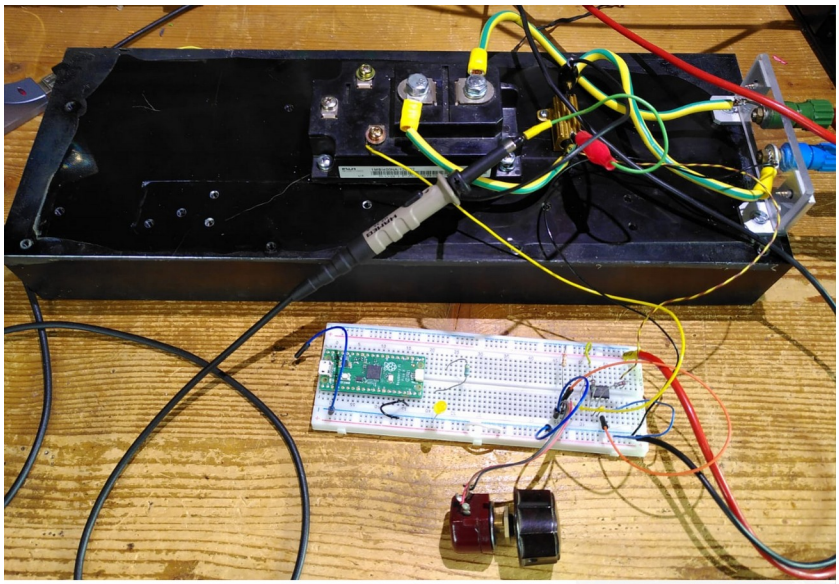
Elektronische Last

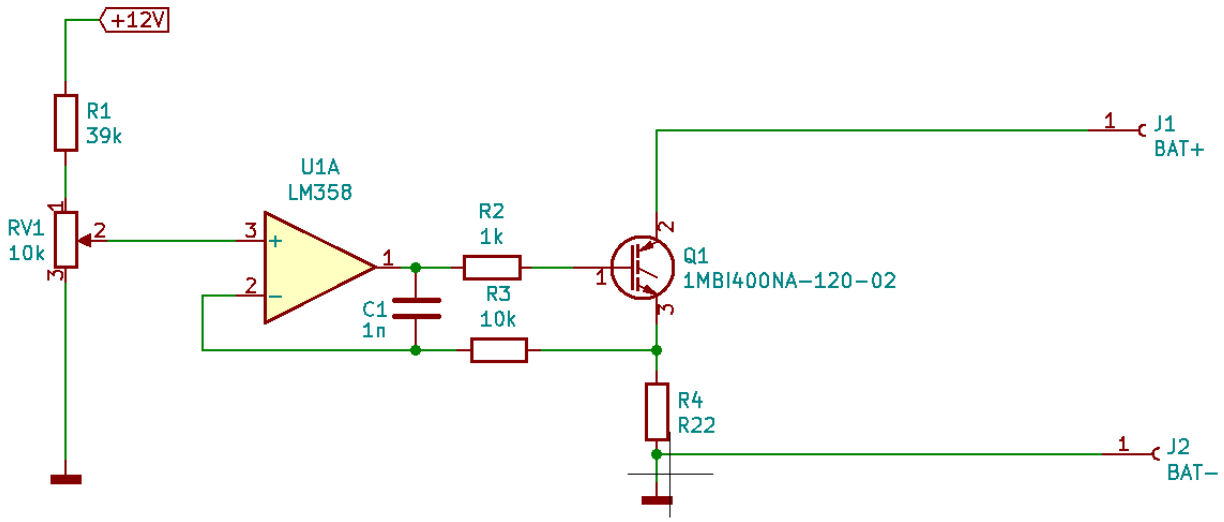
Von jean-claude.feltes@education.lu

1. Zweck

Untersuchung von Akkus, Solarpaneln, Aufnahme der IU und MPP-Kennlinien

2. Stromsenke, 1. Versuch





Als Transistor benutze ich einen herumliegenden (überdimensionierten) IGBT, der 1200V und 400A aushält. Auch der Kühlkörper ist recycelt.

Die obige Schaltung ist eine einfache geregelte Stromsenke.

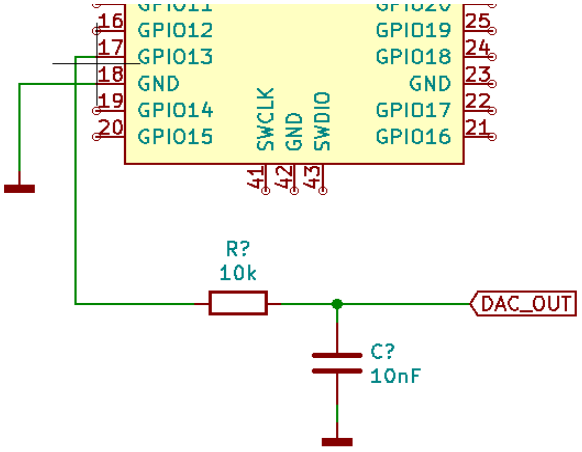
Durch die Gegenkopplung ist die Spannung an R4 gleich der Spannung am Schleifer des Potentiometers. Der Strom ist also immer

$I = \frac{U_{R4}}{R_4} = \frac{U_{IN}}{R_4}$ wobei U_{IN} die Spannung am P-Eingang des OPV ist.

Mit einer Steuerspannung von max. 3.3V die später von einem Raspi Pico geliefert wird, ergibt sich also hier ein maximaler Strom von ca. 15A.
Wegen der kapazitiven Last durch das Gate besteht eine Neigung zu Schwingungen. Diese werden durch C1 vermieden (→ R.A. Pease: Troubleshooting in Analogschaltungen, Elektor).

3. Steuerung durch einen Raspi Pico

DAC



Mit einem PWM-Signal und einem Tiefpass wird ein analoges Steuersignal erzeugt.

Die schon vorhandene PWMc – Klasse (Modul pwmc.py, → https://github.com/jean-claudeF/PWM-generator/tree/main/Micropython/lib) ist leicht durch eine Funktion zum Setzen der Spannung zu erweitern:

```
class PWMc:
    def __init__(self, pin, freq = 5000):
        # set PWM pin + frequency
        self.pwm = PWM(Pin(pin))
        self.pwm.freq(int(freq))
        self.pwm.duty_u16(0)


    def set_pwm(self, value):
        # set PWM value 0.0 to  1.0
        self.value = value
```

```
        pwmval = int(65535 * value)
        if pwmval > 65535: pwmval = 65535
        if pwmval < 0: pwmval = 0
        self.pwm.duty_u16(pwmval)
        return pwmval


    def set_freq(self, freq):
        self.pwm.freq(int(freq))
        self.set_pwm(self.value)
        return self.pwm.freq()

    def stop(self):
        # deinit PWM
        self.set_pwm(0)
        self.pwm.deinit()

    def set_dac(self, v):
        self.set_pwm(v/3.3)
```

Hier ein Testprogramm, um das Zeitverhalten der Schaltung zu testen:

```
...
cs = PWMc(13, 1E6)
while True:
    cs.set_dac(0)
    time.sleep(0.001)
    cs.set_dac(3)
    time.sleep(0.001)
```

Beim Pico kann man die Frequenz recht hoch einstellen, hier 1MHz.
So genügt ein einfaches Filter um ein glattes Analog-Signal zu erzeugen, dachte ich.

Mit 10k, 1nF ist die Grenzfrequenz $f_g = \frac{1}{2\pi RC} = 15.9\,kHz$

Allerdings zeigt das Oszilloskop doch noch eine gewisse Welligkeit, so dass es mir ratsam erschien, C auf 10nF zu vergrössern. Nun braucht das Einschwingen des Filters allerdings ca. 0.5ms,

die Grenzfrequenz ist 1.59kHz.

ADC

Zum Erfassen der Lastspannung wird der interne ADC benutzt.

Zunächst soll aber mit einem Testprogramm das Zusammenspiel von DAC und ADC kontrolliert werden, speziell das Zeitverhalten. Hierzu wird ein zwischen 0 und einem Maximalwert springendes Analogsignal erzeugt und dieses dann wieder eingelesen:

```
...
def get_voltage(nb_mean = 3):
    v = 0
    for i in range(0, nb_mean):
        v += adc1.read_u16()
    v = v/nb_mean * (3.3 / 65535) - 0.008   # 8mV Offset
```

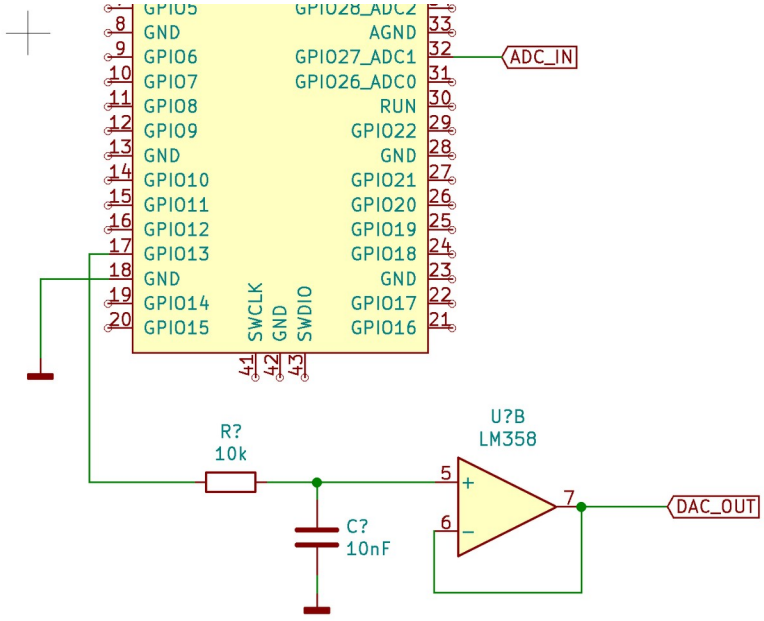
```
    return v

if __name__ == '__main__':

    cs = PWMc(13, 1E6)
    dt = 0.005
    while True:
        cs.set_dac(0)
        time.sleep(dt)
        v1 = get_voltage(nb_mean = 20)

        cs.set_dac(1)
        time.sleep(dt)
        v2 = get_voltage(nb_mean = 20)
        print(v1, v2)
```

Die get_voltage Funktion erlaubt eine Mittelung über mehrere Werte, was angesichts der doch recht “verseuchten” Mikrocontroller – Umgebung sinnvoll ist.

Bei grösseren dt – Werten klappt alles, bei kleineren nicht mehr, obwohl das Filter doch nur 0.5ms zum Einschwingen braucht.

Was ist hier los? Das Oszilloskop zeigt es: die Belastung des DAC mit dem ADC bringt eine kräftige Verzögerung der Signalflanken ins Spiel. Der “hochohmige” ADC stellt eine kräftige kapazitive Belastung dar.

Damit ist die Abhilfe klar: hinter das Filter muss ein Impedanzwandler geschaltet werden.



(Im Test: DAC_OUT + ADC_IN verbunden)

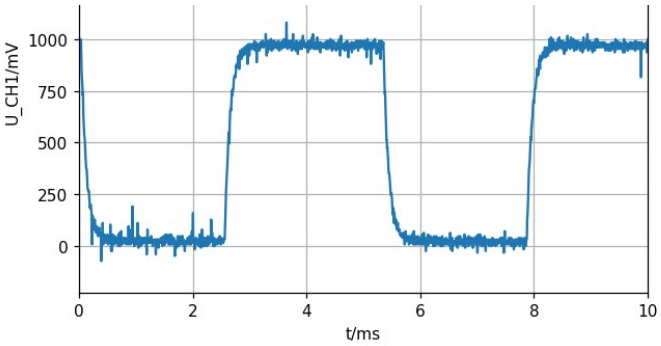
Damit zeigt das Testprogramm Werte die ganz OK sind:

```
0.0008221558 0.9765768
0.0008221558 0.9766573
0.0008624392 0.9766975
0.0009027235 0.9769393
0.0009027235 0.9771004
```

```
0.0008221558 0.9767782
```

(Sollwerte 0 und 1)

Auch mit dt = 2ms funktioniert es noch gut, was Werte und Oszillogramm zeigen:



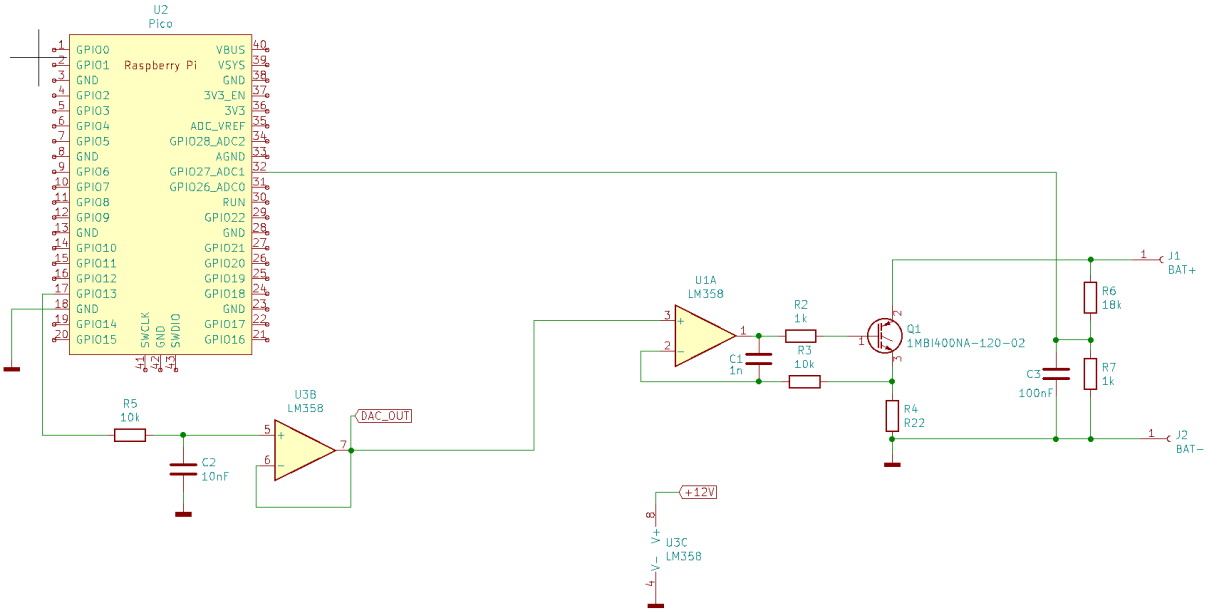
Kleinere Werte bringen merkbare Fehler.

Fazit:

Nach dem Ausgeben des DAC-Wertes sollte ca. 2ms gewartet werden, ehe die Spannung mit dem ADC gemessen wird.

(bei 10k, 10nF für das Filter und 1MHz PWM).

4. Elektronische Last



(Eigentlich ist der linke OPV nicht nötig da die Stromsenke schon einen hochohmigen Eingang hat. Es sind aber ohnehin 2 OPV in einem LM358 vorhanden, und es ist praktisch einenMesspunkt DAC_OUT zu haben.)

Softwaremässig wird noch ein wenig erweitert:

```
class CurrentSink(PWMc):
    def __init__(self, pin, Rs = 0.22, freq = 1E6):
        self.pwm = PWM(Pin(pin))
        self.pwm.freq(int(freq))
        self.pwm.duty_u16(0)
        self.Rs = Rs
        self.ki = .9191176

    def set_current(self, I):
        self.set_dac(self.Rs * I * self.ki)

def get_voltage(nb_mean = 3, k = 18.67):
    v = 0
    for i in range(0, nb_mean):
        v += adc1.read_u16()
    v = v/nb_mean * (3.3 / 65535)
    v = v * k
    return v

def set_and_measure(I):
    # sets I and returns voltage
    cs.set_current(I)
    time.sleep(0.003)
    v = get_voltage(nb_mean = 20)
    return v
```

Hinzugekommen ist die Klasse CurrentSink die die Einstellung des Stroms übernimmt.
Sie erbt von der PWMc-Klasse.

Dann gibt es noch eine Funktion zum Bestimmen der Spannung und eine Funktion set_and_measure die den Strom setzt und die Lastspannung zurückgibt.

Hiermit kann die Schaltung schon gut getestet werden indem zwischen 0 und einem Maximalwert des Stromes hin und hergeschaltet wird. Ich benutze dabei das Netzteil JT-RD6006 welches alle relevanten Daten anzeigt

hier ein Beispiel:

```
  ...
cs = CurrentSink(dac_pin, Rs = 0.22, freq = 1E6)

dt = 3
I1, I2 = 0, 5
while True:
    v1 = set_and_measure(I1)
    time.sleep(dt)
    v2 = set_and_measure(I2)
    print(I1, I2, v1, v2)
    time.sleep(dt)
```

5. MPP-Kurven

Eine wichtige Aufgabe des Gerätes sollte es sein, die Kurven U = f(I) und P = f(I) von Solarpaneln aufzunehmen. Dies ist mit einer kleinen Änderung der Software schon möglich:

```
...
cs = CurrentSink(dac_pin, Rs = 0.22, freq = 1E6)

# Range in mA to avoid floating point
for i in range(0, 5100, 100):
    I = i/1000
    v = set_and_measure(I)
    P = v * I
    print(I, v, P)
    #time.sleep(0.01)
    if v < 2.5:
        break
set_and_measure(0)
```

Da unterhalb von 2.5V die Schaltung nicht mehr korrekt funktioniert, wird die Schleife dort abgebrochen.

Mit den über USB ausgegebenen Werten wurden diese Kurven gezeichnet: