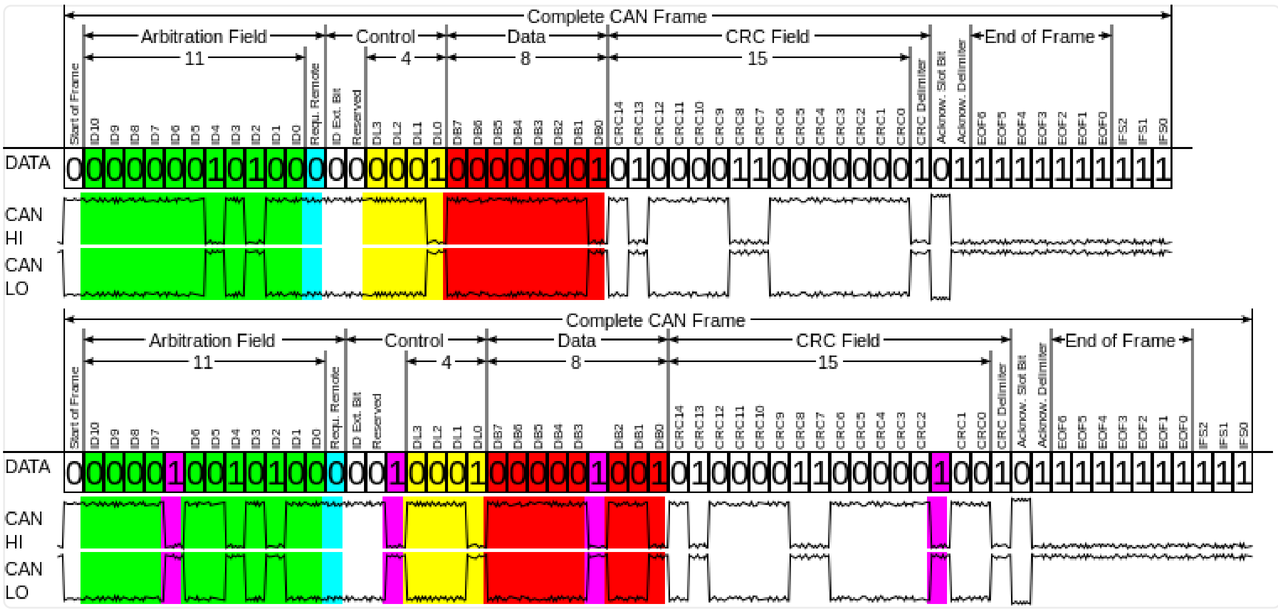
Control with CAN

1. VESC CAN Message Structure

English Korean

We analyze the CAN protocol used in the VESC firmware. VESC CAN communication follows a CAN communication protocol called Extended ID (EID) or Extended Format. While Base Format consists of an ID value of 11 bits, Extended Format consists of an ID value of 29 bits. Therefore, VESC sends controller_id and command together as ID values, which are described in more detail below.

- Comparison between Base Frame format vs. Extended Format

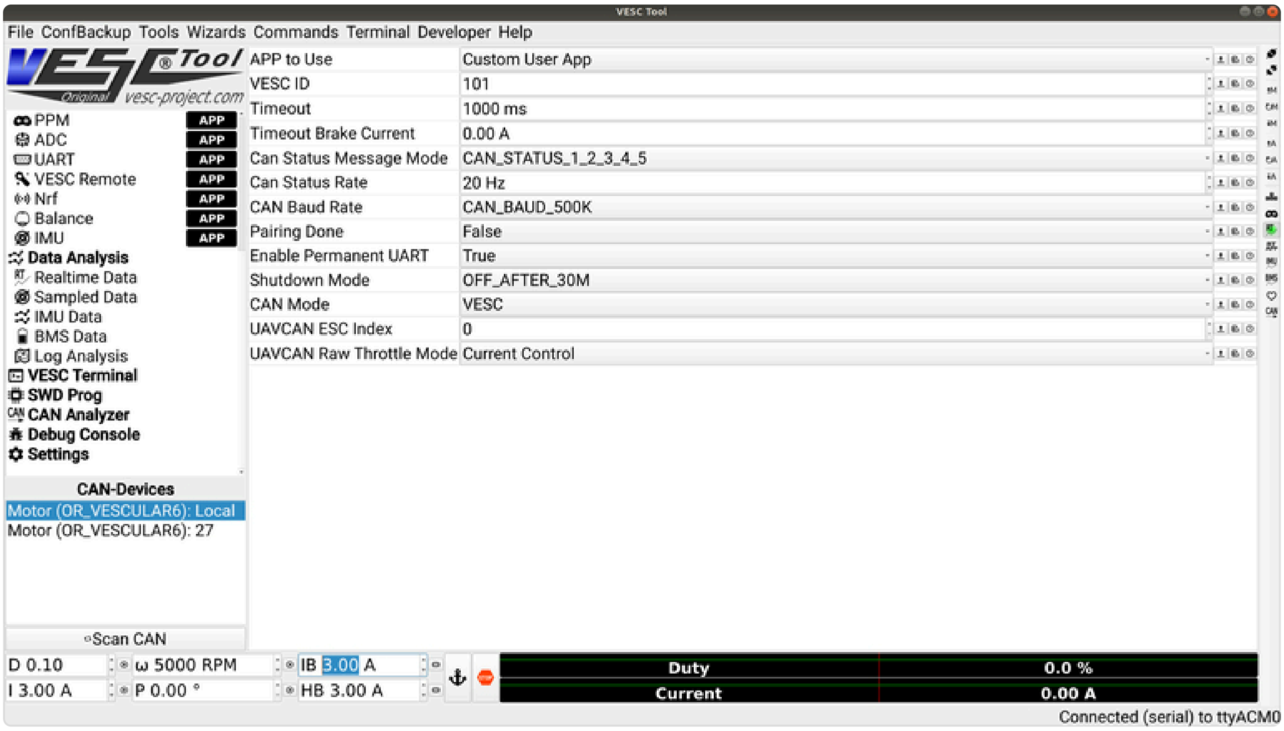


https://en.wikipedia.org/wiki/CAN_bus

Base frame format: with 11 identifier bits / Extended frame format: with 29 identifier bits

1. Periodically Sending Messages

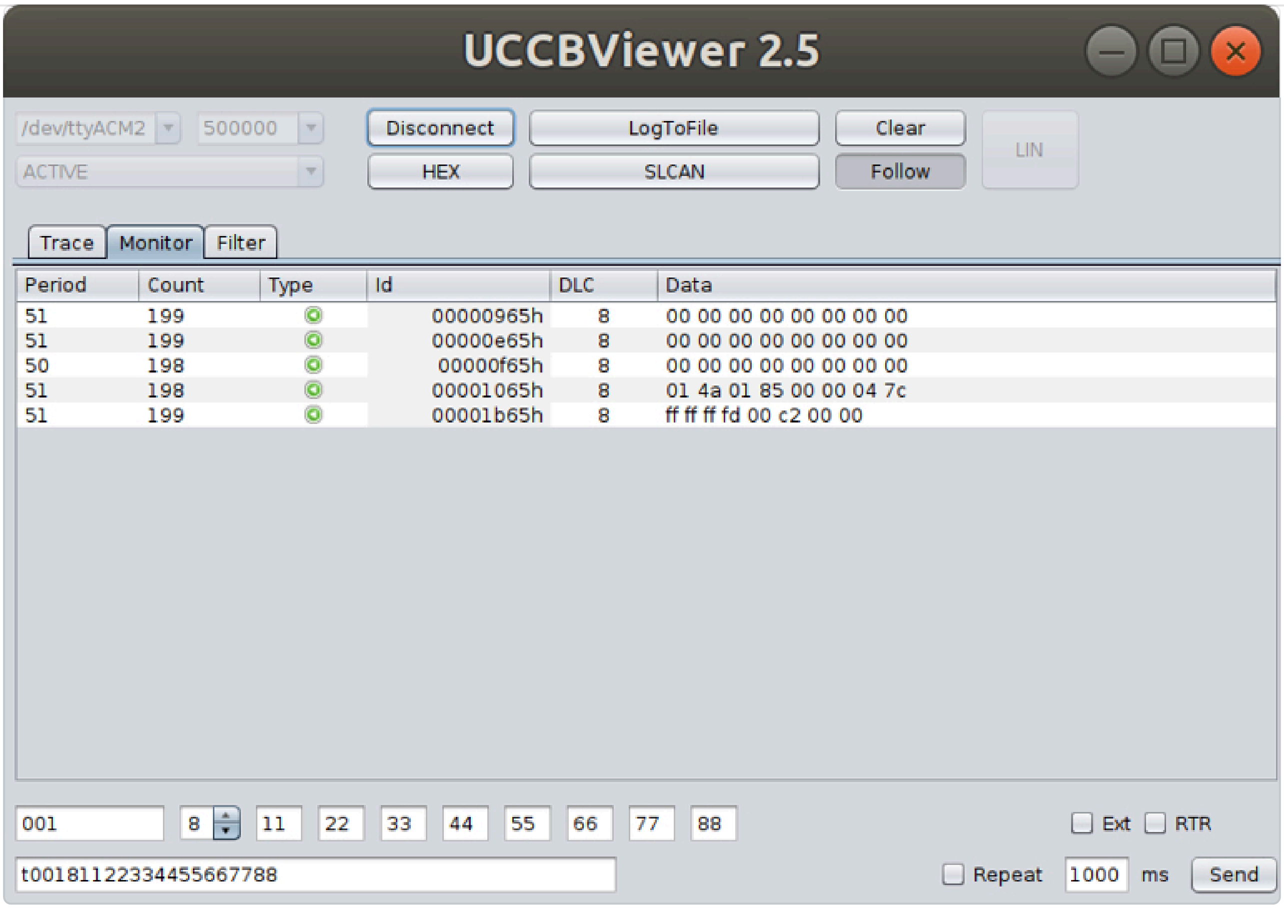
In VESC CAN Protocol, there is can status message which is periodically sent(aka Telemetry). You can check the rate and message mode in VESC-Tool. Currently, there are two VESCs connected on same CAN bus (local controller_id is 101 and the other one is 27) and baudrate is 500Kbps. The periodically sending status msgs rate is 20Hz and the message mode is 1, 2, 3, 4, 5 which is sending 5 different type of messages.



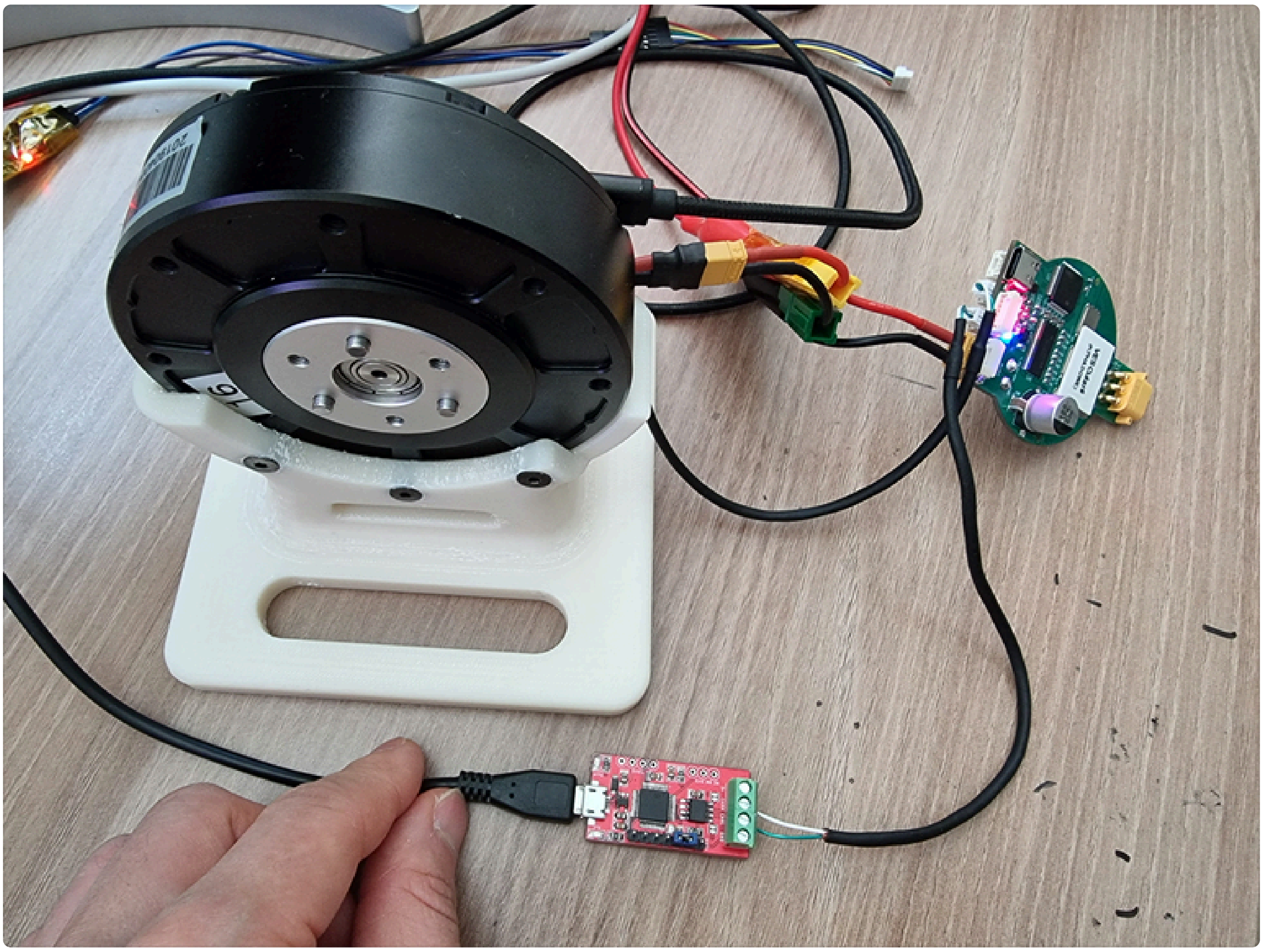
VESC CAN setting shown at VESC-Tool

If you check CAN message using CAN analyzing tool(in my case CANable board and UCCBViewer). I bought CANable board at here. UCCBViewer can be downloaded at https://github.com/UsbCANConverter-UCCbasic/uCCBViewer/releases and you can run the program using below command.

```
sudo java -jar -Dsun.java2d.uiScale=2.5 uCCBViewer-2.5.jar
```

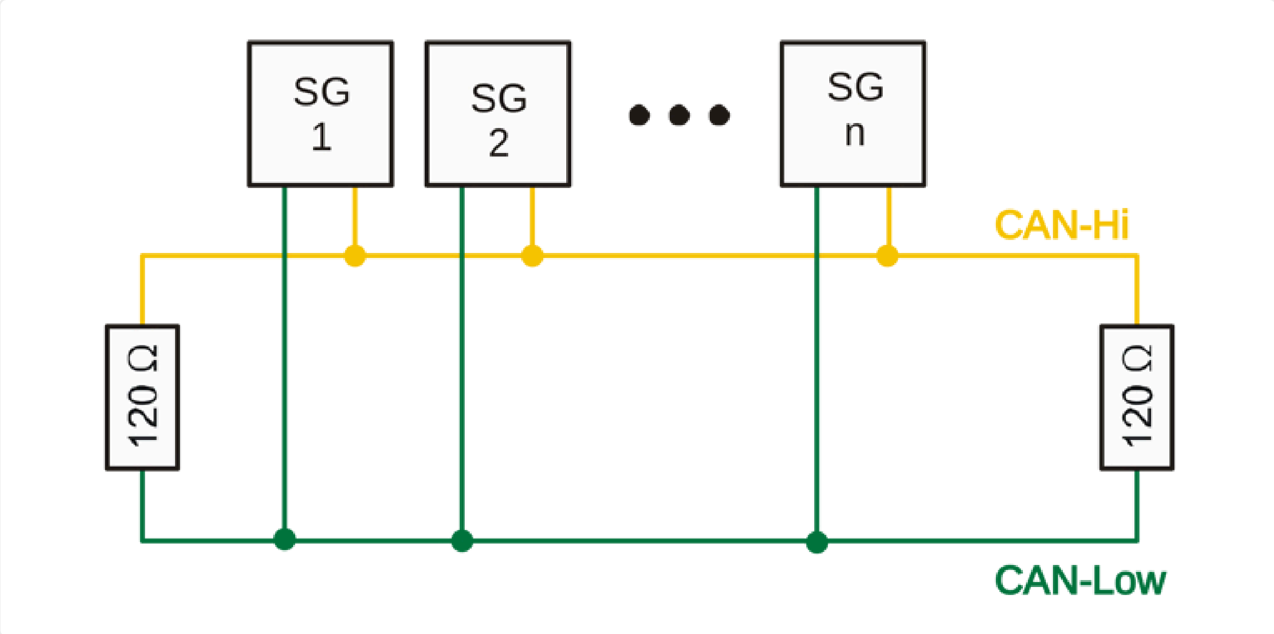


UCCBViewer 2.5 using CANable board



Experimental setup

VESC 101 is connected to PC via USB and VESC 101, 27 and CANable board are connected via CAN BUS. (Caution! How to use terminal resistor)



CAN Terminal Resistor usage

I used CANable board to observe CAN messages. The periodic message are as below.

Id	DLC	Data
00000965h	8	00 00 0c f2 00 02 00 64
00000e65h	8	00 00 00 00 00 00 00 00
00000f65h	8	00 00 00 00 00 00 00 00
00001065h	8	01 75 01 c1 00 00 2c c6
00001b65h	8	00 04 aa d5 00 c2 00 00

CAN status msg from controller_id 101 (0x65)

Let's decode Id. CAN messages from VESC use an extended ID (EID), containing the command and controller_id. The lower 1 byte(8 bits) represents the VESC controller_id and the remaining higher byte is the index value of CAN_PACKET_ID.

- 00000965h means,
  - 0x0000009 = 9(CAN_PACKET_STATUS)
  - 0x65 = 101(controller_id)
- 00000e65h means,
  - 0x000000e = 14(CAN_PACKET_STATUS_2)
  - 0x65 = 101(controller_id)
- 00000f65h means,
  - 0x000000f = 15(CAN_PACKET_STATUS_3)
  - 0x65 = 101(controller_id)

- 00001065h means,
  - 0x0000010 = 16(CAN_PACKET_STATUS_4)
  - 0x65 = 101(controller_id)
- 00001b65h means,
  - 0x000001b = 27(CAN_PACKET_STATUS_5)
  - 0x65 = 101(controller_id)

So, this message are CAN status msgs from controller_id 101. And the Data of CAN status msgs are as below

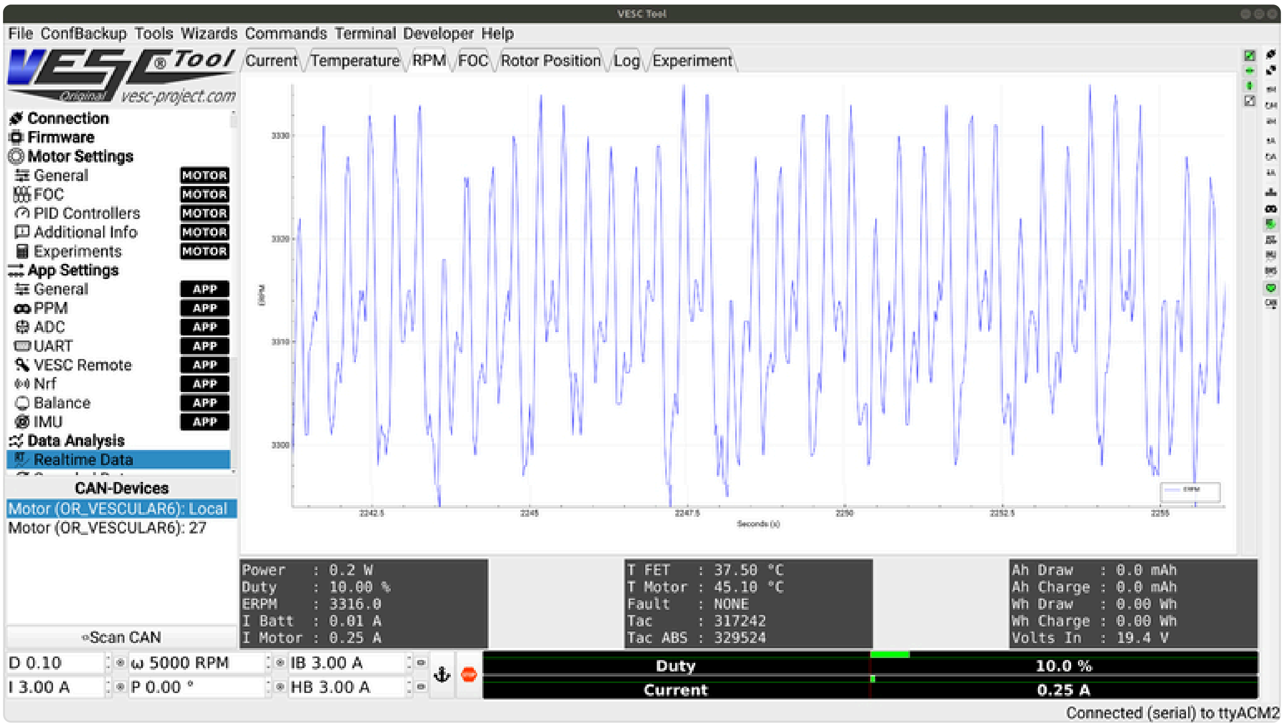
- CAN_PACKET_STATUS :
  - rpm(4 byte), current*10.0(2 byte), duty*1000.0(2 byte)
- CAN_PACKET_STATUS_2
  - amp_hours*10000.0(4 byte), amp_hours_charged*10000.0(4 byte)
- CAN_PACKET_STATUS_3
  - watt_hours*10000.0(4 byte), watt_hours_charged*10000.0(4 byte)
- CAN_PACKET_STATUS_4
  - temp_fet*10.0(2 byte), temp_motor*10.0(2 byte), current_in*10.0(2 byte), pid_pos_now*50.0(2 byte)
- CAN_PACKET_STATUS_5
  - tacho_value(4 byte), v_in*10.0(2 byte), reserved as 0(2 byte)

Let's decode data;

- CAN_PACKET_STATUS : 00 00 0c f2 00 02 00 64
  - rpm = 3314(0x00000cf4)
  - current = 0.2(0x0002)
  - duty = 0.1(0x0064)
- CAN_PACKET_STATUS_2 : 00 00 00 00 00 00 00 00
  - amp_hours = 0
  - amp_hours_charged = 0
- CAN_PACKET_STATUS_3 : 00 00 00 00 00 00 00 00
  - watt_hours = 0
  - watt_hours_charged = 0
- CAN_PACKET_STATUS_4 : 01 75 01 c1 00 00 2c c6

- temp_fet = 37.3(0x0175)
- temp_motor = 44.9(0x01c1)
- current_in = 0(0x0000)
- pid_pos_now = 229.24(0x2cc6)

- CAN_PACKET_STATUS_5 : 00 04 aa d5 00 c2 00 00
  - tacho_value = 305877(0x0004aad5)
  - v_in = 19.4(0x00c2)
  - reserved as 0(2 byte)



VESC-Tool Capture

There is a slight time difference between the data in CAN Status Msg and the data in VESC-Tool, so it is assumed that the values are not exactly the same but roughly correct.

2. Command Messages

Now, let me know the CAN messages for controlling motor.

@comm_can.c, we use below function to send can message basically.

```
void comm_can_transmit_eid_replace(uint32_t id, const uint8_t *data, uint8
    if (len > 8) {
        len = 8;
    }

#if CAN_ENABLE
#ifdef HW_HAS_DUAL_MOTORS
    if (app_get_configuration()->can_mode == CAN_MODE_VESC) {
        if (replace && ((id & 0xFF) == utils_second_motor_id() ||
                (id & 0xFF) == app_get_configuration()->controller_id)) {
            uint8_t data_tmp[10];
            memcpy(data_tmp, data, len);
            decode_msg(id, data_tmp, len, true);
            return;
        }
    }
#else
    (void)replace;
#endif

    CANTxFrame txmsg;
    txmsg.IDE = CAN_IDE_EXT;
    txmsg.EID = id;
    txmsg.RTR = CAN_RTR_DATA;
    txmsg.DLC = len;
    memcpy(txmsg.data8, data, len);

    chMtxLock(&can_mtx);
    canTransmit(&HW_CAN_DEV, CAN_ANY_MAILBOX, &txmsg, MS2ST(5));
    chMtxUnlock(&can_mtx);
#else
    (void)id;
    (void)data;
    (void)len;
    (void)replace;
#endif
}
```

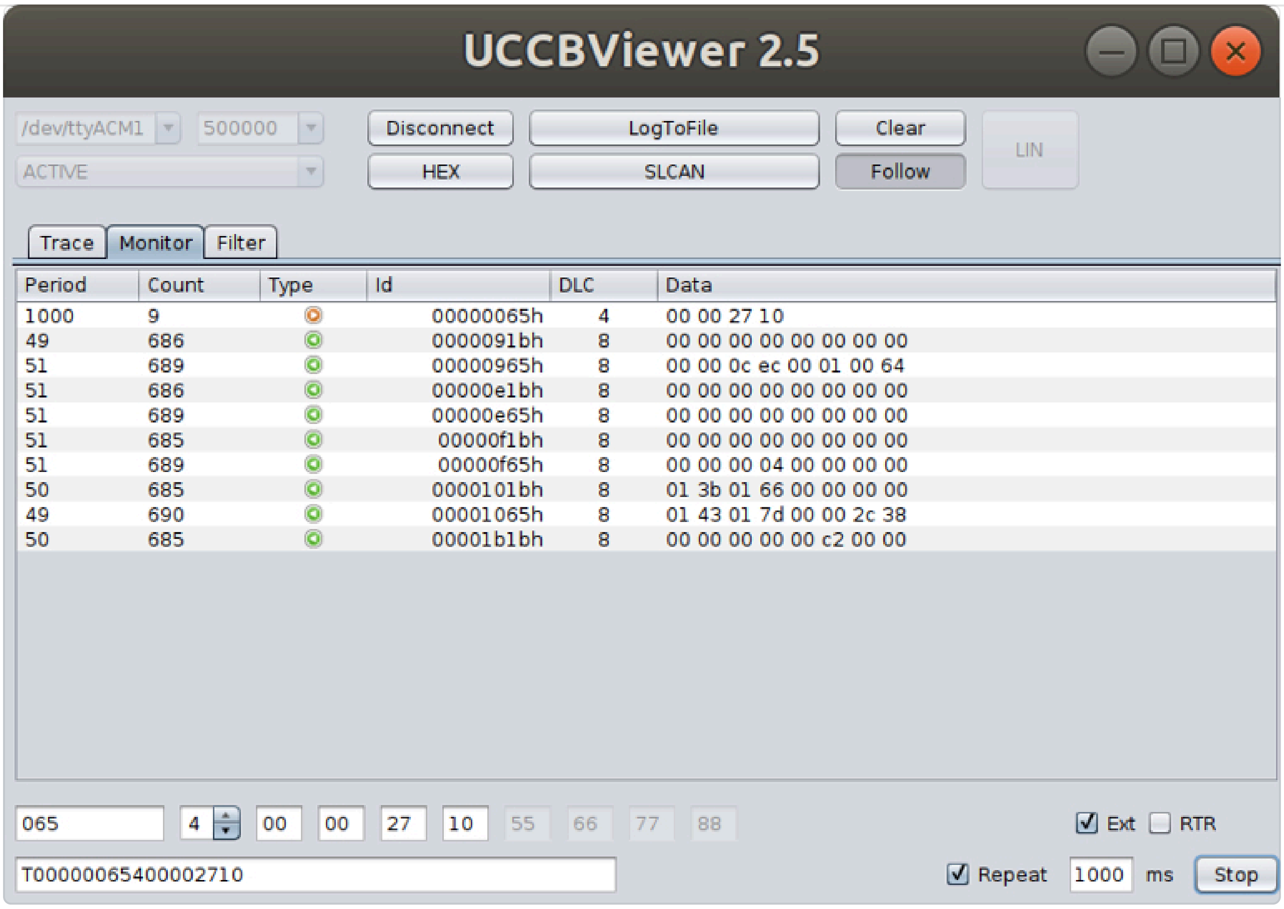
And, above function called as below. For example, to control the current, we send the VESC CAN ID (controller_id) and index number of CAN_PACKET_ID (CAN_PACKET_SET_CURRENT) are muxed and sent to the id value in comm_can_transmit_eid_replace() function. And we send 4 byte current data (current value * 1000) in the DATA frame.

```
void comm_can_set_current(uint8_t controller_id, float current) {
    int32_t send_index = 0;
    uint8_t buffer[4];
    buffer_append_int32(buffer, (int32_t)(current * 1000.0), &send_index);
    comm_can_transmit_eid_replace(controller_id |
            ((uint32_t)CAN_PACKET_SET_CURRENT << 8), buffer, send_index, t
}
```

The typical CAN message commands available in VESC are tabulated. If the controller_id is 101(0x65), we can use below commands.

Message Name	Extended ID	DLC (Data Length Code)	DATA
Set Duty Cycle (index=0)	0x00000065	4	duty * 100000.0
Set Current (index=1)	0x00000165	4	current * 1000.0
Set Current Brake (index=2)	0x00000265	4	current * 1000.0
Set RPM (index=3)	0x00000365	4	rpm
Set Position (index=4)	0x00000465	4	pos * 1000000.0

If we send 10% duty cycle command to the controller_id 101, the can message is as below. At DATA, 0.1*100000.0 = 10000 = 0x2710. When using UCCBViewer, you should turn on Ext and Repeat option.



10% Duty Cycle Command

If we send 5A current brake command to the controller_id 101, the can message is as below. At DATA, 5*1000.0 = 5000 = 0x1388

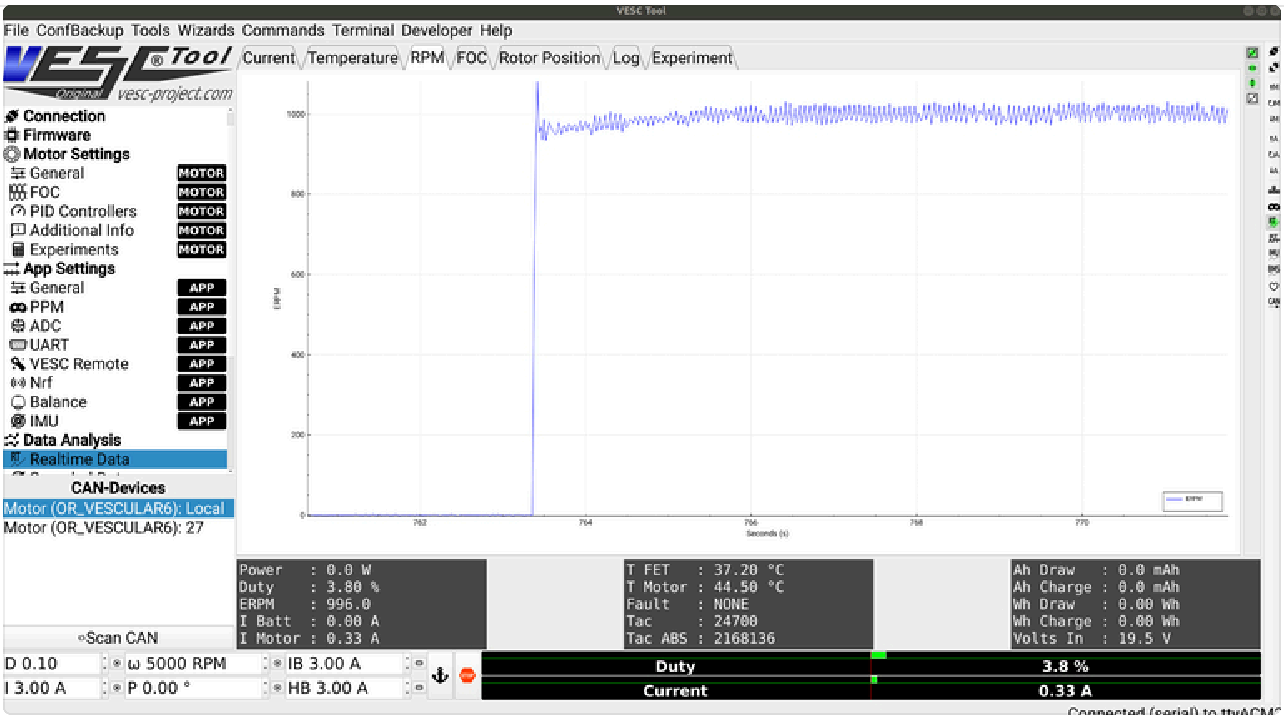
Id	DLC	Data
00000265h	4	00 00 13 88

5A Current Brake Command

If we send 1000 ERPM command to the controller_id 101, the can message is as below. At DATA, 1000 = 0x03E8

Id	DLC	Data
00000365h	4	00 00 03 e8

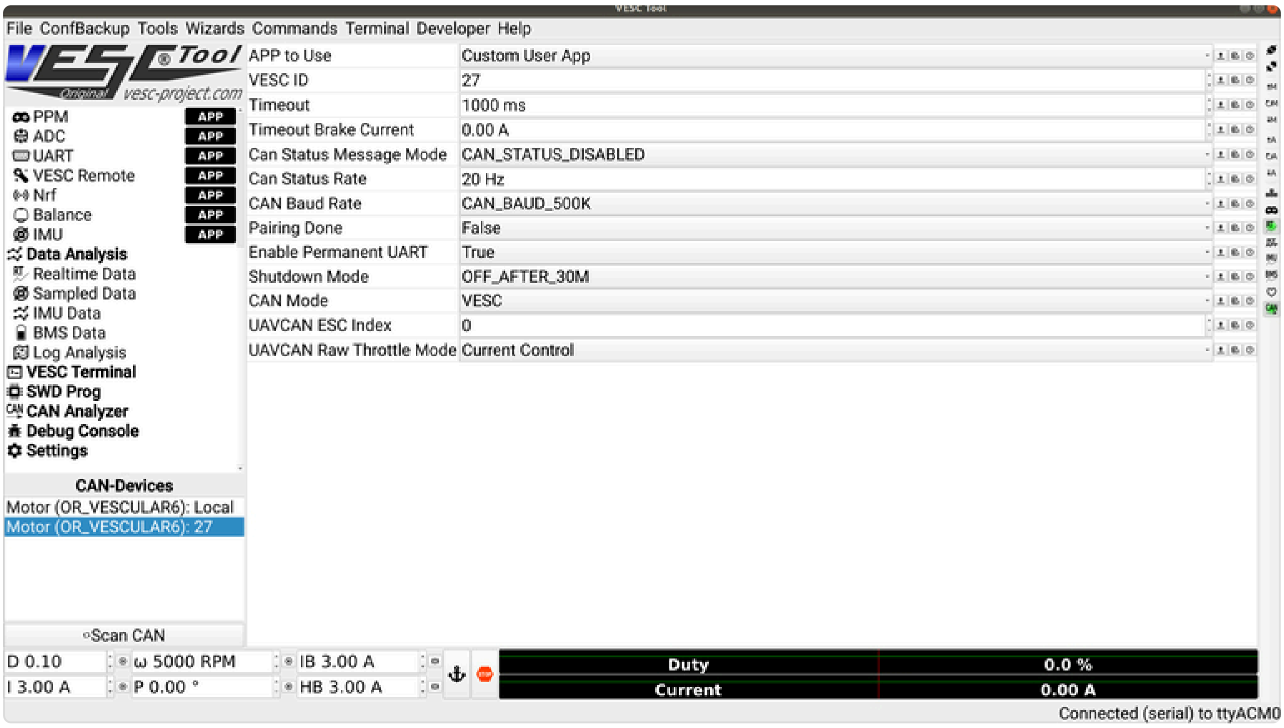
1000 ERPM (ERPM means Electrical RPM which is ERPM = RPM / number of pole)



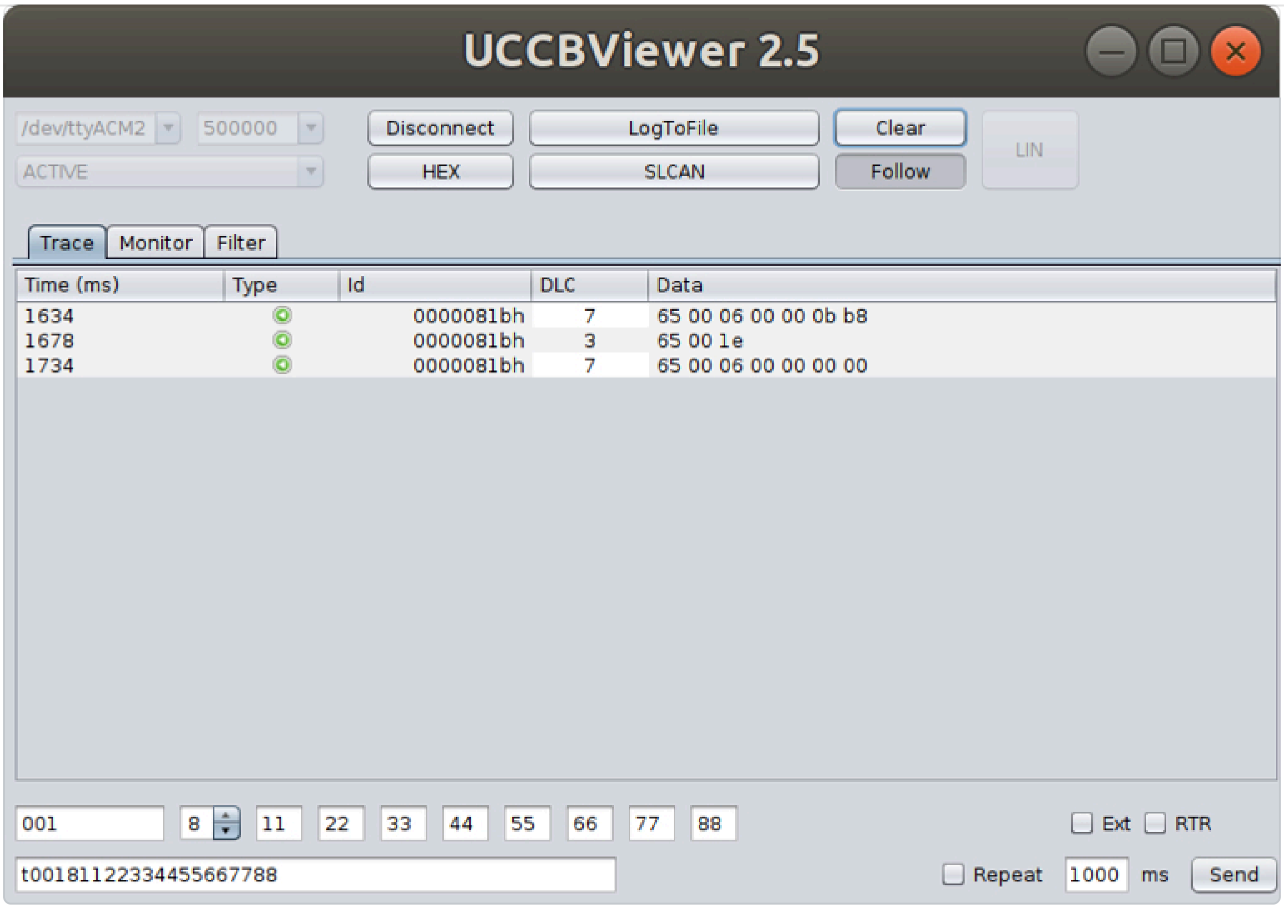
ERPM values observed in VESC-Tool

3. CAN Forward

In VESC-Tool, we can access all the functions of CAN connected device using CAN Forward function. Now, my VESC-Tool is connected to VESC (controller_id 101) by USB (ttyACM0). When we select Motor 27, it turns on CAN Forward function automatically, then we can find CAN Forward Messages as below.



VESC-Tool when using CAN Forward



Sending command current 3.0A using CAN Forward

CAN Status function is disabled on both controller. Above figure shows the trace of CAN messages when I command 3.0A current using CAN Forward.

Let's decode this;

- 0000081bh means,
  - 0x000008 = 8(CAN_PACKET_PROCESS_SHORT_BUFFER)
  - 0x1b = 27(controller_id)
- 0x65 00 06 00 00 0b b8 means,
  - 0x65 = 101(controller_id)
  - 0x00 ('send' value at comm_can_send_buffer() function)
  - 0x06 = 6(COMM_SET_CURRENT)
  - 0x00000bb8 = 3000(3000/1000.0 = 3A)

The 3.0A current command is given to VESC 101 from VESC-Tool via USB.The commands sent from VESC-Tool are probably as follows:

- 0x221b0600000bb8 which means
  - 0x22 = 34(COMM_FORWARD_CAN)

- 0x1b = 27(CAN Forward Target VESC controller_id)
- 0x06 = 6(COMM_SET_CURRENT)
- 0x00000bb8 = 3000(current *1000.0 so, it means 3000/1000.0 = 3A)

This messages is processed on 'command.c' using below code. In our case, HW_HAS_DUAL_MOTORS is not defined.

```
case COMM_FORWARD_CAN: {
        send_func_can_fwd = reply_func;

#ifdef HW_HAS_DUAL_MOTORS
        if (data[0] == utils_second_motor_id()) {
            mc_interface_select_motor_thread(2);
            commands_process_packet(data + 1, len - 1, reply_func);
            mc_interface_select_motor_thread(1);
        } else {
            comm_can_send_buffer(data[0], data + 1, len - 1, 0);
        }
#else
        comm_can_send_buffer(data[0], data + 1, len - 1, 0);
#endif
    } break;
```

That is, the data[0] is CAN Forward Target VESC controller_id(0x1b) and the rest of the data(0x0600000bb8) is passed by CAN bus. The actual code of comm_can_send_buffer() function is as below and the len of original data (0x221b0600000bb8) is 7 (7byte), so the len -1 = 6.

So, at the below code, actual values of input variables are as follows:

- uint8_t controller_id = 0x1b
- uint8_t *data = 0x0600000bb8
- unsigned int len = 6
- uint8_t send = 0

Because len <=6, send_buffer[] will be packed up as follows;

- send_buffer[0] = 101(0x65) -> local VESC controller_id
- send_buffer[1] = 0
- send_buffer[2] = 0x06
- send_buffer[3] = 0x00

- send_buffer[4] = 0x00
- send_buffer[5] = 0x0b
- send_buffer[6] = 0xb8

```
void comm_can_send_buffer(uint8_t controller_id, uint8_t *data, unsigned i
    uint8_t send_buffer[8];

    if (len <= 6) {
        uint32_t ind = 0;
        send_buffer[ind++] = app_get_configuration()->controller_id;
        send_buffer[ind++] = send;
        memcpy(send_buffer + ind, data, len);
        ind += len;
        comm_can_transmit_eid_replace(controller_id |
                ((uint32_t)CAN_PACKET_PROCESS_SHORT_BUFFER << 8), send_buf
    } else {
        unsigned int end_a = 0;
        for (unsigned int i = 0;i < len;i += 7) {
            if (i > 255) {
                break;
            }

            end_a = i + 7;

            uint8_t send_len = 7;
            send_buffer[0] = i;

            if ((i + 7) <= len) {
                memcpy(send_buffer + 1, data + i, send_len);
            } else {
                send_len = len - i;
                memcpy(send_buffer + 1, data + i, send_len);
            }

            comm_can_transmit_eid_replace(controller_id |
                    ((uint32_t)CAN_PACKET_FILL_RX_BUFFER << 8), send_buffe
        }

        for (unsigned int i = end_a;i < len;i += 6) {
            uint8_t send_len = 6;
            send_buffer[0] = i >> 8;
            send_buffer[1] = i & 0xFF;

            if ((i + 6) <= len) {
                memcpy(send_buffer + 2, data + i, send_len);
            } else {
                send_len = len - i;
                memcpy(send_buffer + 2, data + i, send_len);
            }

            comm_can_transmit_eid_replace(controller_id |
                    ((uint32_t)CAN_PACKET_FILL_RX_BUFFER_LONG << 8), send_
        }

        uint32_t ind = 0;
        send_buffer[ind++] = app_get_configuration()->controller_id;
```

```
            send_buffer[ind++] = send;
            send_buffer[ind++] = len >> 8;
            send_buffer[ind++] = len & 0xFF;
            unsigned short crc = crc16(data, len);
            send_buffer[ind++] = (uint8_t)(crc >> 8);
            send_buffer[ind++] = (uint8_t)(crc & 0xFF);

            comm_can_transmit_eid_replace(controller_id |
                    ((uint32_t)CAN_PACKET_PROCESS_RX_BUFFER << 8), send_buffer
        }
    }
```

When the CAN message is actually sending, the extended ID (eid) contains the command(CAN_PACKET_PROCESS_SHORT_BUFFER=8).

As a result, we can see that the CAN message equals the value observed by the UCCBViewer.

On the same principle, we can decode all the messages

Id	Data	Meaning
0x0000081b	65 00 1e	CAN_Forward to 0x1b(27), 0x1e = 30(COMM_ALIVE)
0x0000081b	65 00 06 00 00 00 00	CAN_Forward to 0x1b(27), 0x06 = 6(COMM_SET_CURRENT) 0x00000000 = 0(current *1000.0)

! Note that COMM_ALIVE is used to maintain the current state of control.

For short instructions with a length of less than 6, the CAN_FORWARD command is processed as shown above.

This time, let's talk about a longer command.

I sended CAN messages (Id = 0x0000081, DLC = 3, DATA = 0x 650004). This message means follows:

- 0000081bh means,
  - 0x000008 = 8(CAN_PACKET_PROCESS_SHORT_BUFFER)
  - 0x1b = 27(controller_id)

- 0x650004 means,
  - 0x65 = 101(controller_id)
  - 0x00 ('send' value at comm_can_send_buffer() function)
  - 0x04 = 4(COMM_GET_VALUES)

The above message is handled by VESC 27 and the code below is called because the command is 'CAN_PACKET_PROCESS_SHOT_BUFFER' from 'comm_can.c'.

```
case CAN_PACKET_PROCESS_SHORT_BUFFER:
            ind = 0;
            rx_buffer_last_id = data8[ind++];
            commands_send = data8[ind++];

            if (is_replaced) {
                if (data8[ind] == COMM_JUMP_TO_BOOTLOADER ||
                        data8[ind] == COMM_ERASE_NEW_APP ||
                        data8[ind] == COMM_WRITE_NEW_APP_DATA ||
                        data8[ind] == COMM_WRITE_NEW_APP_DATA_LZO ||
                        data8[ind] == COMM_ERASE_BOOTLOADER) {
                    break;
                }
            }

            switch (commands_send) {
            case 0:
                commands_process_packet(data8 + ind, len - ind, send_packe
                break;
            case 1:
                commands_send_packet_can_last(data8 + ind, len - ind);
                break;
            case 2:
                commands_process_packet(data8 + ind, len - ind, 0);
                break;
            default:
                break;
            }
            break;
```

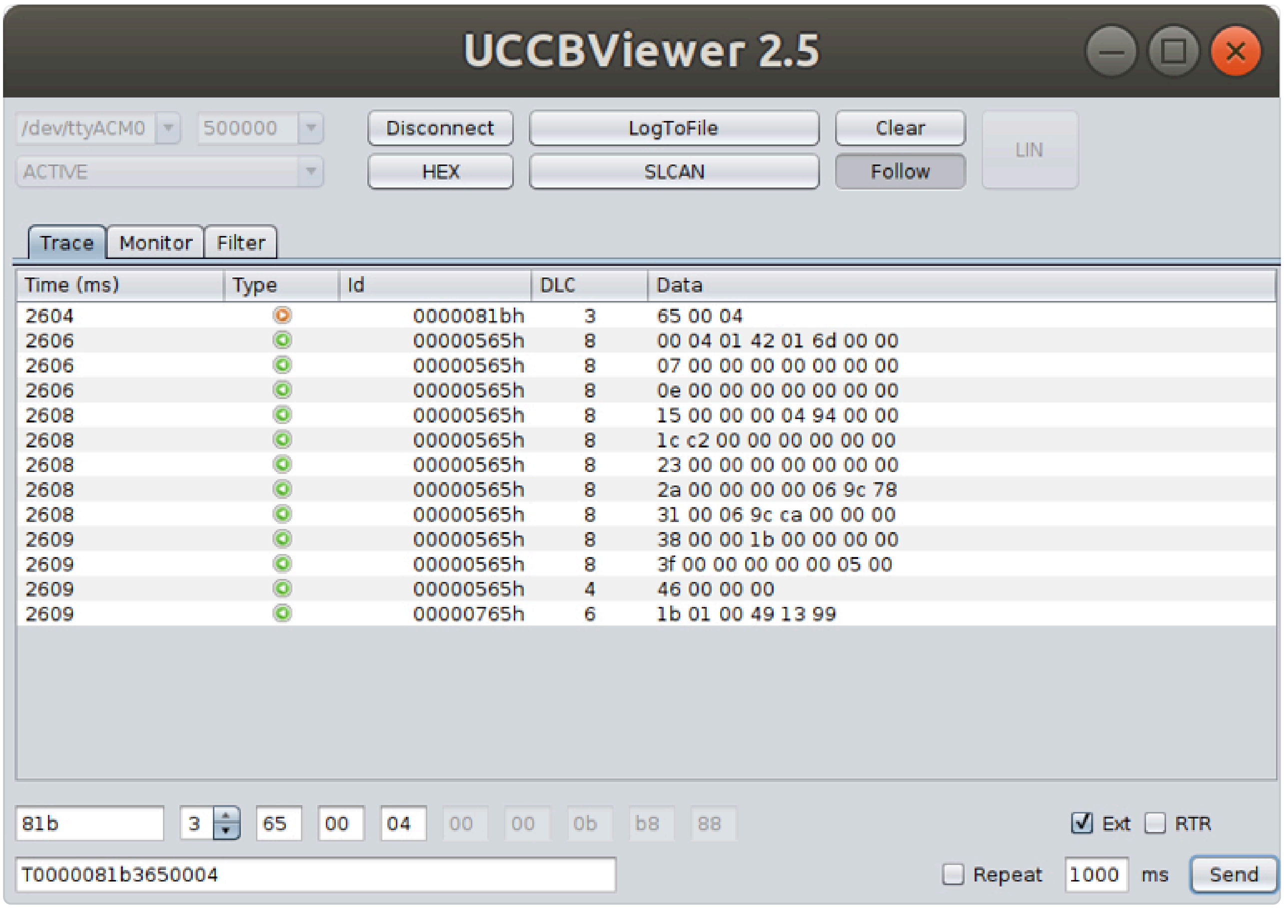
```
static void send_packet_wrapper(unsigned char *data, unsigned int len) {
    comm_can_send_buffer(rx_buffer_last_id, data, len, 1);
}
```

In here,

- rx_buffer_last_id = 0x65
- commands_send = 0

- commands_process_packet (0x04, 1, function pointer)

Third input of function 'commands_process_packet()' is function pointer of 'send_packet_wrapper()' and eventually 'comm_can_send_buffer()' is called to reply to messages. As a result, we can observe below replying messages about 'COMM_GET_VALUES'.



- 00000565h
    - 0x000005 = 5(CAN_PACKET_FILL_RX_BUFFER) -> receive data stream
    - 0x65 = 101
- 00000765h
    - 0x000007 = 7(CAN_PACKET_PROCESS_RX_BUFFER) -> process received data
    - 0x65 = 101

In the data of Id (00000565h), every first byte indicate data length Index, that is

- 0x00 = 0
- 0x07 = 7
- 0x0e = 14
- 0x15 = 21

- 0x1c = 28
- 0x23 = 35
- 0x2a = 42
- 0x31 = 49
- 0x38 = 56
- 0x3f = 63
- 0x46 = 70

Except for the first byte, the remaining data corresponds to continuous data in the reply message for COMM_GET_VALUE. Separating the data into bytes per variable shown in 'commands.c'. It can be interpreted as follows (| indicate bytes separation):

Id	Data	Meaning
00000565h	00	04	01 42	01 6d	00 00 07	00 00	00 00 00 00	00 0e	00 00 00	00 00 00 00 | 15	00 00	00 04 94 00	00 1c	c2	00 00 00 00	00 00 23	00 00	00 00 00 00	00 2a	00 00 00	00 06 9c 78 | 31	00 06 9c ca	00	00 00 38	00 00	1b | 00 00	00 00 | 3f	00 00	00 00 00 05	00 46	00 00 00	00 = 0(length index) 04 = 4(COMM_GET_VALUES) 01 42 = temp_fet*10 ->temp_fet = 32.2 01 6d = temp_motor*10 ->temp_motor = 36.5 00 00 00 00 = motor_current*100 00 00 00 00 = input_current*100 00 00 00 00 = id*100 00 00 00 00 = iq*100 00 00 = duty*1000 00 04 94 00 = rpm ->rpm=300032 00 c2 = volt*10 ->volt = 19.4 00 00 00 00 = amp_hours*10000 00 00 00 00 = amp_hours_charged*10000 00 00 00 00 = watt_hours_charged*10000 00 00 00 00 = watt_hours_charged*10000 00 06 9c 78 = tacho ->433272 00 06 9c ca = tacho_abs ->433354 00 = fault_code 00 00 00 00 = pid_pos_now*1000000 1b = 27 (controller_id) 00 00 = temp_mos1*10 00 00 = temp_mos2*10 00 00 = temp_mos3*10

		00 00 00 05 = vd*1000 ->vd=0.005 00 00 00 00 = vq*1000
00000765h	1b 01 00 49 13 99	1b = 27 (controller_id) 01 = send 00 49 = 73 = length of data 13 99 = CRC

Above, we introduce how to obtain data by requesting 'COMM_GET_VALUE' in a 'CAN_FORWARD' manner. The method of using "CAN_FORWARD" can be used to transmit virtually all the "commands" of the VESC, but this is somewhat complicated because it requires proper interpretation of the data for the reply message as seen earlier. However, I think it will be able to use it well if necessary.

Reference 1 : electric-skateboard.builders (VESC CAN Message Structure)

[VESC CAN Message Structure ESK8 Electronics vesccan-bus News Log In]()

<table>
<tr><th>Message Name</th><th>Message Extended ID</th><th>Byte 1</th><th>Byte 2</th><th>Byte 3</th><th>Byte 4</th><th>Byte 5</th><th>Byte 6</th><th>Byte 7</th><th>Byte 8</th></tr>
<tr><td>Set Duty Cycle</td><td>0x000000TA</td><td colspan="4">Duty Cycle * 100000</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Set Current</td><td>0x000001TA</td><td colspan="4">Current * 1000</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Set Current Brake</td><td>0x000002TA</td><td colspan="4">Brake Current * 1000</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Set RPM</td><td>0x000003TA</td><td colspan="4">RPM</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Set Pos(ition?)</td><td>0x000004TA</td><td colspan="4">Pos(ition?) * 1000000</td><td></td><td></td><td></td><td></td></tr>
<tr><td rowspan="8">Get Data Long Buff (when instruction 0x04 is sent with message 8)</td><td rowspan="8">0x000005TA (TA in this case is the node that asked for the data, which is byte 1 in message 8)</td><td>0x00</td><td>0x04</td><td colspan="2">mostemp1</td><td colspan="2">mostemp2</td><td colspan="2">mostemp3</td></tr>
<tr><td>0x07</td><td colspan="2">mostemp4</td><td colspan="2">mostemp5</td><td colspan="2">mostemp6</td><td>temppcb</td></tr>
<tr><td>0x0E</td><td>temp pcb</td><td colspan="4">avg motor current</td><td colspan="2">avg input current</td></tr>
<tr><td>0x15</td><td colspan="2">avg input current</td><td colspan="2">duty cycle</td><td colspan="3">rpm</td></tr>
<tr><td>0x1C</td><td>rpm</td><td colspan="2">voltage</td><td colspan="4">amp hours consumed</td></tr>
<tr><td>0x23</td><td colspan="4">amp hours charged</td><td colspan="3">watt hours consumed</td></tr>
<tr><td>0x2A</td><td>whc</td><td colspan="4">watt hours charged</td><td colspan="2">odometer</td></tr>
<tr><td>0x31</td><td colspan="2">odometer</td><td colspan="4">odometer abs</td><td>fault code</td></tr>
<tr><td>Process Buffer (acts as flag for 5)</td><td>0x000007TA</td><td>SA</td><td>0x01</td><td colspan="2">total buffer length</td><td colspan="2">crc</td><td></td><td></td></tr>
<tr><td>Ask For Data</td><td>0x000008TA</td><td>SA</td><td>0x00</td><td>instruction</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Status Broadcast</td><td>0x000009SA</td><td colspan="4">rpm</td><td colspan="2">current</td><td colspan="2">duty cycle</td></tr>
</table>

https://esk8content.nyc3.digitaloceanspaces.com/uploads/db2454/original/3X/b/4/b4d3b287266b1668fef1b6d91a67a7849bd0dd56.png

Reference 2

https://www.vesc-project.com/sites/default/files/imce/u15301/VESC6_CAN_CommandsTelemetry.pdf

271KB

VESC6_CAN_CommandsTelemetry.pdf
pdf

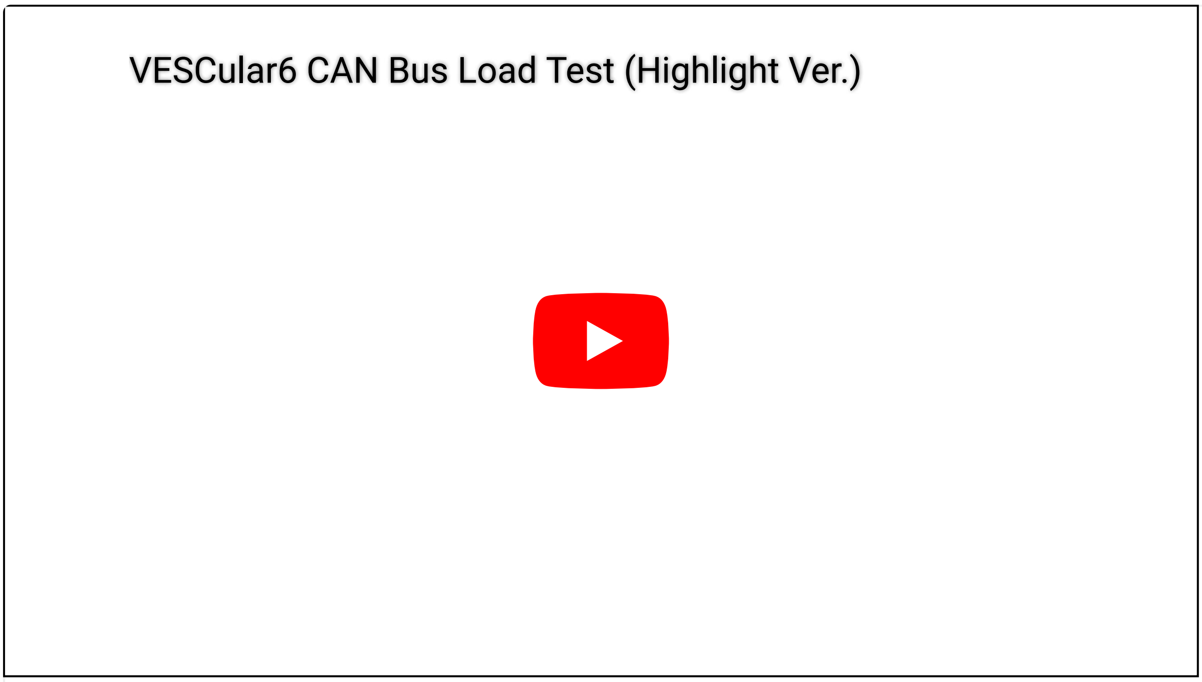
4. CAN Bus Load Test

Condition:

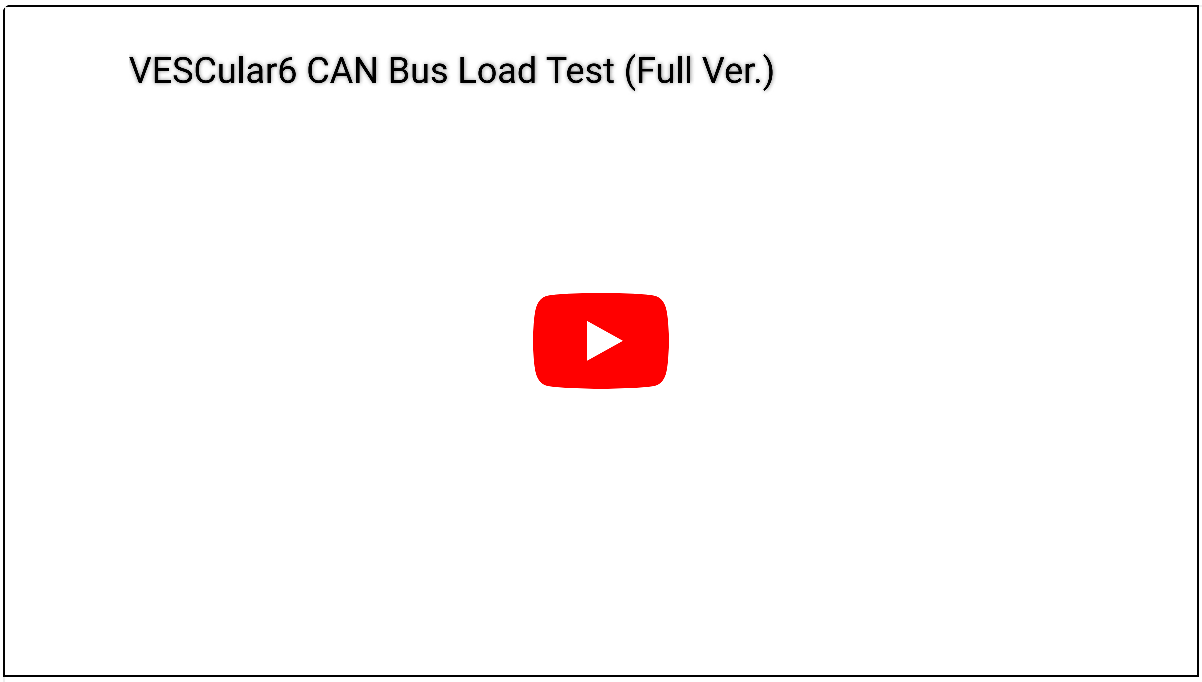
- CAN Bus Baudrate : 1Mbps
- Command Freq : 1kHz
- Status Message Freq : 1kHz
- Status Message Data : pos(rad), vel(rad/s), curr(A), motor_temp(degree)

Result :

- Max CAN bus Load : 85%
- Max Controller : 6EA



30sec Short Version



Full Version

Previous
Control with Python

Next
VESCular6

Last updated 2 years ago