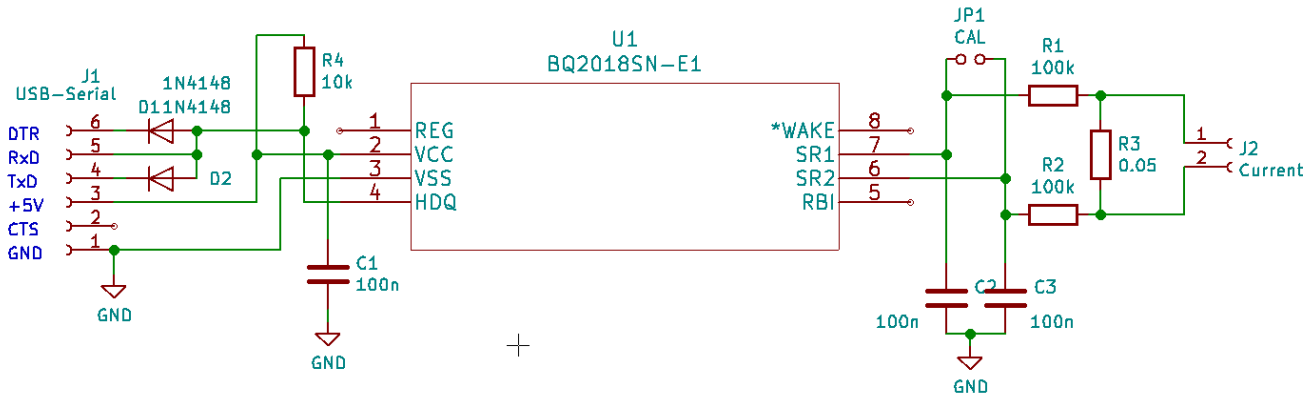
BQ2018 Battery monitor (“Gas gauge”)

Purpose

The circuit connected to a computer, is used to monitor charge and discharge of a battery (or any other component). The software registers charge in As and mean current in A.

Hardware



J1 is connected to a USB-Serial interface
J2 is the current terminal. The monitored current flows over the shunt resistance R3.
The diode to DTR is needed only when the BQ2018 shall be calibrated.

Definition of charge and discharge:

U_{12} positive (current flowing from 1 to 2) → charge
U_{12} negative (current flowing from 2 to 1) → discharge

Hardware limitation:
The BQ2018 has a maximum input voltage of 200mV between SR1 and SR2, so the maximum current that can be measured is

$$I_{max} = \frac{U_{Smax}}{R_3} = \frac{200\,mV}{50\,m\Omega} = 4\,A$$

Coulomb counting

The BQ2018 contains a voltage controlled oscillator (voltage to frequency converter VFC) controlled by the voltage on the shunt resistor, and two counters incremented by the pulses of the VFC.

The VFC gain is 22.2Hz/V for the BQ2018, so for example with I = 1A and a shunt resistance of 50mOhm:

$$U_S = R_S \cdot I = 0.05\,\Omega \cdot 1\,A = 50\,mV$$

$$f = 22.2\,Hz/V \cdot 0.05\,V = 1.1\,Hz$$

Depending on the polarity of the shunt voltage (charge or discharge), the discharge (DCR) or the charge (CCR) count register is incremented. So this registers contain information about the charge or discharge in As.

In our example (with I = +1A, flowing from 1 to 2), the charge count register CCR would be incremented with a frequency of 1.1Hz (so every 0.909 seconds). More current would mean a faster counting, less current a slower counting.

This example shows that it makes no sense to read the CCR and DCR very often, as the change in register value is very slow. The resolution depends on the time needed for the register(s) to increment. For example a resolution of 1mA would mean a frequency resolution of 1.1mHz. At 1mA, the charge or discharge register would be updated every 1/1mHz = 909s = 15.14min.

There are 2 other counters, the charge time count register (CTC) and the discharge time count register (DTC) incrementing with a rate of $r_T = 4096$ per hour, only when a current is flowing. With these, the charge and discharge time can be calculated:

$$\Delta t = \frac{TC}{4096/h} \cdot 3600\frac{s}{h}$$ where TC is either the charge or discharge time counter.

<u>Calculation of the charge Q</u>

In general, the counter frequency is

$$f = G_{FC} \cdot U_S = G_{FC} \cdot R_S \cdot I$$

Supposed all counters were reset at the beginning, the charge or discharge in As can be calculated like this:
The VFC frequency is

$$f = G_{FC} \cdot U_S = G_{FC} \cdot R_S \cdot I$$

The charge counters count to a value of

$CR = f \cdot \Delta t$ where CR is the charge or discharge count register

$\rightarrow \quad CR = G_{FC} \cdot R_S \cdot I \cdot \Delta t$

With the charge / discharge quantity $Q = I \cdot \Delta t$:

$$Q = \frac{CR}{G_{FC} \cdot R_S}$$

Getting the average current

$$I_{average} = \frac{Q}{\Delta t}$$

Confusing information in the datasheet

As seen above, the VFC gain is the only thing needed to calculate the charge Q.
In the datasheet at some place we find the information that the rate of DCR and CCR is 12.5µV per hour. Said like this it makes no sense.
But we can calculate easily that the minimum voltage to increment DCR or CCR in one hour is 12.5µV. That makes sense.

With this in mind, the information found in another place, that the counters count in 12.5µVh increments also makes sense.

Anyway, it would have been less confusing to stick to the notion of VFC gain.

Communication

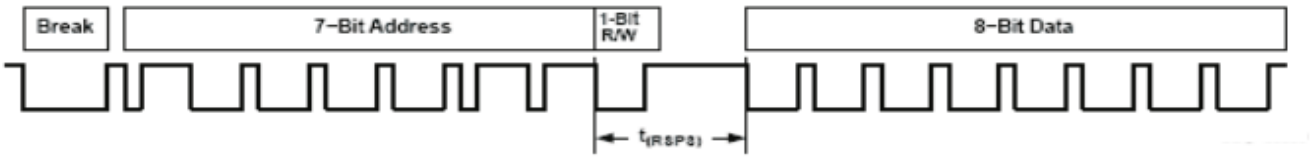
The BQ2018 communicates over a one wire bus called HDQ.

Information is sent and received over one wire. This is possible similarly to other one wire protocols, by pulling the data line (that has a steady high potential) low, either from the master or from the BQ2018 slave.

Communication is command based. The master sends a command byte, and the BQ2018 replies with a data byte. The command byte is nothing other than an internal register address of the BQ2018.

At the beginning of the communication the HDQ line is pulled low for at least 190µs (BREAK signal).

Then the bits (LSB first) are transmitted serially. The bit information is coded into the length of the Low pulses (short = 1, long = 0).



In an overview:

- LSB first, MSB last
- 0 → long negative pulse, 1 → short negative pulse
- Idle state is High
- At the beginning of a transmission, a BREAK signal is sent: 0 for t_B >= 190us.

- The communication is command-based. The command byte is a register address.
  The chip responds with 1 byte of data to a read command
- Distinction between Read and Write commands is made with the command's MSB (MSB = W/R), bit 7
  W/R = 0 → read
  W/R = 1 → write

The communication can be done with a bit banging technique, but the datasheet mentions an elegant method: using an UART with a baud rate of 57600 and 2 stop bits.

Communication using an UART

Why and how does this work?

The hardware side

(See schematic)

The UART has 2 wires: RxD and TxD, whilest HDQ is a one wire bus.
We can connect RxD directly to the HDQ pin, with a pullup resistor.
This way we can receive data.
To send data the UART must be able to pull the HDQ line low, using TxD.
In the datasheet we see a rather complicated method using a 7404 as inverter and a MOSFET to pull HDQ down.

Using only a diode between HDQ and TxD is working also. When TxD is high the diode is in idle or reverse state. When TxD is low, the diode is conducting, thus pulling HDQ low.

The bit side

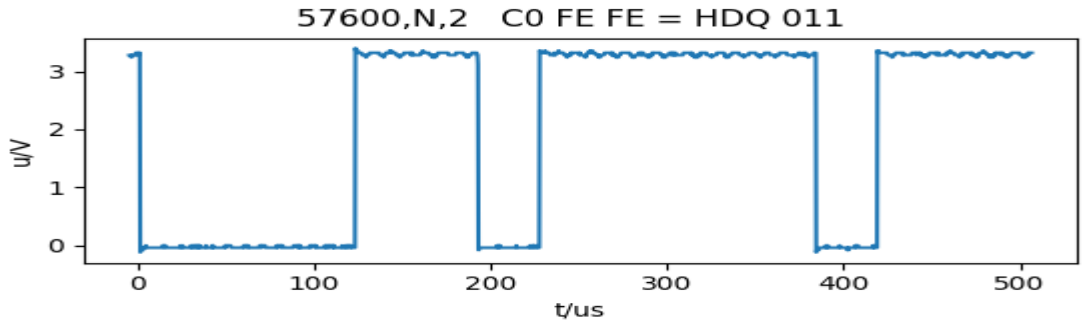
For a revision of general serial communication, see appendix.

The trick is that one HDQ bit is transmitted by using two special bytes (0xFE and 0xC0) for the transmission, so that the HDQ timing is fulfilled. So one UART byte corresponds to one HDQ bit.

One HDQ byte takes 8 UART bytes to be transmitted.

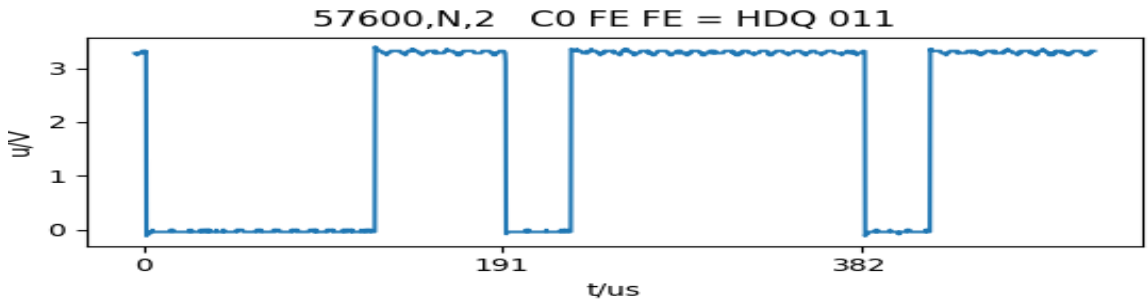
The effective HDQ transmission frequency is in the order of 5kHz.

Let's take a look at a simple example: the transmission of the HDQ bits 011:



- Transmission parameters are 57600 baud, no parity, 2 stop bits
- The HDQ bits are sent as UART bytes, so 1 HDQ bit corresponds to 11 UART bits (start and 2 stop bits included), that means one HDQ bit takes 191us. (Datasheet: 190...250us cycle time)
  That gives us a transmission rate of 5.22kHz.
- HDQ 0 is sent as UART 0xC0
- HDQ 1 is sent as UART 0xFE

The same plot rescaled to HDQ bit length (= UART byte length) shows one long(0) and two short (1) negative pulses:



- 0 bit pulse: 122us (datasheet: min. 100us)
- 1 bit pulse: 33us (datasheet: max. 50us)

So this can work!

This was for sending bytes. When receiving data, the following translation is applied

- byte value >= 0xF0: HDQ bit = 1
- byte value < 0xF0: HDQ bit = 0

Don’t forget to send a BREAK signal (does a Low pulse for TxD) at the beginning of every communication!

Communication via Python

As Python has a good Serial library, the communication is rather easy.

My software is inspired by https://github.com/zerog2k/hdq_python

From this I took the basic routines to read and write registers:

```
import serial

port = '/dev/ttyUSB0'
ser = serial.Serial(port, 57600, stopbits=2, timeout=1)

HDQ_BIT1 = 0xFE
HDQ_BIT0 = 0xC0
HDQ_BIT_THRESHOLD = 0xF8

def reset():
    ser.send_break()
    ser.read()

def write_byte(byte):
    #convert and write 8 data bits
    buf = bytearray()
    for i in range(8):
        if (byte & 1) == 1:
                buf.append(HDQ_BIT1)
        else:
                buf.append(HDQ_BIT0)
        byte = byte >> 1

    ser.write(buf)
    ser.read(8)      # chew echoed bytes

def read_byte():
    #read and convert 8 data bits
    buf = ser.read(8)
    buf = bytearray(buf)
    # lsb first, so reverse:
    buf.reverse()
    if debug:
        print ("recv buf:", binascii.hexlify(buf))
    byte = 0
    for i in range(8):
        byte = byte << 1
        if buf[i] > HDQ_BIT_THRESHOLD:
            byte = byte | 1
    return byte

def uint16le(bl, bh):
    word = bh << 8 | bl
    return word

def read_reg(reg):
    write_byte(reg)
    return read_byte()
```

```
def write_reg(reg, byte):
    write_byte(0x80 | reg)
    write_byte(byte)
```

Once you have these, you can test the communication (Yes, there is nothing more frustrating than a powerful circuit with whom you can't establish a communication!).

A simple test is to read all registers:

```
def read_all_regs():
    for i in range(0, 0x80):
        b = read_reg(i)
        print (hex(i), "\t", hex(b))
read_all_regs()
```

The datasheet gives information about the registers.

With this, more specific functions can be built to manipulate DCR, CCR, DTC, CTC:

```
def reset_charge_regs():
    write_reg(0x74, 2)

def reset_discharge_regs():
    write_reg(0x74, 1)

def reset_discharge_time():
    write_reg(0x74, 8)

def reset_charge_time():
    write_reg(0x74, 16)


def read_discharge_regs():
    DCR = read_reg(0x7E) + read_reg(0x7F) * 0x100
    #print('Discharge registers: ' + str(DCR))
    return DCR

def read_charge_regs():
    CCR = read_reg(0x7C) + read_reg(0x7D) * 0x100
    #print('Charge registers: ' + str(CCR))
    return CCR

def read_CCR_and_DCR():
    CCR = read_charge_regs()
    DCR = read_discharge_regs()
    return CCR, DCR

def read_CTC_and_DTC():
    CTC = read_reg(0x76) + read_reg(0x77) * 0x100
    DTC = read_reg(0x78) + read_reg(0x79) * 0x100
    return CTC, DTC
```

After this, we can do higher level functions:

```
def reset_all_charge_and_discharge_regs():
    reset_charge_regs()
    reset_discharge_regs()
    reset_charge_time()
    reset_discharge_time()

def get_charge_and_discharge_time():
    # in seconds
    # time counters increment at a rate of 4096 per hour
    CTC, DTC = read_CTC_and_DTC()
    t_charge = CTC * 3600.0 / 4096
    t_discharge = DTC * 3600.0 / 4096
    return t_charge, t_discharge
```

This function gives the charges in As:

```
def get_charge_As(Rs, GFC):
    CCR, DCR = read_CCR_and_DCR()
    QC = CCR / (Rs * GFC)
    QD = DCR / (Rs * GFC)
    return QC, QD
```

When the charge Q is known, we can deduce the average current:

```
def get_mean_current(Rs, GFC):
    t_charge, t_discharge = get_charge_and_discharge_time()
    QC, QD = get_charge_As(Rs, GFC)

    try:
        I_charge = QC / t_charge
    except:
        I_charge = None

    try:
        I_discharge = QD / t_discharge
    except:
        I_discharge = None

    return I_charge, I_discharge
```

A main program using these functions could look like this:

```
import serial
import time

debug = 0

port = '/dev/ttyUSB0'
ser = serial.Serial(port, 57600, stopbits=2, timeout=1)

HDQ_BIT1 = 0xFE
HDQ_BIT0 = 0xC0
```

```
HDQ_BIT_THRESHOLD = 0xF8


try:
    ser.open()      # open serial, if not already open
except:
    pass

ser.reset_input_buffer()
ser.reset_output_buffer()
reset_all_charge_and_discharge_regs()

Rs = 0.05
GFC = 22.2

t0 = time.time()

while(1):
    dt = time.time() - t0


    QC, QD = get_charge_As(Rs, GFC)
    I_charge, I_discharge = get_mean_current(Rs, GFC)
    print(dt, '\t',   QC, '\t', QD, '\t', I_charge, '\t', I_discharge)
    time.sleep(1)
```

Remarks:

- In this example, the results are acquired and printed every second. As Coulomb counting is a very slow process (if we want an a good resolution), it makes sense to use a longer delay for practical purposes.
- The primary measured thing is the charge in Coulomb (= As), and not the current. The calculated value of the current is an average value. We have no information about the instantanous value of the current. For this we would have to directly use the shunt voltage. The BQ2018 does an integration of current over time.

Calibration

The BQ2018 as all analog circuits, has an offset. Current zero is not measured as zero.

The chip can be calibreated to eliminate this offset.

The procedure takes an hour, as we must have a good resolution.

To do the calibration, the jumper CAL has to be set that shortens the sense input.
Then the calibration bit (bit 6) in the register 0x75 must be set to 1 and HDQ must be pulled low for one hour.

The pulling low of HDQ is done via DTR and the diode D1.

Here is the Python function:

```
def calibrate():
    write_reg(0x75, 0b001000000)

    print("Calibration countdown 3600...0 (1 hour)")
    for i in range(3601, 0, -1):
        ser.dtr = True          # Low!
        print(i)
        time.sleep(1)
    x = read_reg(0x75)
    print(x)
    ser.dtr = False             # High!
    print('Leaving calibration mode')
```

At the end of the process, bit 6 of register 0x75 must have returned to 0.

Appendix: General serial communication

Here is a simple example of the transmission of 2 bytes

Example: Transmission of 0x4C, 0x5B, 57600 Baud, 8 bits, 2 stop bits
Binary: 0100 1100 0101 1011
Decimal: 76 91
Time for 1 bit: 1s/57600 = 17us
Remember: the transmission is LSB first!

Python test program on the PC:

```
import serial
s=serial.Serial('/dev/ttyUSB0', baudrate=57600, bytesize=8, parity='N', stopbits=2,
  timeout=None)
s.write(bytearray([0x4C, 0x5B]))
s.close()
```