



### OpenValveControl

**Imot:** 25.0 mA    **Temp:** 19.6 °C

### Valve Positions

<input checked="" type="radio"/> VZ1	<div></div>	49 %
<input type="radio"/> VZ2	<div></div>	0 %
<input type="radio"/> VZ3	<div></div>	65 %
<input type="radio"/> VZ4	<div></div>	30 %

Set Position [%]

Imot max. [mA]

Move VZ

Home VZ

Info

WLAN-SSID

WLAN-Password

Save & Reboot

/move?vz=1&set\_pos=70&max\_mA=30.0

OpenValveControl Web-UI v0.3 from Klaus Deutschkämmer

## Content

1	Introduction.....	3
1.1	Features.....	3
2	Initial Setup.....	4
2.1	Flashing the "ESP8266 D1-mini" for the first time.....	4
2.1.1	Arduino IDE (2.2.1).....	4
2.1.2	tasmotizer.....	4
2.1.3	Via command line using "esptool.py".....	5
2.2	WiFi Configuration.....	5
2.3	Reboot and Login to your home network.....	7
2.4	Additional settings (ovc.ini).....	8
2.5	Optional JavaScript file (customize.js).....	9
2.6	First driving trials.....	11
2.6.1	Delivery status of the drives.....	11
2.6.2	Checking the reference run.....	11
3	Controlling with node-RED.....	13
3.1	Adjusting the valves using node-RED.....	13
3.1.1	Task definition.....	13
3.1.2	Specifying a setpoint.....	14
3.1.3	flow: "Subscribe" thermostat values from our MQTT broker.....	14
3.1.4	flow: Convert temperature difference into valve position.....	15
3.1.5	flow: Output the query parameters as http GET request.....	17
4	Driving the motors by PWM.....	19
4.1	Introduction to the H-bridge.....	19
4.2	PWM generation and interrupts.....	20
4.3	Acquisition of motor current.....	21
4.3.1	Exemplary measurement results.....	22
4.4	Measuring the motor inductance.....	23
4.5	Measuring the motor speed.....	25
4.6	Measuring the back EMF.....	26
4.6.1	Comparing Simulation and Measurement.....	28
4.6.2	Back EMF sample oscillograms.....	30
4.7	Calculating the motor constant $k_e$ .....	31
5	Motor simulation with LTspice XVII.....	32
5.1	LTspice Circuit diagram.....	32
6	Data Logger.....	33

## 1 Introduction

Open Valve Control is a low cost smart control for motorized 3 V floor heating valve actuators (like HmIP VdMOT and others) with web interface. The web interface (ESP-ValveControl) uses an ESP8266 D1-mini and the motor control (PIC-ValveControl) uses a PIC18F16Q41. All components are low cost and pluggable.

### 1.1 Features

- Connect up to 4 motorized, 3 V floor heating valve actuators (like HmIP VdMOT and similars)
- OLED Display (visualization of valve positions, motor current, temperature and status)
- WiFi-Interface (visualization, command execution and status requests using GET, POST, JSON)
- Supports DS18B20 (heating flow temperature)
- Filesystem support on ESP
- (upload web-UI "/index.html", setting data "/ovc.ini" and custom labels "/customize.js")
- Access point (until local WiFi has been configured)
- OTA-Support for ESP (Over-the-Air Update)
- OTA-Support (Over-the-Air Bootloader) for PIC- $\mu$ C
- MQTT Support
- Data logger for motor current and speed ( $V_{BEMF}$ )
- Examples for NodeRed, MQTT (, FHEM, Raspberry Pi, ...) - see Documentation.

While the source code and doxygen documentation is available on github, this documentation gives some additional insights in the project, especially it gives some hints for installation and setup. But also the basics of running a motor using PWM and measuring current and back EMF voltage is provided.

This document will be continued as time permits. Feel free to contact me, if you miss certain topics. Thank You.

## 2 Initial Setup

### Prerequisites

- The board is assembled with all the required components and
- The PIC controller has been flashed (e.g. with a PICkit4) with the latest firmware.

### 2.1 Flashing the "ESP8266 D1-mini" for the first time

There are many ways to flash the ESP for the first time. They all involve connecting the board via the USB interface. The appropriate driver for the USB interface IC installed on the D1 mini may need to be installed first.

Under Linux, the command 'lsusb' can be used to check this. The name of the USB port can be found in the /dev/ directory:

```
~$ lsusb
Bus 002 Device 006: ID 1a86:7523 QinHeng Electronics CH340 serial converter
...

~$ ls -l /dev/ttyUSB*
crw-rw---- 1 root dialout 188, 0 Dez 15 14:00 /dev/ttyUSB0
~$
```

#### 2.1.1 Arduino IDE (2.2.1)

Board: LOLIN(WEMOS) D1 mini

FlashSize: 4 MB (FS: 1MB, OTA: ~1019KB)

#### 2.1.2 tasmotizer

"Tasmotizer" is a Python application and must therefore be started via Python (currently python3). The tool *tasmotizer.py* can be downloaded from github, the path in the following screenshot must be adapted to the download location:

```
~$ python3 .local/bin/tasmotizer.py
esptool.py v2.8
Serial port /dev/ttyUSB0
Connecting...
Chip is ESP8266EX
Features: WiFi
Crystal is 26MHz
Uploading stub...
Running stub...
Stub running...
Configuring flash size...
Auto-detected Flash size: 4MB
Erasing flash (this may take a while)...
Chip erase completed successfully in 3.1s
Compressed 383456 bytes to 276588...
Wrote 383456 bytes (276588 compressed) at
0x00000000 in 24.4 seconds (effective 126.0
kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...
```

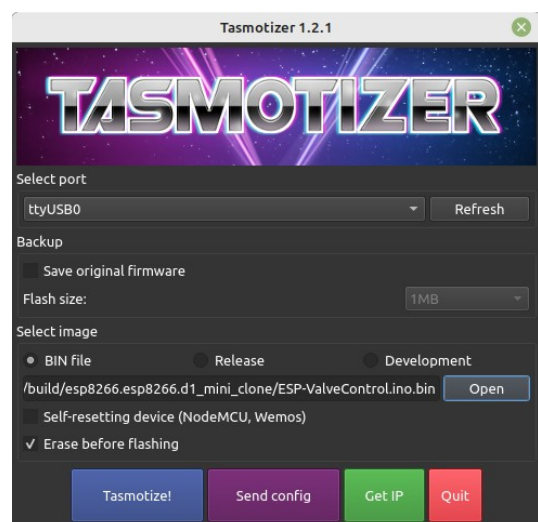


Figure 1: User interface of „Tasmotizer“ (see <https://github.com/tasmota/>)

## User Manual (DRAFT)

---

In the graphical user interface, simply select the USB port and the binary program (ESP-ValveControl.ino.bin) with "Open" (navigate to the storage location) and press "Tasmotize!". The rest happens automatically.

### 2.1.3 Via command line using "esptool.py"

The Python tool "esptool.py" is usually part of the Arduino IDE, but can also be downloaded and installed from "Espressiv" if Phyton is not already installed on the computer. The path information for esptool.py and the ESP-ValveControl binary file must of course be adapted again:

```
~$ python3 ~/.arduino15/packages/esp8266/hardware/esp8266/3.1.2/tools/esptool/esptool.py -p /dev/ttyUSB0
write_flash 0x000000 ~/.git/OpenValveControl/ESP-ValveControl/build/esp8266.esp8266.dl_mini_clone/
ESP-ValveControl.ino.bin
esptool.py v3.0
Serial port /dev/ttyUSB0
Connecting....
Detecting chip type... ESP8266
Chip is ESP8266EX
Features: WiFi
Crystal is 26MHz
MAC: 24:a1:60:3a:bb:83
Uploading stub...
Running stub...
Stub running...
Configuring flash size...
Compressed 383456 bytes to 276588...
Wrote 383456 bytes (276588 compressed) at 0x00000000 in 24.4 seconds (effective 125.9
kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...
~$
```

Figure 2: Using esptool in linux console

## 2.2 WiFi Configuration

If the flashing was successful, the following message<sup>1</sup> should appear in the first two lines of the **OLED display** (as an indication that the ESP is working as an access point):

```
OVC-Access-Point
OVC-password
```

Shortly afterwards, the message changes and the access point's IP address is displayed:

```
IP: 192.168.4.1
21.1 C          0.0 mA
```

---

1 If the OLED display does not work, the hardware and especially the connection (and polarity) of the OLED must be checked.

## User Manual (DRAFT)

Now you can connect via WiFi to this access point "OVC-Access-Point" and the password "OVC-password" (case sensitive!) and enter the displayed IP address in a browser.

The following message should appear in your browser:



Figure 3: Appearance in browser prior upload of filesystem user interface

The file "fs.html" must be selected from the ESP directory with "Browse" and then copied to the ESP file system by clicking the button "Upload".

The page then updates itself and shows the ESP8266 file manager with the content "fs.html", the file that has just been uploaded:

### ESP8266 Filesystem Manager

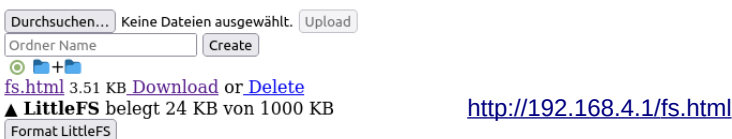


Figure 4: Appearance in browser after upload of basic filesystem user interface (fs.html)

In the same way, we then upload the files

"style.css" (css formatting for fs.html) and

"index.html" (the ESP Valve Control user interface)

to the ESP file system:

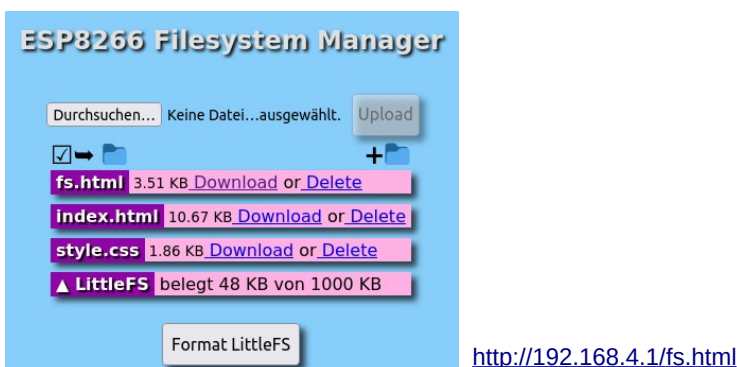
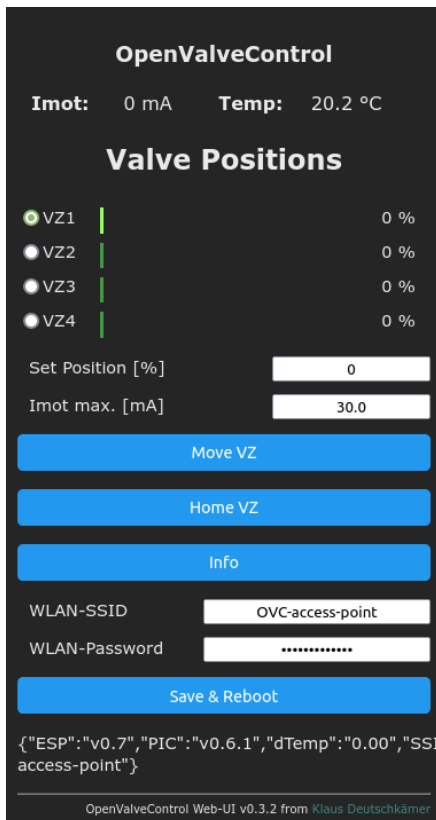


Figure 5: Appearance in browser after upload of fs.html, stylesheet (style.css) and web-UI (index.html)

After entering the top URL (in our example: <http://192.168.4.1>) into your browser will display the graphical user interface of ESP-ValveControl (it is loaded from the ESP file system):



<http://192.168.4.1> (IP of access point)

- ← Enter here the WiFi SSID of your home network,
- ← and the WiFi password.
- ← Press this button to save the settings.

Figure 6: OpenValveControl Graphical User Interface in Web-Browser

## 2.3 Reboot and Login to your home network

After the reboot, the ESP attempts to log into the home network with the data entered. If this is not successful within about 2 minutes, the access point is activated again.

**i** If problems occur, please check whether your router is configured so that only known devices are allowed to connect. Change this setting temporarily if necessary.

When restarting, the SSID you entered for your home network is displayed in the top line of the OLED. This line flashes while the connection has not yet been established.

The version of the ESP8266 software is displayed in the second line:

Example:

```
FRITZ!Box 6390 Cable
ESP v0.7
```

After successful login, the first row of the OLED now shows the IP address as assigned by your router.

User Manual (DRAFT)

---

The second line shows the actual temperature<sup>2</sup> and motor current.

Example:

```
IP: 192.168.2.131
19.5 C          0.0 mA
```

You can now connect to the home network and enter the IP address displayed. The ESP-ValveControl graphical user interface should then be displayed again, but now with the SSID of your home network.

## 2.4 Additional settings (ovc.ini)

After entering and saving the WiFi parameters, an "ovc.ini" file is created in which the settings are saved permanently. Even after a firmware update, the data is normally still available unless you specify that the flash is to be completely erased (e.g. with Tasmotizer: "Erase before flashing").

However, a loss is not a problem, the installation described can be repeated at any time.

An example file **ovc.ini** is available in github in the folder "/ESP-ValveControl/".

```
## ovc.ini
# As default (or when it cannot connect within 2 minutes to the WiFi using the current
# settings), the ESP creates an access point with the credentials shown below.
# Please change the SSID and PSK to meet the settings of your WiFi, then upload the file into
# the ESP's file system, which can be accessed in your browser with the URI <local IP>/fs.html

# WLAN credentials
#SSID = Your SSID
#PSK = Your WiFi password
SSID = OVC-access-point
PSK = OVC-password

# MQTT: server (IP) and token to publish data every period (seconds) as "<mqtt_prefix>/status"
MQTT_HOST   = 192.168.2.72
MQTT_PREFIX = OVC-1
MQTT_PERIOD = 900

# ValveZone aliases (not used yet)
VZ1 = ESS
VZ2 = KÜ
VZ3 = BAD
VZ4 = WZ1

# Adjust temperature sensor
dTemp = -0.5
```

Figure 7: Example file **ovc.ini**

---

2 If no DS18B20 is connected, -128.2 C is displayed.



User Manual (DRAFT)

---

It means:

SSID	The name of your home network
PSK	Password of your home network
MQTT_HOST	IP-Adresse of your MQTT server.
MQTT_PREFIX	Prefix for the sent data (to differentiate between several identical devices (all data is sent with the MQTT token "<mqtt_prefix>/status").
MQTT_PERIOD	Time interval in seconds for sending the MQTT data ("publishing").
VZ1	Alternative identifiers for the valve zones (not yet used)
VZ2	
VZ3	
VZ4	
dTemp	Adjustment value for the DS18B20 temperature sensor

As a further alternative, you can also download the „ovc.ini“ file to your computer and save it or change it with a text editor and add further information (e.g. for MQTT).

You can also duplicate the settings on several devices or skip entering the WiFi settings in the access point and upload a prepared „ovc.ini“ file using the Filesystem Manager. After the reboot, all settings are then immediately active.

## 2.5 Optional JavaScript file (customize.js)

Starting from WebUI v0.4 you can upload a JavaScript file ‚customize.js‘ to the ESP file system. This script can replace some default headers and labels by your own texts, e.g.

- Change title (e.g. 1st Floor, to differentiate between several devices)
- Replace valve identifiers (VZ1, VZ2,...) with more meaningful names.

The script is read at the end of <head> section in the WebUI's „index.html“, but the function **CustomizeLabels()** is called once at the end of <body>, when the complete HTML document has been read by the browser.

The script also can be extended by your own instructions.

User Manual (DRAFT)

---

An example file **customize.js** is available in github in the folder `"/ESP-ValveControl/"`.

```
/* customize.js
   Upload this script into the ESP filesystem.
   It replaces some default headers and labels by the given texts.
*/
function CustomizeLabels ()
{
  document.getElementById('header1').innerHTML = '1<sup>st</sup> Floor'; // -> OpenValveControl
  document.getElementById('header2').innerHTML = 'Valve Positions';       // -> Valve Positions

  document.getElementById('radio1').labels[0].innerHTML = '1-Living';    // -> VZ1
  document.getElementById('radio2').labels[0].innerHTML = '2-Dilbert';   // -> VZ2
  document.getElementById('radio3').labels[0].innerHTML = 'VZ3';
  document.getElementById('radio4').labels[0].innerHTML = 'VZ4';
}
```

Figure 8: Example file **customize.js**

As an example, you will find the first replacement item in the `<body>` section of `index.html`:

```
<h2 id='header1'>OpenValveControl</h2>
```

The header text „OpenValveControl“ will be replaced with the given text in the script, because the `<h2>` expression was given the attribut `id = 'header1'`:

```
<h2 id='header1'>1<sup>st</sup> Floor</h2>
```

As you see, you can even enter html attributes like superscript `<sup>` (superscript) to get „1<sup>st</sup> Floor“ or similar HTML expressions.

Hence, basically this script overwrites the default headers to differentiate several devices and to give the valve radio buttons more meaningful (room) names.

These texts are preserved even if the `index.html` file is updated.

## 2.6 First driving trials

**i** A connected motor without a valve can move beyond its end position if it does not experience any resistance when moving. This may result in damage to the mechanics. The control unit is operated at your own risk.

### 2.6.1 Delivery status of the drives

The following photo shows an actuator in its as-delivered condition. You can see that the round piston (in the center) is offset a few millimeters inwards. This corresponds to the OPEN valve position on my Rotex heating circuit manifold and makes it easier to install the motor.

When referencing (homing), the actuator moves the piston out, so when installed it pushes the valve pin in, which corresponds to the CLOSED valve position (on my Rotex heating circuit manifold). When the valve is closed, the piston presses against the spring force of the valve pin, which is why installing the motor in motor position CLOSED is not recommended.

**i** This may be different with other devices!  
OpenValveControl does not currently allow the direction of rotation to be changed!



Figure 9: As-delivered condition

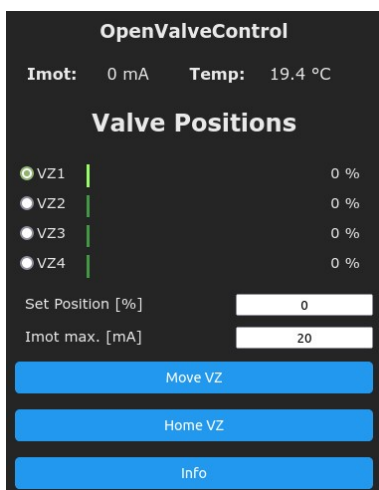
### 2.6.2 Checking the reference run

First of all, you should carry out a reference run - WITHOUT installing the motor in the heating circuit manifold. The easiest way to do this is on the desk. Connect the motor to the left-hand socket, for example, as shown in the next picture. This corresponds to valve zone VZ1 in the user interface.



Figure 10: Connection of a motor as VZ1 to test the reference run.

- Now select „VZ1“ in the user interface (by clicking with the mouse)
- Then enter the (small) value 20 for „Imot max [mA]“
- Now click on the "HOME VZ" button.



The motor should then move the piston "out" and display the motor current during this process. The LED on the PCB also lights up as an indicator.

The reference run is terminated when the motor current exceeds the entered limit value. Valve position "CLOSED" is then assumed and the actual position is set to 0 %.

You can easily simulate this with such a low limit value by pressing against the piston with a suitable tool or with your thumb while it is being moved out.

Figure 11: Graphical User Interface

**i** Caution! The piston must not move out so far that the guide lug comes completely out of the groove, otherwise the piston cannot be moved in without manual support.

The reference run is therefore aborted after approx. 2 minutes, even if the current limit value is not exceeded. The LED goes out.

In case of problems, simply unplug the drive!

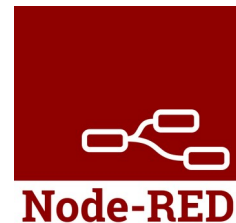
If this "reference run" has worked, you can specify a „Set Position“ of e.g. 20 % for the motor and move to this position using the "Move VZ" button.

If the motor stops prematurely, you can increase the current limit value in small steps. Proceed in a similar way once you have installed the motor. For comparison: I have entered a current limit value of 55 mA for my Rotex heating manifold.

### 3 Controlling with node-RED<sup>3</sup>

Node-RED is the "Swiss army knife" of the Internet of Things (IoT) and the ideal tool when it comes to exchanging data between sensors and actuators. In this chapter, we want to control *OpenValveControl* with a node-RED „flow“.

In node-RED, "flows" refer to the graphical representations and data connections of "nodes". A node can, for example, represent an interface to MQTT and is represented as a block (node) with inputs and outputs. The properties of these nodes are defined via parameters (e.g. the IP address of the MQTT server).



You can install node-RED on a RaspberryPi, for example, so that it is permanently available (24/7). On the same RaspberryPi, a mosquitto server can run in parallel as an MQTT host and other applications (e.g. grafana).

I have been running this configuration on a 3rd generation Pi for years without any problems.

#### 3.1 Adjusting the valves using node-RED

##### 3.1.1 Task definition

As we have seen, we can adjust our valves via the web interface. To do this, we need to know the IP address of the web UI.

Then we can select the ValveZone (the actuator or valve), enter the target position (0 to 100%), enter the maximum motor current in milliamperes and press the "Move" button.

This sends an "http GET" command to the ESP, which contains the parameters mentioned. The command is displayed in plain text at the bottom of the Web UI screen.

##### Example

Valve 1 (vz=1) should be moved to 26%. If the motor current exceeds the value 30 mA, the process should be aborted (blocking protection):

```
/move?vz=1&set_pos=26&max_mA=30.0
```

We can achieve exactly the same thing by sending the GET command via a web browser on our LAN. We just need to prefix it with the IP address of our control device (the IP address will be displayed on the OLED after the ESP has successfully logged into the home network):

```
http://192.168.2.131/move?vz=1&set_pos=26&max_mA=30.0
```

---

3 **node-RED** is a flow-based programming tool, originally developed by IBM's Emerging Technology Services team (<https://emerging-technology.co.uk/>) and now a part of the OpenJS Foundation (<https://openjsf.org/>).

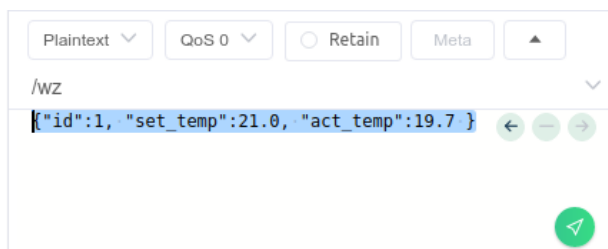
### 3.1.2 Specifying a setpoint

In order to solve the task in a practical way, we first have to specify the target value.

We assume that we have a temperature sensor with display in our living room for a reasonable individual room control as well as a possibility for a setpoint specification. This could be realized with a small ESP or with a commercially available industrial product (display of actual and setpoint temperature, 2 buttons ↕ for the setpoint).

This ESP could now send the described GET command directly to our controller via WiFi and, for example, control the valve to 80% if it is too cold in the room or to 20% if it is too warm (simple "two-point" control). In addition, each thermostat must of course be informed of the valve assignment, the maximum current and possibly the control algorithm. However, there is other information that is useful for control, e.g. outdoor temperature, solar radiation, night setback, extended absence (vacation) and others. It therefore makes sense to only report the actual and setpoint temperatures of the respective room and to manage all rooms centrally, e.g. with node-RED.

To do this, we send a sensor ID, the actual temperature and setpoint value to an MQTT server (also known as a broker, e.g. mosquitto). Until we have programmed the sensor, we can simulate this with the MQTTX<sup>4</sup> tool:



If we are connected to the MQTT broker (e.g. as test@raspi3:1880), we can define a **topic** (e.g. "/wz") and send the entered **payload** to the MQTT broker by clicking on the **green arrow**. The data can be changed as required.

Figure 12: Using MQTTX to simulate data source

### 3.1.3 flow: "Subscribe" thermostat values from our MQTT broker

In node-RED, we now place an "mqtt in" node in our "flow". Double-clicking on the node opens the "Properties" window, in which we enter the following data:

I have entered "localhost:1883" as the **server** because both node-RED and mosquitto (the MQTT broker) run on the same RaspberryPi. Otherwise, the corresponding IP address of the MQTT server must be entered instead of "localhost".

---

<sup>4</sup> „Your All-in-One MQTT Client Toolbox“ (<https://mqttx.app/>). MQTTX is a free open-source project available under the [Apache-2.0 LICENSE](#).

## User Manual (DRAFT)

We can assign the **topic** `/wz` (deutsch: WohnZimmer = Living Room) in the thermostat (or temperature sensor). We have entered this in MQTTX as a test and it is used to differentiate between the various rooms or temperature sensors.

Quality of Service "**QoS**" describes the increasing degree of reliability in the delivery of messages, the highest value is 2.

As **output**, we want a "parsed JSON object", i.e. the same message that was sent from the sensor to the MQTT broker.

The **name** can be chosen at will and is primarily used to make the flow clearer.

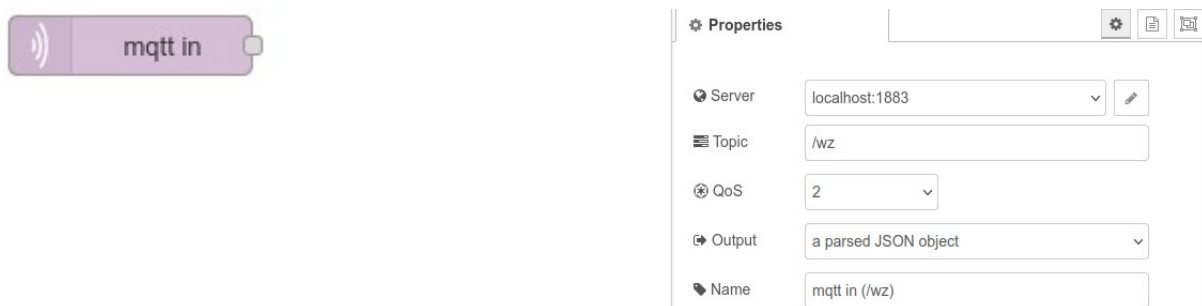


Figure 13: Configuring sample Node-Red „mqtt in“ node

We now add a "debug" node so that we can check the output of the "mqtt in" node. As soon as we send a data record to the MQTT broker via MQTTX, the subscribed data is sent to node-RED and is available at the output of the "mqtt in" node:

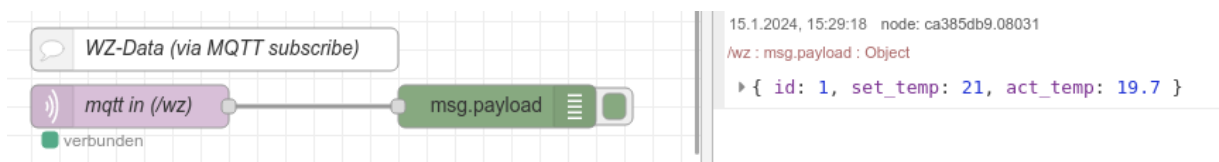


Figure 14: Using Node-Red „debug“ node

### 3.1.4 flow: Convert temperature difference into valve position

We now know the room ID, the setpoint and actual temperature and can now make a decision for the valve position.

To simplify matters, we take the room ID as the valve zone (vz) and program the two-point controller described above, which we want to formulate in JavaScript as follows:

If the deviation is more than 0.5 degrees, we switch the valve closed (20%) or open (80%), otherwise we leave it as it is.

Note: The simplified assumption is that the flow temperature is weather-compensated and we only make a fine adjustment with a valve. Of course, we can also program an arbitrarily complicated control algorithm here!

## User Manual (DRAFT)

For this purpose, node-RED has the "function" node, in which JavaScript can be executed almost arbitrarily. A "message" is accepted as input, which can be modified (almost) at will by the program and output at one or more outputs. The instruction `return (x)` is used for this purpose, whereby `x` can also be `null`, in which case no output is generated. We use this hysteresis to avoid unnecessary adjustments if the temperature is reasonably correct.

At the start of the JavaScript program, our message has the following values, for example:

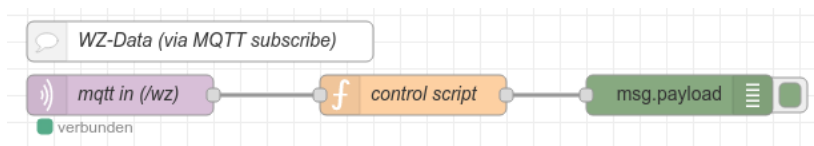
```
msg.payload = {"id":1,"set_temp":21,"act_temp":19.7}
```

At the output, we want to have the appropriate parameters for the "http GET" request:

```
msg2.payload = {"id":1, "set_pos":80, "max_mA":50.0}
```

All we have to do is generate a new message `msg2`. We temporarily take the missing value for "max\_mA" from a variable that was somehow saved in the flow with the instruction `flow.set("max_mA", <value>)`, for example.

Our JavaScript program could then look like this:



function Node bearbeiten

Löschen

Abbrechen

Fertig

Properties

Name

control script

Funktion

```
1 msg2 = {};  
2  
3 max_mA = flow.get('max_mA') || 49.5;  
4 vz = msg.payload.id;  
5 set_temp = msg.payload.set_temp;  
6 act_temp = msg.payload.act_temp;  
7  
8 if ((set_temp - act_temp) > 0.5)  
9 { // too cold:  
10   set_pos = 80;  
11 }  
12 else if ((act_temp - set_temp) > 0.5)  
13 { // too warm  
14   set_pos = 20;  
15 }  
16 else  
17 { // acceptable  
18   return(null);  
19 }  
20 msg2.payload = { "vz":vz,  
21                  "set_pos":set_pos,  
22                  "max_mA":max_mA  
23                };  
24 return msg2;
```

Debugging

15.1.2024, 17:25:52 node: ca385db9.08031  
msg.payload : Object  
{ vz: 2, set\_pos: 20, max\_mA: 49.5 }

15.1.2024, 17:26:03 node: ca385db9.08031  
msg.payload : Object  
{ vz: 2, set\_pos: 80, max\_mA: 49.5 }

Figure 15:



## User Manual (DRAFT)

The 2 outputs in the debugger were the responses to the following 3 messages:

```
{"id":2, "set_temp":18.0, "act_temp":19.7 }      // too warm
{"id":2, "set_temp":22.0, "act_temp":19.7 }      // too cold
{"id":2, "set_temp":20.0, "act_temp":19.7 }      // acceptable
```

This is a very good way of testing that no output is produced on the third input as intended.

#### Note

To prevent an error message from occurring when reading the max\_mA variable or an invalid value being used if no value has (yet) been saved, a logical AND has been appended, which assigns a default value in this case:

```
max_mA = flow.get('max_mA') || 49.5;
```

### 3.1.5 flow: Output the query parameters as http GET request

There is another (network) node for sending the "http request". We place it at the output of the JavaScript "function" node and connect both nodes. We move the debug node to the output of the http request. Then we fill in the properties as follows:

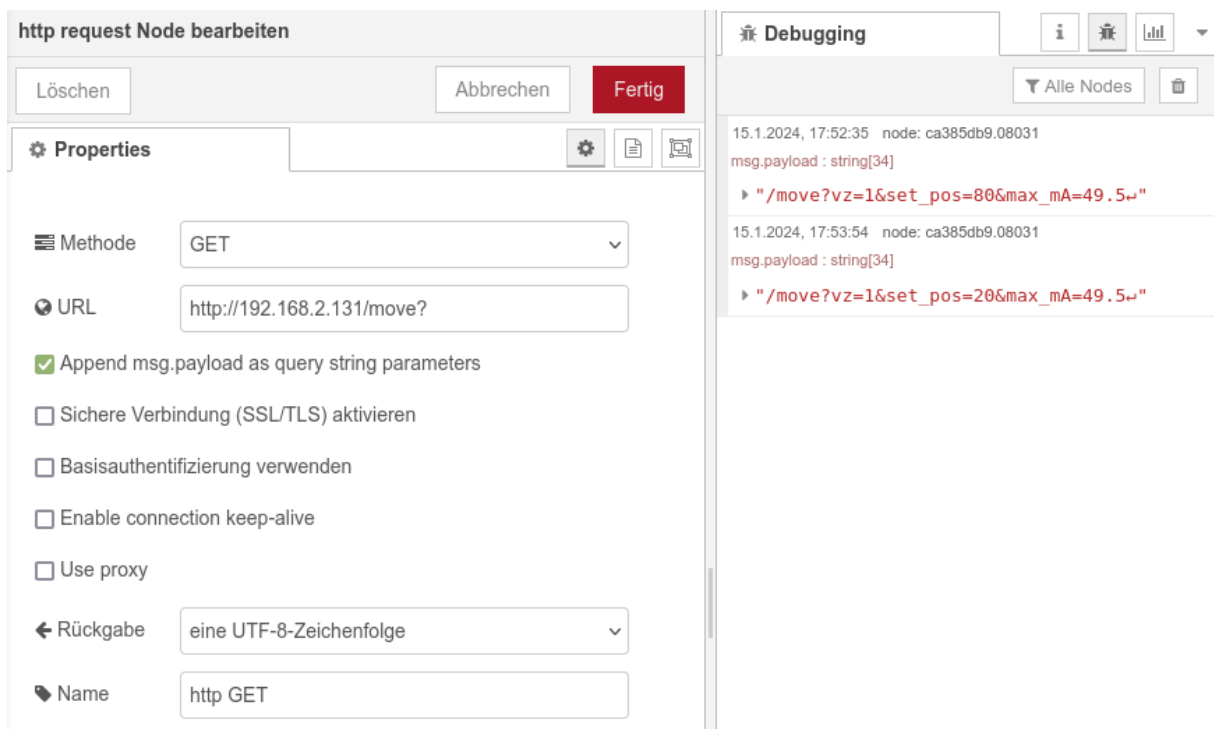


Figure 16: Configuring sample Node-Red node „http request“

User Manual (DRAFT)

---

The debug output when publishing our thermostat values shows the response of our ESP server, when it receives the GET request. In our case, fortunately the ESP echoes back the request (watch the CR ↵ at the end of the debugged payload). So we can see that all parameters from the message were appended to the URL with the addition `"/move?"` for the move command.

Similarly, the commands `"home"` or `"info"` or `"status"` can also be implemented and further processed in node-RED.

Of course, you can delete or deactivate the debug node if you no longer need it.

This completes the task and we have our finished flow:

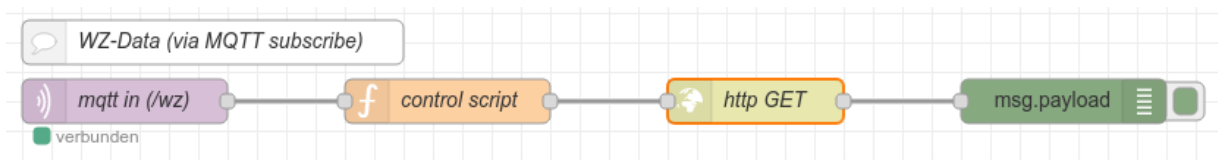


Figure 17: Complete sample in Node-Red for valve control

## 4 Driving the motors by PWM

### 4.1 Introduction to the H-bridge

In our application we are using two „MX1508 Brushed DC Motor Drivers“. Each board can drive two motors independently. So we can run up to 4 motors in total.

These drivers incorporate an H-bridge for each motor and can drive the motors in both directions. A simplified schematic of each H-bridge is shown in the following figure:

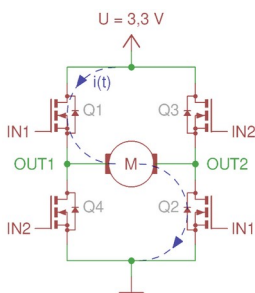


Figure 18: Simplified schematic of each H-bridge

The motor is connected to outputs OUT1 and OUT2.

When IN1 is high and IN2 is low, a current flows from the voltage source through Q1, the motor and through Q2 to GND, moving the motor in one direction.

When IN1 is low and IN2 is high, the current flows from the voltage source through Q3, the motor and through Q4 to GND, moving the motor in the opposite direction.

If both inputs are low, all transistors are high-impedance and no current flows.

Internal logic inhibits a short between U and GND, when both inputs are high. Instead, transistors Q1 and Q3 are switched off, Q2 and Q4 are switched on, shorting the current through the motor. This has a similar effect as a brake.

To drive a motor in one direction, instead of a constantly high level we also can apply a pulse width modulated (PWM) signal. In this case, we are able to control the motor power  $p(t) = u(t) * i(t)$  by changing the duty cycle of the PWM signal.

But there is yet another advantage: During the period in which the H-bridge is switched off, we want to measure the motor's „Back EMF“ (Electro Magnetic Force), a voltage almost proportional to the motor speed with a characteristic shape, allowing to count the number of armature segments that the motor has passed over.

And the PWM period is a good trigger to measure the motor current, which is required to switch off the motor in the event of a blockage.

## 4.2 PWM generation and interrupts

After a couple of experiments we configured two of the PIC's PWM signals with a period of 8 ms, named *slice1, output1* (SaOUT1) and *slice1, output2* (SaOUT2), following the naming convention of the PIC datasheet.

SaOUT1 is used to enable the H-bridge, either asserted to IN1 or IN2, depending on the desired motor direction. The signal is set 7.2 ms high and 800  $\mu$ s Low. When the signal is low, at a low level, no current is driven through the motor and so the motor acts like a generator where we can measure the back EMF voltage ( $V_{\text{BEMF}}$ ). Therefore we enable a PWM interrupt with the falling edge of SaOUT1.

To measure the motor current, we have to wait until the current has settled to a stable value. This is realized with the synchronized PWM output SaOUT2: The signal is set 2 ms high and the falling edge also triggers the PWM interrupt.

The interrupt service routine can detect, which signal was the source of interrupt and perform the desired data acquisition.

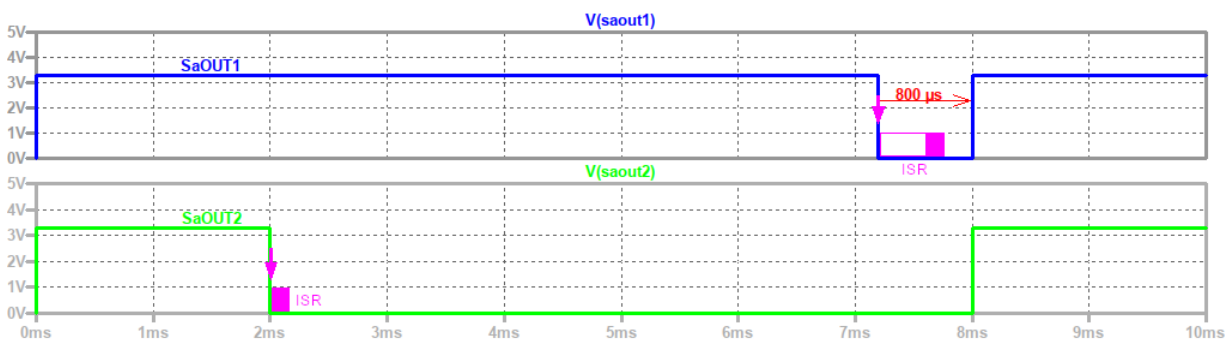


Figure 19: Timing of PWM outputs SaOUT1 and SaOUT2 (LTSpice simulation)

### 4.3 Acquisition of motor current

As shown in the schematics, the motor current is measured by an INA219 breakout board, where the small voltage drop across a shunt is sensed and can be read via the I2C interface. Our INA219 board uses a 0.1 Ohm shunt, but may be different on others. 100 mA motor current would then generate a voltage drop of only 10 mV. Due to the resolution of 10  $\mu$ V, current changes of 0.1 mA can be measured.

These defaults are working pretty good for our application, as we can simply read the INA216 register at address 1, which continuously is updated with the shunt voltage measurement data using the default configuration. No further initialisation or configuration is required.

The shunt voltage register is in signed 16 bit format and LSB = 10  $\mu$ V, which is equivalent to 0.1 mA ( $I = U / R$ ).

As previously explained, we read the motor current in the interrupt service routine, on the falling edge of our PWM signal *slice1\_output2* (*SaOUT2*). This is 2 ms after the H-bridge is switched on and allows the current to stabilize.

To read INA216 register 0x01, we first have to write the desired register address (0x01) to the configuration register at address 0x00. After this, we can start two I2C read cycles from the INA219. The most significant byte (MSB) is read first.

This measurement is carried out by the PIC with a I2C clock of 400 kHz. Address setup and 16 bit read of the shunt voltage register require 138.5  $\mu$ s (add some microseconds for data handling and cleanup).

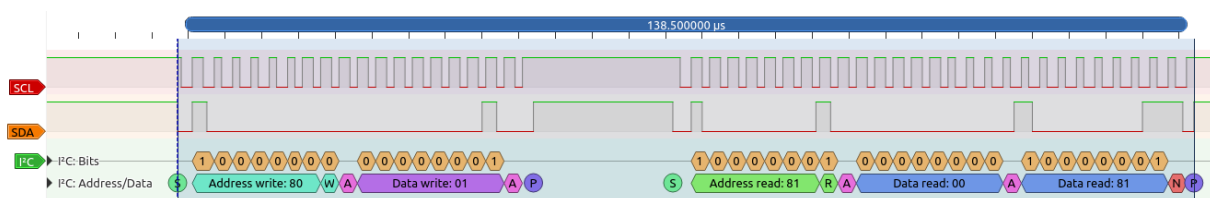


Figure 20: Visualisation of this I2C communication recorded with a logic analyzer

The reading is repeated every PWM cycle (8 ms). But we also want to display the current at standstill, so we must also read the current when the PIC is in the 'idle' state.

These are the function calls to read the motor current from INA219 module:

```
/* Read motor current via INA219 Shunt Current Register
 * (1 LSB = 10μV, Rs = 0,1 Ohm => I / 0.1 mA = Us / 10μV).
 * Exectime ~171 μs.
 */
ina219_reg(1);           // exec time ca. 49 μs (@SCL 400 kHz)
g_mAx10 = ina219_read();  // exec time ca. 72 μs (@SCL 400 kHz)
if (g_mAx10 < 0) g_mAx10 = 0; // offset may cause negative readings
```

### 4.3.1 Exemplary measurement results

The following oscillograms have been recorded using the PIC's digital to analog converter (DAC). This DAC has only 8 bit, and the motor was expected to reach up to 100 mA, which produces a (16 bit) register value of up to 1000 (x 0.1 mA). So the read 16-bit value was shifted 2 bit to the right, which corresponds to a „division by 4“ and truncated to 8 bit.

100 mA  $\rightarrow$  1000 = 0x3E8, divide by 4  $\rightarrow$  0xFA (unsigned) = 250

This value results in  $U_{DAC} = 3.3 \text{ V} \cdot 250 / 255 = 3.24 \text{ V}$  at the output.

Accordingly, we obtain 30.9 mA/volt in the oscillogram..

These are the results in both directions, with no load (free running on the desk) and with some load (applied with my thumb pressing onto the piston).

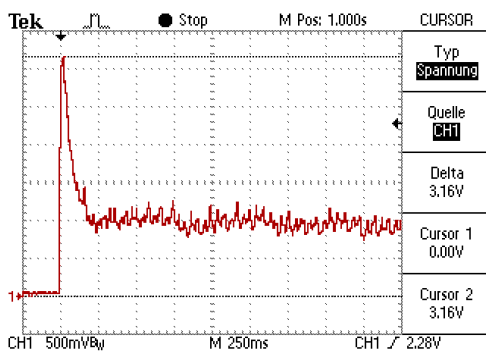


Figure 21: Motor current, open direction, with load

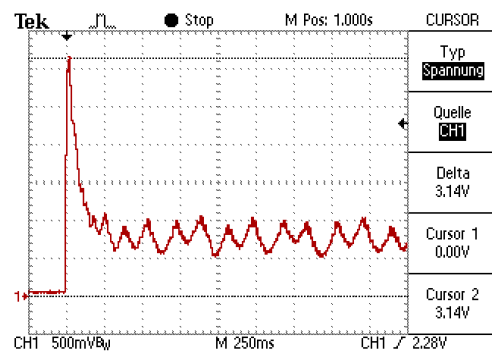


Figure 22: Motor current, open direction, no load

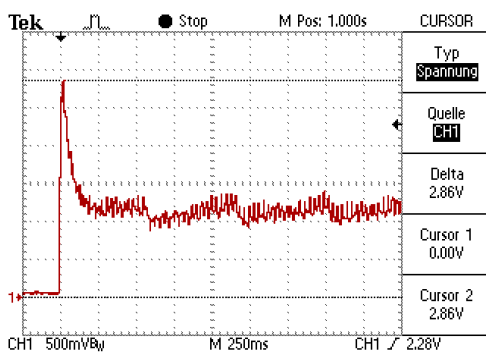


Figure 23: Motor current, close direction, with load

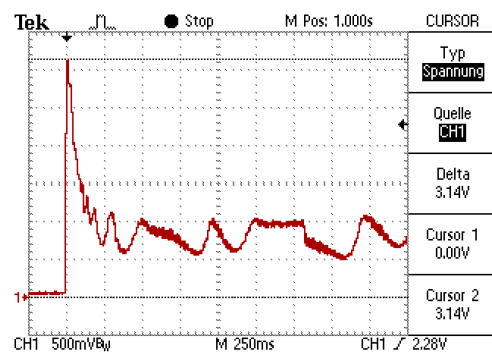


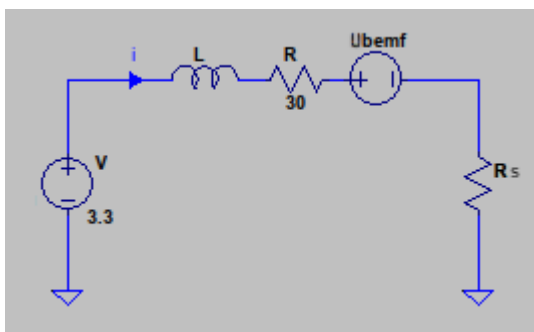
Figure 24: Motor current, close direction, no load

As result we can see, that the motor start-up is completed after approx. 200 ms, causing a peak current close to 100 mA. Without load the motor draws approx. 15 to 30 mA, with moderate load approx. 10 mA more. In reality, when the motor runs against a resistance, the current consumption may increase, hence with a ohmic resistance of  $R = 30 \text{ Ohm}$  and  $U = 3.3 \text{ V} - 2 \times U_{DS} \text{ (H-bridge)}$ , the current cannot exceed 100 mA, which is well below the max. current of 120 mA of the VDMOT. And as soon as the motor turns, the back EMF counteracts the supply voltage until the system reaches a steady state.

#### 4.4 Measuring the motor inductance

In the previous chapter we measured the motor current 2 ms after the H-bridge has been switched on. We have seen that the motor current decreases when the motor is turning, because the generated Back EMF ( $U_{BEMF}$ ) is proportional to the motor speed. Ignoring the effects of brushes and commutation, this can be approximated by the differential equation:

$$L \frac{di(t)}{dt} + R i(t) = V - k_e \omega \quad (\text{Eq. 1})$$



$R_s = 0$  (preliminary)

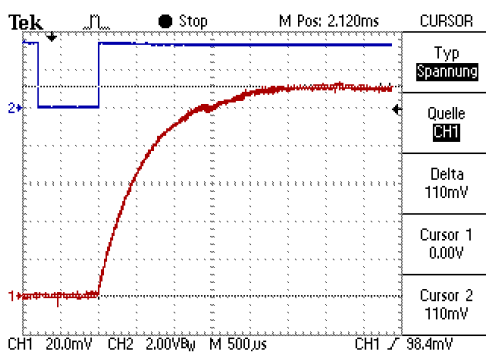
$U_{BEMF} = k_e \omega$

Figure 25: Basic circuit for Brushed DC Motor

The resistance of the motor coil can easily be measured using an ohmmeter. We got approx. 28.5 ohms, let us round this to 30 ohms to include some contact resistance:

$$R = 30 \, \Omega$$

To measure the inductance, we can record the motor current when switched on at standstill because then  $V_{bemf}$  is negligible. As a first step we measured the maximum motor current, as this value is required to compute the time constant of an exponential curve.



max. motor current: 110 mA

CH1: Motor current x  $R_{shunt}$  (1  $\Omega$ )

CH2: PWM

Figure 26: Max. motor current

$$L = 17,6 \text{ mH}$$

after this period, the exponential curve has reached approx. 97 % of its final value.



## 4.5 Measuring the motor speed

Opening the VDMOT housing, we can see the gearbox with 5-stage transmission. A further stage converts the rotation into a linear stroke (without image):

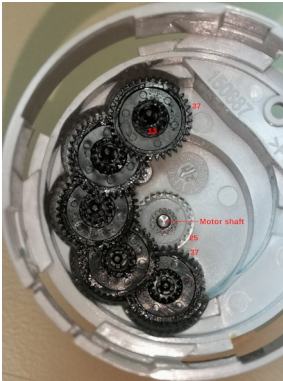


Figure 29: Gearbox of VDMOT

The first gear stage reduces the motor speed  $n_M$  by a factor of 25:37. There are 4 further identical stages with gear ratios of 13:37. So the speed  $n_5$  of the last gear wheel is

$$n_5 = \frac{25}{37} \left( \frac{13}{37} \right)^4 n_M = \frac{714025}{69343957} n_M = 0,010297 n_M$$

By taking a video capture, we counted 19 revolutions of the last gear wheel within 44 seconds. So we can calculate the **motor speed without load** as:

$$n_m = \frac{1}{0,010297} \frac{19}{44 \text{ s}} = 41.937 \text{ s}^{-1}$$

$$n_m = 2516 \text{ min}^{-1}$$

In this case, the motor is running with absolutely no load, but let us remember that one motor revolution takes approx. 24 ms, which corresponds to around 3 PWM periods.

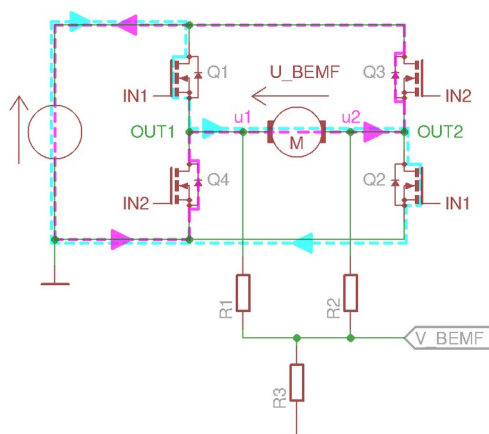
## 4.6 Measuring the back EMF

Each turning motor also acts as a generator. The generated voltage is called the Back Electro Magnetic Force  $U_{BEMF}$  and is proportional to the motor speed. This voltage can be measured, when the motor is free running, that is, when the H-bridge is switched off (at least one motor terminal).

An optimum solution would use a differential amplifier for each motor to measure  $(u_1 - u_2)$  at the motor terminals. Because we want to measure both driving directions, this difference will have a negative sign in one direction which adds some complexity.

Another method could measure  $U_{BEMF}$  when one side of the motor is floating and the other is grounded. Hence, the given break-out board doesn't allow this combination.

A simple solution<sup>5</sup> uses a voltage divider with two symmetrical inputs to measure the sum  $(u_1 + u_2)$  as shown in the next figure. This circuit works for both directions of motor rotation and requires only 3 resistors ( $R_1$  to  $R_3$ ) per motor:



cyan: H-bridge is enabled, motor is driven by supply voltage and flows through enabled FETs Q1 and Q2.

pink: H-bridge is disabled, motor current is driven by inductance and flows through bypass diodes.

Figure 30: Measuring back EMF voltage

As soon as the H-bridge is switched off, the inductance tries to maintain the current in the motor, until the energy stored in the coil is dissipated. The current cannot flow via the blocked FETs, instead it takes the path via the bypass diodes (see the pink dashed line in the figure). Because of the diodes, the potential  $u_2$  is therefore not GND, but has an offset  $-V_f$  and Potential  $u_2$  cannot exceed the supply voltage by more than  $+V_f$ .

The following applies to the voltage divider with 2 inputs:

$$V_{BEMF} = \frac{u_1}{1 + \frac{R_1}{R_2} + \frac{R_1}{R_3}} + \frac{u_2}{1 + \frac{R_2}{R_1} + \frac{R_2}{R_3}} \quad (\text{Eq. 4})$$

5 Texas Instruments Article „Easy cruise control for brushed motors using BEMF“ by Mark Buccini, 2013 ([https://e2e.ti.com/blogs\\_/b/industrial\\_strength/posts/easy-cruise-control-for-brushed-motors-using-bemf](https://e2e.ti.com/blogs_/b/industrial_strength/posts/easy-cruise-control-for-brushed-motors-using-bemf)).

User Manual (DRAFT)

---

As we should treat both directions of rotation equally, we can set  $R1 = R2$ .

The equation shows that for the highest possible output voltage  $V_{BEMF}$ , the resistance  $R3$  should be significantly greater than  $R1$  so that  $R1/R3$  becomes negligible.

Because this voltage divider forms a bypass for the motor current, the current through the voltage divider should be considerably lower than the motor current, i.e. it should be well below 1 mA.

However, the current through the voltage divider should not be too small as otherwise the charging current of the hold capacitor in the ADC will distort the voltage divider.

Choosing  $R1 = R2 = 4k7$  and  $R3 = 10k$  we get

$$V_{BEMF} = 0,405 (u1 + u2) \quad (\text{Eq. 5})$$

When the H-bridge is blocked, the voltage  $u2$  on the low side is determined by the forward voltage<sup>6</sup> of the bypass diode. The anode is connected to GND, the voltage at the cathode is  $u2$ . Therefore

$$u2 = -V_f$$

Negative voltages at ADC inputs are clamped by diodes within the PIC, resistors  $R1$  and  $R2$  must limit the clamping current to below 20 mA (absolut maximum ratings). Since we cannot measure negative voltages with the ADC, the following must apply:

$$u1 > V_f$$

$$U_{BEMF} > 2 V_f \quad (\text{Eq. 6})$$



This means that we cannot expect a useful EMF voltage at very low speeds. Only at medium speeds can the EMF be used for speed estimation.

As our project is designed to be as simple as possible, we do not use differential amplifiers and only use the back EMF for stall detection and speed estimation from medium speed. The motor ramps up so fast, that the position error is much less than the error using a fixed time interval.

It is generally recommended to open and close the valves once a week to prevent limescale build-up. The reference run carried out during this process resets any position difference.

---

<sup>6</sup> We measured a forward voltage of approx. 0.5 V to 0.6 V on the MX1508.

#### 4.6.1 Comparing Simulation and Measurement

When we measured the output signal  $V_{BEMF}$  of the voltage divider of the first PCBs with an oscilloscope, we recorded the following signal course:

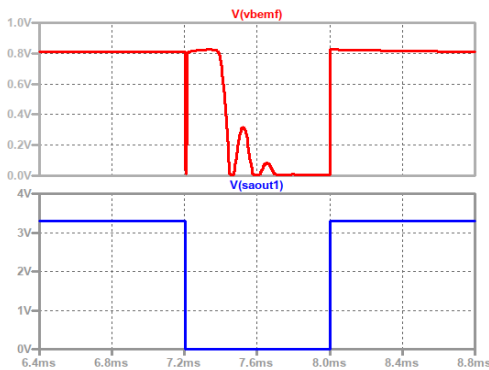


Figure 31: LTspice simulated  $V_{BEMF}$  ( $U_{BEMF} = 0$ )

$C1 = 100 \text{ nF}$ ,  $R1 = R2 = 1 \text{ k}$ ,  $R3 = 500$

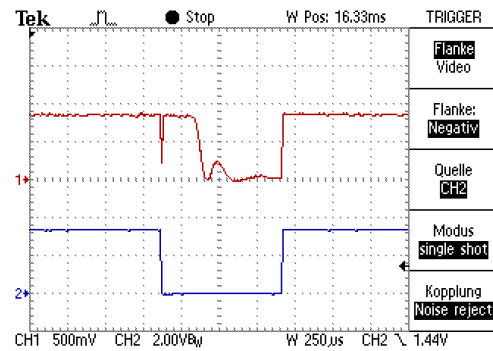


Figure 32: Measured  $V_{BEMF}$  ( $U_{BEMF} = 0$ ), first PCB design

One cause was that the interference suppression capacitor ( $C1$  to  $C4$ )<sup>7</sup> formed a resonant circuit with the motor inductance. New recommended values are 10 pF or simply omit these capacitors.

Next, the voltage divider on the first PCB version was realized using resistors 2 x 1k and 500 ohms, which meanwhile (since PCB rev. A) has been changed to 2 x 4k7 and 10 k.

Here is the result with the new component values:

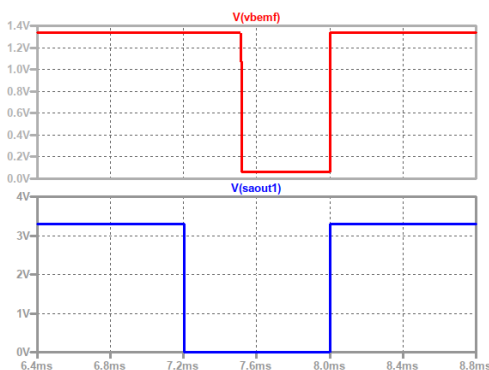


Figure 33: LZSpice simulated  $V_{BEMF}$  ( $U_{BEMF} = 0$ ),  $C1 = 10 \text{ pF}$

new attenuator:  $R1 = R2 = 4 \text{ k}$ ,  $R3 = 10 \text{ k}$

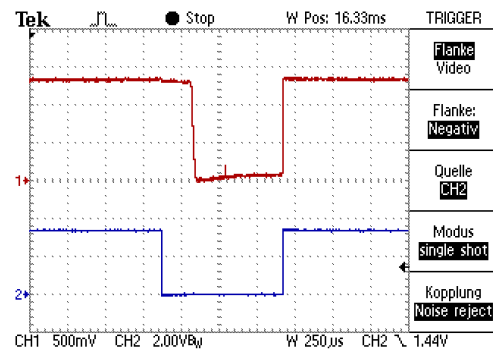


Figure 34: Measured  $V_{BEMF}$  ( $U_{BEMF} = 0$ ),  $C1 = \text{n.a.}$

When disabling the H-bridge, there is no more  $V_{BEMF}$  oscillation and  $V_{BEMF}$  is nearly stable after approx. 320  $\mu\text{s}$ . The voltage level has increased significantly from 0.8 to 1.3 V.

<sup>7</sup> In the first design, these capacitors had 100 nF, which led to the oscillations.

The next oscillogram on the left shows the first 6 PWM cycles with an increasing  $V_{BEMF}$  signal. The oscillogram on the right is a magnification of the 6<sup>th</sup>  $V_{BEMF}$  measurement.

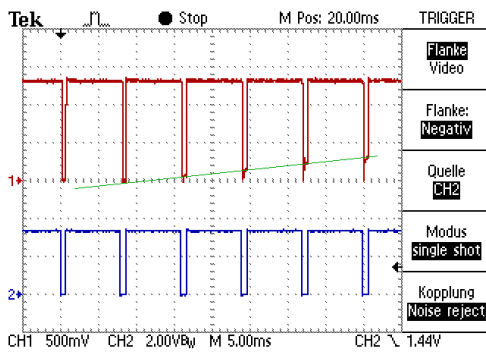


Figure 35: Increasing  $V_{BEMF}$  (first 6 PWM cycles)

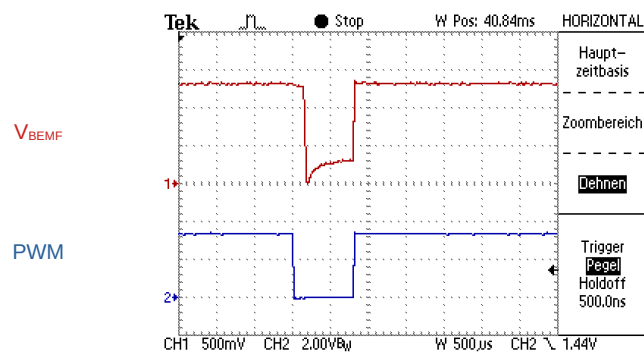


Figure 36: Zoomed  $V_{BEMF}$  (last PWM cycle)

The measurement must be delayed until the energy stored in the inductance has dissipated. The simulation shows, that this settling time becomes shorter as the BEMF voltage increases:

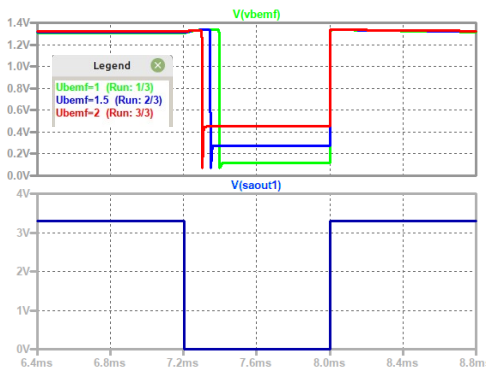


Figure 37:  $V_{BEMF}$  (LTSpice STEP: 1 V, 1.5 V, 2 V)

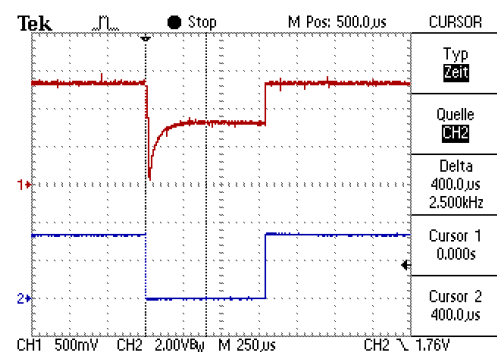


Figure 38: Measured  $V_{BEMF}$  (max speed, with load)

$V_{BEMF}$  in the real measurement rises significantly slower than in the simulation. The reason for this behaviour could not yet be determined, but is probably caused by the components on the H-bridge break-out board.

#### 4.6.2 Back EMF sample oscillograms

Now lets see, what our PIC controller reads when moving position from 0 to 20%.

The following oscillograms have been recorded using the PIC's digital to analog converter (DAC).  $U_{BEMF}$  is expected to reach max. 4 V, which is reduced by the voltage divider to approx. ( $V_{BEMF} = 0.4 * 4 \text{ V} = 1.6 \text{ V}$ ). This value is read by the PIC's 12 bit ADC using a 2.048 V reference voltage, resulting in ADRES values of approx.  $4095 * 1.6 \text{ V} / 2.048 \text{ V} = 3199 = 0xC7F$ . Because the DAC has only 8 bit, the read 12-bit value was shifted 4 bit to the right, which corresponds to a „division by 16“:

$4 \text{ V } (U_{BEMF}) \rightarrow 1.6 \text{ V } (V_{BEMF}) \rightarrow \text{ADRES} = 3199 = 0xC7F, \text{ div. by } 16 \rightarrow 0xC7 = 199 \text{ (DAC)}$

This value results in  $U_{DAC} = 3.3 \text{ V} * 199 / 255 = 2.57 \text{ V}$  at the DAC output. Accordingly, 1 Volt in the oscillogram corresponds to 1.55 volts of the back EMF.

These are the results in both directions, with no load (free running on the desk) and with some load (applied with my thumb pressing onto the piston):

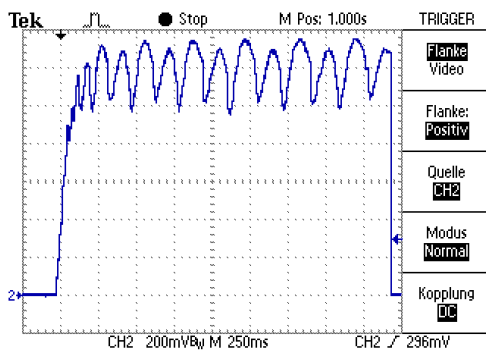


Figure 39: Back EMF voltage, 0 to 20%, no Load

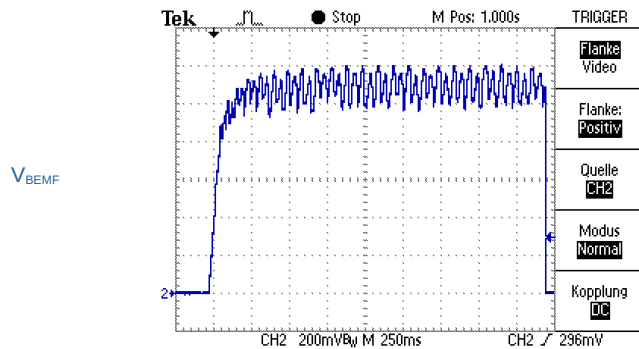


Figure 40: Back EMF voltage, 0 to 20%, with Load

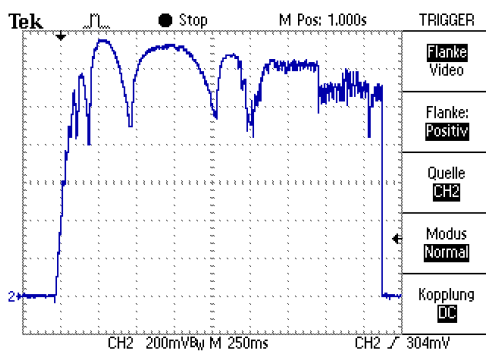


Figure 41: Back EMF voltage, 20 to 0%, no Load

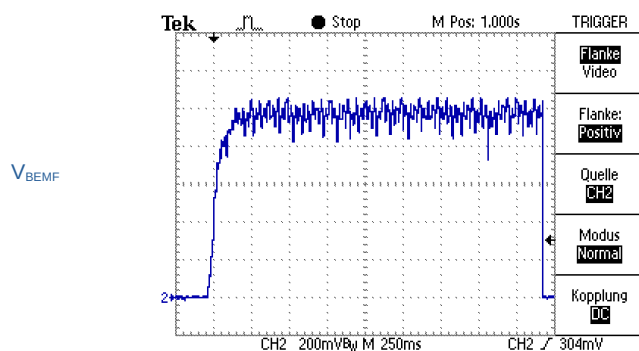


Figure 42: Back EMF voltage, 20 to 0%, with Load

**Note:** When a debugger is connected (ISPDAT/ICSPCLK),  $V_{BEMF}$  of VZ1 and VZ2 cannot be measured. The PICkit4 can also cause a voltage offset when not properly grounded. The signal shape also depends from the motor, maybe the gear is making the difference.

As expected we see, that the speed depends on the load and the direction. The shape of the curve is probably mainly influenced by the friction of the gearbox. With more load, the speed gets lower. The time to stop the motor is nearly zero, in contrast to the start-up which takes approx. 150 ms.

To get an estimation of the position, we simply can add up all the  $V_{BEMF}$  readings, hopefully giving a better position control than using constant time intervals.

#### 4.7 Calculating the motor constant $k_e$

Assuming that max.  $V_{BEMF} = 1.4 \text{ V}$  (no load) corresponds to max speed  $n_m = 42 \text{ s}^{-1}$ , we can compute the motor constant  $k_e$  as follows:

$$U_{BEMF} = V_{BEMF} / 0.404 = k_e \omega = k_e 2 \pi n_m$$

$$k_e \approx 1.4 \text{ V} / (0.404 \cdot 2 \pi \cdot 42 \text{ s}^{-1})$$

$$k_e \approx 13.13 \text{ mV} / \text{s}^{-1}$$

(Eq. 7)

And with a given  $V_{BEMF}$  we can compute the motor speed:

$$n_m = V_{BEMF} / (0.404 \cdot k_e \cdot 2 \pi)$$

(Eq. 8)

In the following oscillogram we have measured the motor run-up. The first significant speed value (after the 3<sup>rd</sup> PWM cycle) is 44 mV. This corresponds to a speed of approx.  $1.32 \text{ s}^{-1}$ . But remember, that the bypass diodes cause an offset that has not been taken into account here!

In the first two PWM cycles, we do not obtain an evaluable BEMF voltage, but from the third cycle onwards the signal is clearly different from zero. Within the third cycle of 8 ms, the motor only moved by 0.0106 revolutions, which gives an impression of the positioning accuracy that can be achieved with this method.

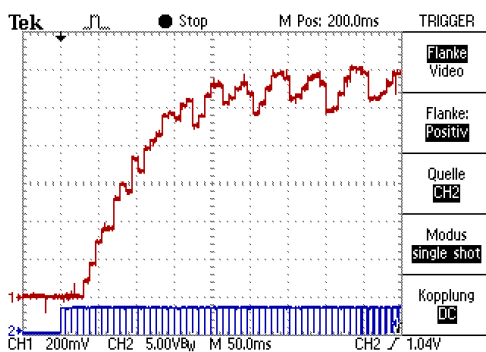


Figure 43:  $V_{BEMF}$  and PWM signal at motor run up

 $V_{BEMF}$ 

PWM

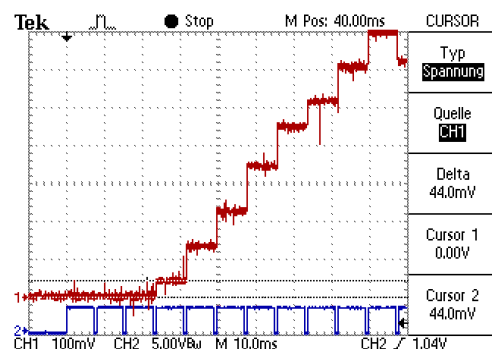


Figure 44:  $V_{BEMF}$  and PWM at Motor run up, Detail

## 5 Motor simulation with LTspice XVII

To gain an even deeper understanding of our circuit, we carried out an LTspice simulation. The following figure shows the circuit diagram.

### 5.1 LTspice Circuit diagram

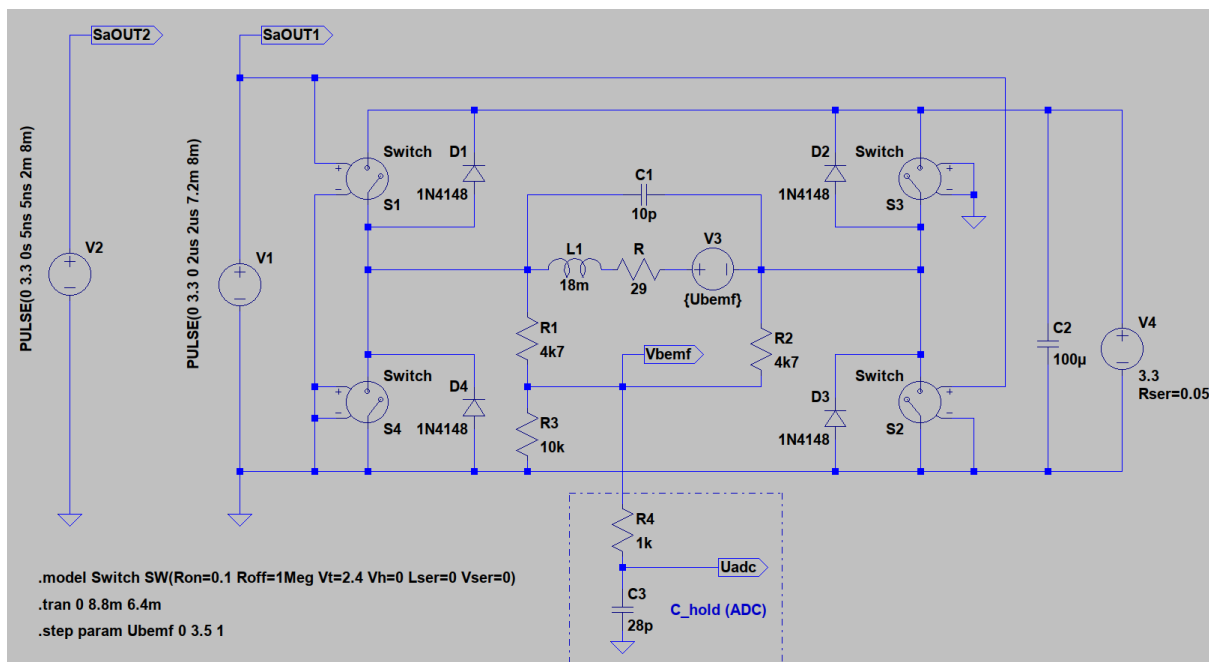


Figure 45: LTspice circuit diagram

Component V1 (**SaOUT1**) generates our PWM signal to control the H-bridge.

Component V2 (**SaOUT2**) is just simulated to plot our two PWM signals.

The H-bridge is built using 4 switches, which have been configured with **Ron = 0.1** ohms.

**1N4148** type diodes are used to bypass the „FETs“.

Power supply **V4** has 3.3 volts with some series resistance (50 mΩ) and with a **100 µF** supporting capacitor.

Our voltage divider (**R1 to R3**) uses the values as used in the calculations before.

As the simulations showed, interference suppression capacitors (**C1**) above approx. 100 pF in parallel with the motor cause oscillations when measuring the back EMF, so they should only have very small capacitances or preferably be omitted altogether.

As **V<sub>BEMF</sub>** is dependent on the speed, it was simulated using a STEP directive from 0 to 3.5 V in steps of 1 V.



## 6 Data Logger

Starting from Version 0.8, the software logs up to 1024 readings of VBEMF and the motor current in two arrays of size LOGSIZE (= 1024 actually):

```
volatile uint8_t bemf_log[LOGSIZE]; ///< data logger: g_vbemf >> 4
volatile uint8_t curr_log[LOGSIZE]; ///< data logger: g_mAx10 >> 2
```

To reduce memory space, the 12 bit value `g_vbemf` is shifted 4 bit, and the motor current `g_mAx10` is shifted 2 bit to the right, so that the values fit into a single unsigned byte.

The data logger is active on every MOVE or HOME travel until the buffer is filled. If the MOVE or HOME travel ends before LOGSIZE readings have been written, the buffer is filled up with zeros.

The buffer content can be read by sending the following GET command via a web browser on our LAN:

```
http://192.168.2.131/logdata?
```

After this command, the ESP sends a request to the PIC controller to transmit all the 1024 data sets in the form

```
index i, bemf_log[i], curr_log[LOGSIZE]\n
```

The ESP saves all the data records in file `/logdata.csv`, where it can be downloaded via the LittleFS manager (p.e. <http://192.168.2.131/fs.html>) and processed p.e. with Excel or Calc. The following diagram shows a sample log with the raw values converted to mA and Volts respective ( $\text{mA} = 4 * \text{curr\_log}[i]/10$  and  $U_{\text{BEMF}}[\text{V}] = 16 * 2,048 * \text{bemf\_log}[i]/4095/0,405$ ):

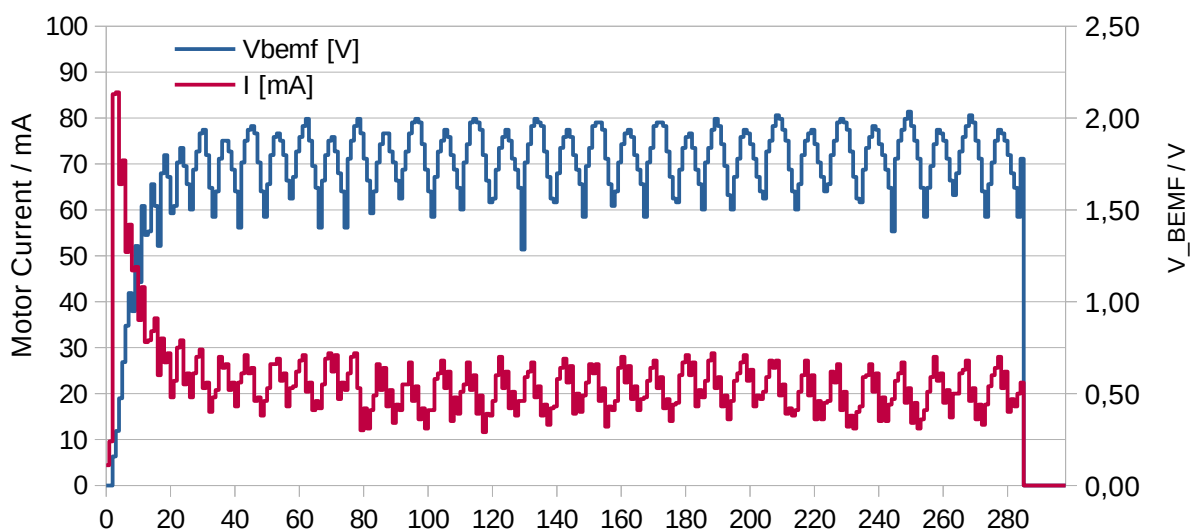


Figure 46: Sample log diagram (motor running on the desktop)