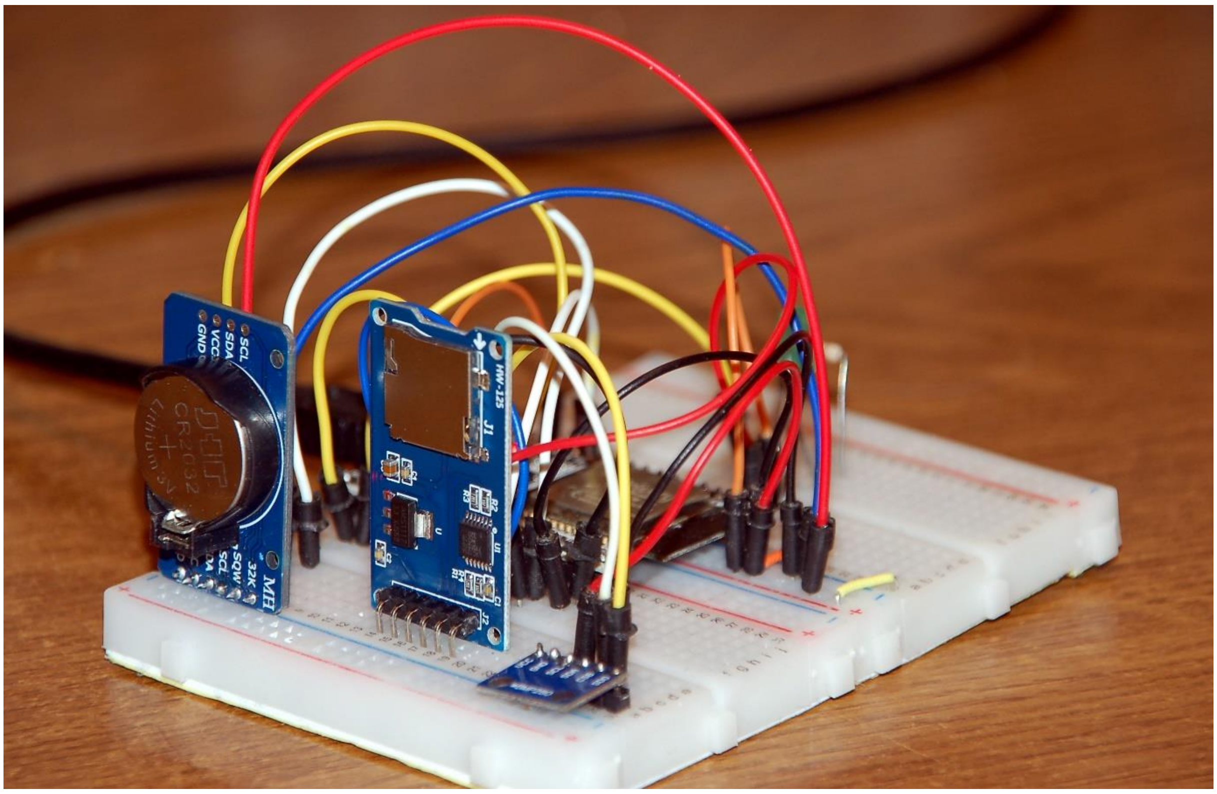


Abbildung 1: Datenlogger mit RTC, SD-Card und BMP280

How were the temperature and air pressure values last month? Perhaps you are also interested in the values of lighting, sunshine duration and relative humidity in the greenhouse. How often did the light in the basement stay overnight and how often did the heating stay on for how long? Questions that are not immediately available to answer. Nevertheless, the results of such long-term measurements are very helpful for a pragmatic solution to certain grievances. What to do if there is no server within reach to which a measurement servant of the ESP32 category could send data. Quite simply, you give the ESP32 the task to save the data in a file. Today's post tells you what options you have and how to do it. Welcome to the blog on the topic

MicroPython and ESP32/ESP8266 as Datalogger

Data loggers are applications that collect data from sensors and store it in a file together with a time stamp. ESP8266 and ESP32 are masters in connecting sensors of all kinds. Various interfaces are available for this, from simple GPIO pins through analog voltage inputs to bus systems such as One Wire, I2C and SPI. The question: "What to do with the data?" is not that easy to answer at first, because an ESP32 is not a PC on which you just open a file to push in the data.

As strange as it may sound, it works the same way on the ESP32 as it does on Windows or Linux or ... The "how" is clarified immediately after the list of required hardware. Despite the three storage options, there are only two assigned parts lists.

Hardware

Dateien im Flash des ESP32/ESP8266
oder im EEPROM auf dem DS3231-Board ablegen:

1	ESP32 LOLIN32 oder D1 Mini NodeMcu mit ESP8266-12F WLAN Modul oder NodeMCU Lua Lolin V3 Module ESP8266 ESP-12F
1	BMP280
1	DS3231-Modul (Real-Time-Clock aka RTC)
1	LED, Farbe egal
1	Widerstand 1k0
1	Netzteil oder Batterie (mit Regler) auf 3,3V oder 5V

Dateien auf einer SD-Speicherkarte ablegen:

1	ESP32 LOLIN32
1	SPI Reader Micro Speicher SD TF Karte
	Rest (ohne Controller) wie oben

MicroPython offers the option of using the flash memory as a carrier for a file system. The methods required for this are very similar to those of the LUA and Linux systems, but should not be completely unknown to Windows fans either. The dialects just have certain differences in syntax. Here as there are commands that allow the handling of storage and read processes and are contained in the basic vocabulary and those that relate more to the operating system (aka Operating System or OS for short) and are used to create, manage and remove directories and files. We shall deal with the first genus today. But first, a little bit of information about the software.

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

Verwendete Firmware:

MicropythonFirmware
Bitte eine Stable-Version aussuchen

MicroPython-Programs for the Project:

24cxx.py
bmp280.py
datalogger.py
datalogger_ds.py
datalogger_eep.py
datalogger_flash.py
datalogger8266.py
datalogger8266_ds.py
ds3231.py
ds3231.py
readEEPROM.py
sdcard.py

MicroPython - Language - Modules and Programs

You can find detailed instructions for installing Thonny here. There is also a description of how the Micropython firmware is burned onto the ESP chip.

MicroPython is an interpreter language. The main difference to the Arduino IDE, where you always and exclusively flash entire programs, is that you only have to flash the MicroPython firmware once at the beginning on the ESP32 before the controller understands MicroPython instructions. You can use Thonny, µPyCraft or esptool.py for this. I have described the process for Thonny here.

As soon as the firmware is flashed, you can have a casual conversation with your controller, test individual commands and immediately see the answer without first having to compile and transfer an entire program. This is exactly what bothers me about the Arduino IDE. You simply save an enormous amount of time if you can do simple tests of the syntax and hardware through to trying out and refining functions and entire program parts via the command line before you knit a program out of it. For this purpose I also like to create small test programs over and over again. As a kind of macro, they combine recurring commands. From such program fragments, entire applications can develop.

Autostart

If the program is to start autonomously when the controller is switched on, copy the program text into a newly created blank file. Save this file under boot.py in the workspace and upload it to the ESP chip. The program starts automatically the next time it is reset or switched on.

Testing programs

If the program is to start autonomously when the controller is switched on, copy the program text into a newly created blank file. Save this file under boot.py in the workspace and upload it to the ESP chip. The program starts automatically the next time it is reset or switched on.

In between, Arduino IDE again?

If you want to use the controller together with the Arduino IDE again later, simply flash the program in the usual way. However, the ESP32 / ESP8266 then forgot that it ever spoke MicroPython. Conversely, every Espressif chip that contains a compiled program from the Arduino IDE or the AT firmware or LUA or ... can easily be provided with the MicroPython firmware. The process is always as described here.

1. Data files on ESP32 / ESP8266?

Because in most cases the flash area of the ESP32 is by no means exhausted by programs, there is (ample) space there for the storage of files. The handling is very simple, we open a file object (aka handle) for writing, appending or reading, write to the file or read from it and close the object again at the end. Handling is as simple as this sentence, as the following program snippet shows. The individual commands can also be entered one after the other via the REPL command line.

```
f=open("testdatei.txt","w")
f.write("Hallo World\n")
f.close()
```

```
  12
```

After this sequence has been carried out, the test file.txt file is located in the root directory of the ESP32 / 8266. MicroPython tells us that 12 bytes have been written. We can convince ourselves of this by performing a refresh on the device directory of the ESP32 in Thonny.

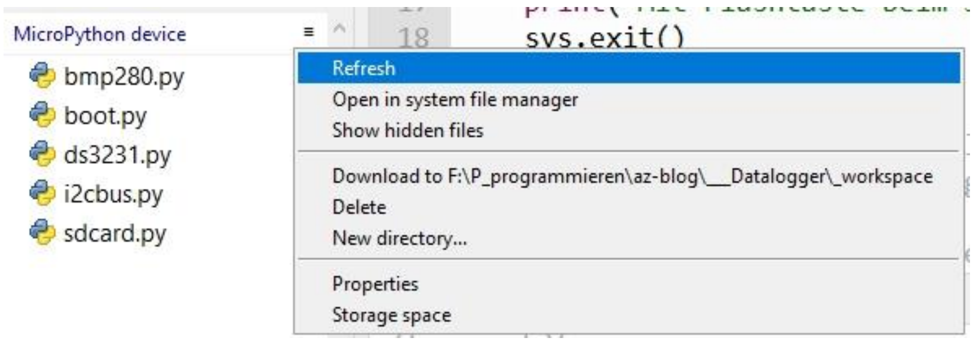


Abbildung 2: Refresh des Device Directories

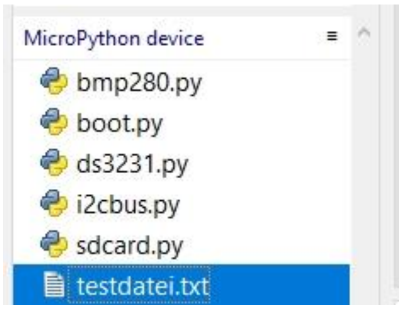
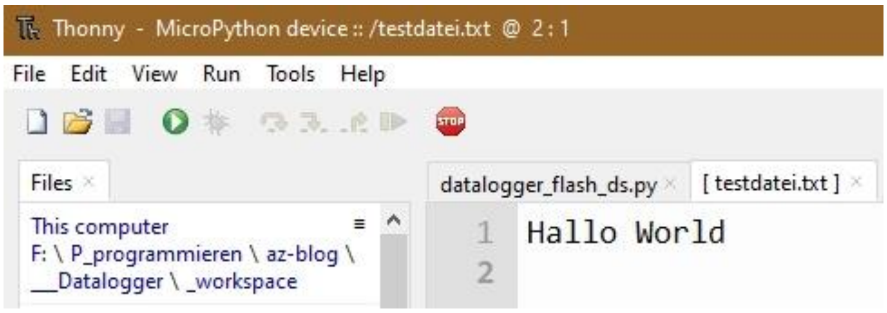


Abbildung 3: Anzeige der Testdatei

We can also open the file by double-clicking the entry.


Thonny - MicroPython device :: /testdatei.txt @ 2 : 1
File
Edit
View
Run
Tools
Help
Files
This computer
F: \ P_programmieren \ az-blog \ __Datalogger \ _workspace
datalogger_flash_ds.py
[testdatei.txt]

```
Hallo World
```

Abbildung 4: Hallo World

Of course, MicroPython can also open the file for reading. The "r" stands for reading, the "w" for (re) writing and the "a" for appending.

```
f=open("testdatei.txt","r")
print(f.readline())
f.close()
```

```
Hallo world
```

Most of the time, our data logger will most likely idle and wait for the next measurement job. In doing so, it draws approx. 45 mA for self-preservation. It is more energy-saving if we at least send the ESP32 to sleep during this time, then it only needs around a third of the current on average compared to the waking state. Together with the periphery, this is approx. 15 mA. The program that enables this operating mode does not have 90 lines.

You need to know the following about deep sleep mode. ESP8266 and ESP32 handle deepsleep mode very differently. I am describing the very simple procedure for the ESP32 here. We'll take a look at what the ESP8266 looks like towards the end of this post.

The deepsleep () method resides in the machine module. So we have to import them. The method requires a parameter that specifies the idle time in milliseconds. As soon as deepsleep () is called, the ESP32 goes to the sofa and cannot be addressed by anyone or anything for the specified time. When the idle time has expired, the ESP32 wakes up and restarts completely, as if it had just been switched on. Of course, this takes a few seconds until all subsystems are running again. We'll take this into account through the value in the variable start time.

So that we don't lock ourselves out completely, we have to install an emergency brake, this is the ESP32's flash button. If the button is pressed and held at the right time, the program aborts and we can access the ESP32 again via REPL. Pressing the RST button does not help, because then the controller starts up as it did after waking up. Without the option to cancel, we would no longer get into the system. The ESP32 signals states to us through a LOW-active LED on pin GPIO2. You can release the button when the LED has gone out.

```
# datalogger_flash_ds.py
# Autor: J. Grzesina
# Rev.: 1.0
# Date: 27.08.2021
# ********************************************************
import esp
esp.osdebug(None)
import os
import gc              # Platz fuer Variablen schaffen
gc.collect()
from machine import Pin,I2C,deepsleep
from time import sleep
from bmp280 import BMP280
import os, sys
from ds3231 import DS3231

taste=Pin(0,Pin.IN,Pin.PULL_UP)
led=Pin(2,Pin.OUT,value=0)
sleep(2)
led.value(1)
if taste.value()==0:
    print("Mit Flashtaste beim Start abgebrochen")
    sys.exit()
# ****************** Filesystem Info   ******************
def df():
    s = os.statvfs('//')
    return ('{0} KB'.format((s[0]*s[3])/1024))
```

```
# ********************** I2C+ Sensors ********************
i2c=I2C(-1,scl=Pin(22),sda=Pin(21))
b=BMP280(i2c)
rtc=DS3231(i2c)
d2="{:0>2}."
h3="{:0>2}:"
y5="{:>4} "
s3="{:0>2};"
p4="{:0>4}\n"
t6="{:> 5.1f};"
Y,M,D,h,m,s,flag=0,0,0,0,0,0,0

i2cDevices=i2c.scan()
print("Found I2C-Devices @:",end="")
for i in i2cDevices:
    print(hex(i),end=";")
    # 0x57: DS3231.EEPROM 32Kb
    # 0x68: DS3231.RTC
    # 0x76: BMP280
print("")
# ********************* MAIN PART **********************
Minutenabstand=10
Startzeit=5800 # ms
duration=1000*60*Minutenabstand - Startzeit
```

```
duration=10000-Startzeit
# ********************* MAIN PART **********************
led.value(0)
Y,M,D,_,h,m,s=rtc.DateTime()
T=b.calcTemperature()
P=int(b.calcPressureNN(465,T))
th=t6.format(T).replace(".",",")
dtwString=d2.format(D)+d2.format(M)+y5.format(Y)+\
          h3.format(h)+h3.format(m)+s3.format(s)+\
          th+p4.format(P)
print(dtwString)
f=open("logging.txt","a")
f.write(dtwString)
f.close()
sleep(3)
led.value(1)
if taste.value()==0:
    print("Mit Flashtaste abgebrochen")
    sys.exit()
deepsleep(duration)
```

So that we can address the BMP280 and the RTC in the DS3231, we need the corresponding driver classes from the modules bmp280.py and ds3231.py. These two files and the i2cbus.py module must be uploaded to the device directory. We create an I2C object and instantiate a BMP280 and a DS3231 object by calling the constructors.

The following are definitions for format strings that guarantee a uniform output of the measurement data. Then we declare the variables for the date, time and measured variables and make sure that all interfaces of the sensors can be addressed. During the test phase, we set the rest time to 10 seconds, after which times between 10 minutes and a few hours are useful.

Then we enter the main part of the program. The hot phase is indicated to the outside by the switched on LED. We get the date, time, temperature and pressure values and use our format strings to compose the output string. The separators are chosen so that the file can later be imported directly as a CSV file. The output takes place in the terminal area of Thonny and in the file logging.txt by appending. The file must be explicitly closed.

If no button is pressed for the next 3 seconds, the ESP32 takes a nap for the set duration. Then the story of the red horse starts all over again, if - and I haven't mentioned that yet - yes, if the lines of the above listing are in the file boot.py in the root directory on the ESP32. Only then can the program run automatically after a restart.

While the LED is on, the ESP32 should not be switched off, because in the worst case the file system, it is a VfsLittle2, could be damaged and the recording of the measurement data would be for naught. If the LED is not lit, you can end the recording of the measured values by switching off. This is useful when bedtime is more than a few seconds.

To read out the log file, Thonny is started on the PC and the ESP32 is connected to the computer via a USB cable. As soon as the LED lights up for the first time, the autostart is aborted with the flash button on the ESP32. This can be necessary several times in a row. Then we can right-click on logging.txt to load the file into the workspace on the PC in order to process it from there.

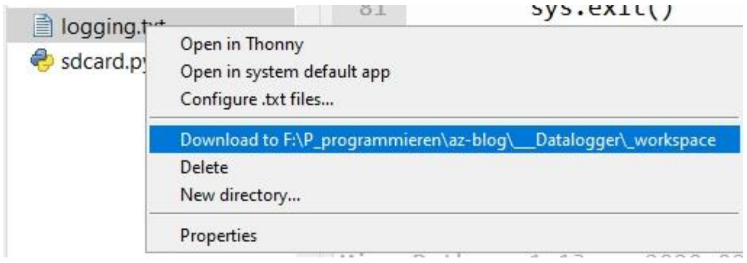


Abbildung 5: Logdatei auslesen

Now I am sure you would like to try it all out. Here is the minimalist circuit for it.

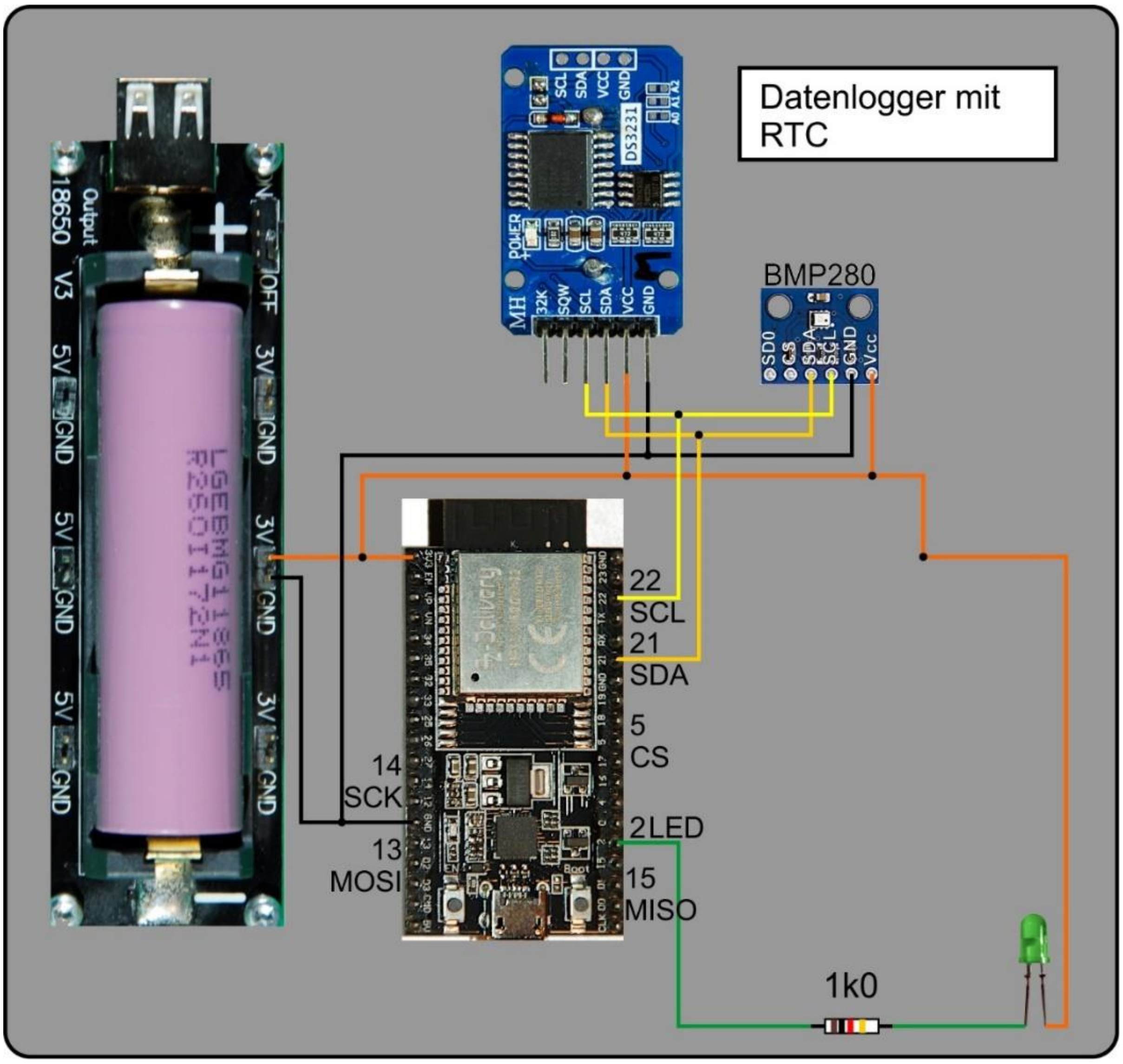


Abbildung 6: Datalogger_Schaltung ohne SD-Card Reader

The function df () returns the available space on the VfsLittle2 file system. Calling os in the terminal when the object inspector is switched on tells us that this is the type of file system involved.

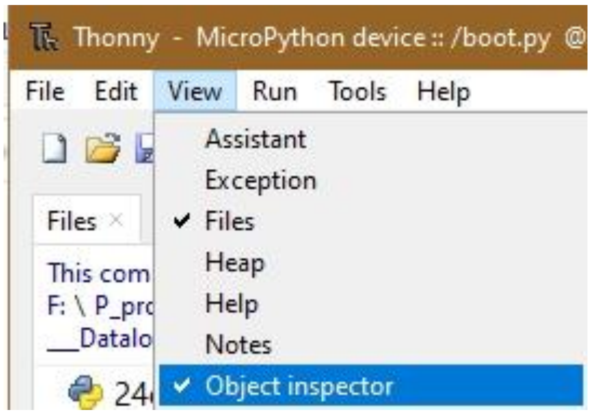
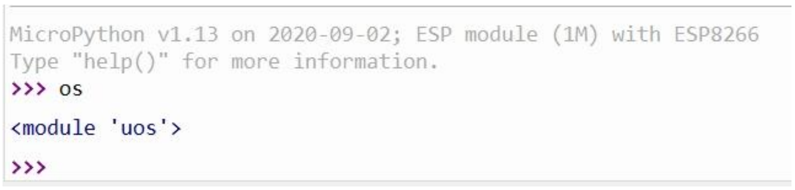
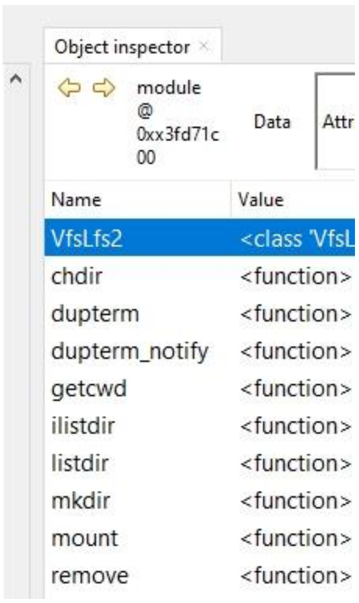


Abbildung 7: Object inspector einschalten



```
MicroPython v1.13 on 2020-09-02; ESP module (1M) with ESP8266
Type "help()" for more information.
>>> os
<module 'uos'>
>>>
```

Abbildung 8: Der Befehl os



Name	Value
VfsLfs2	<class 'VfsL
chdir	<function>
dupterm	<function>
dupterm_notify	<function>
getcwd	<function>
ilistdir	<function>
listdir	<function>
mkdir	<function>
mount	<function>
remove	<function>

Abbildung 9: Filesystem-Info im Objekt inspetor

With a data record length of 31 bytes, we can easily calculate how many data records will fit on the ESP32. Similar to a hard disk, there are logical sectors of 4096 bytes each. The os.statvfs command delivers this value and the number of free blocks, here 854, together with other values in a tuple. This command is also used in the df () function.

```
>>> os.statvfs('//')
(4096, 4096, 866, 854, 854, 0, 0, 0, 0, 255)
```

In Abbildung 9 ist außerdem ersichtlich, wo sich die restlichen Befehle für die Datei- und Verzeichnisbehandlung verbergen, nämlich auch in der Klasse os.

2. Daten speichern in einem externen EEPROM

For the sake of completeness, let's briefly consider data storage in an external EEPROM of the type AT24C32 or similar. As the type designation suggests, it is a non-volatile 32Kb memory. 127 of our data sets fit into this 4KB = 4096Byte. As we have seen above, an ESP32 / ESP8266 offers more than 800 times the storage space. I only mention this storage option because such a memory chip lives on the RTC module with the DS3231. It has the device address 0x57 and can be controlled via the I2C bus using the methods from class ds3231.AT24C32 after a corresponding object has been instantiated. You are welcome to download the sample program datalogger_eep.py and examine it more closely. Because of the limited memory, it is not designed for deep sleep mode. Instead, a timer takes over the time control. The memory address is calculated and assigned via an index (memory location number). When the upper limit of 127 is reached, the acquisition is automatically stopped. This status has been reached when the LED flashes rapidly. Pressing the flash button ends the program run.

The program readEEPROM.py can read out the EEPROM and output it in the terminal window. Via the clipboard, the data is sent to a simple text file for further processing.

The module 24cxx.py provides the class AT24CXX isolated with the same methods as ds3231.AT24C32, but can also control larger EEPROMs with 8KB.

3. Eine SD-Card am ESP32

Enough with the peanuts, let's turn to the storage giant. The SD memory cards allow us a factor of 1000 or 1024 compared to the ESP32. To use them we need a memory card module, a card in mini format (with an adapter for reading on the PC) and the class sdcard.py. I downloaded the original from GitHub and modified it slightly to make it usable for my purposes

It is important to specify the corresponding GPIO pins for the signals SCK, CS, MISO and MOSI. No default values are preset by the constructor. The pin GPIO12 specified in the original for the MISO line could not be used because the ESP32 hung up with every restart. Therefore, the connection was made to GPIO15 as a replacement. I also don't define the SPI interface object in the sdcard.SDCard class, but outside in datalogger.py and then pass it to the constructor of the SDCard class. But let's take a look at the extended circuit first.

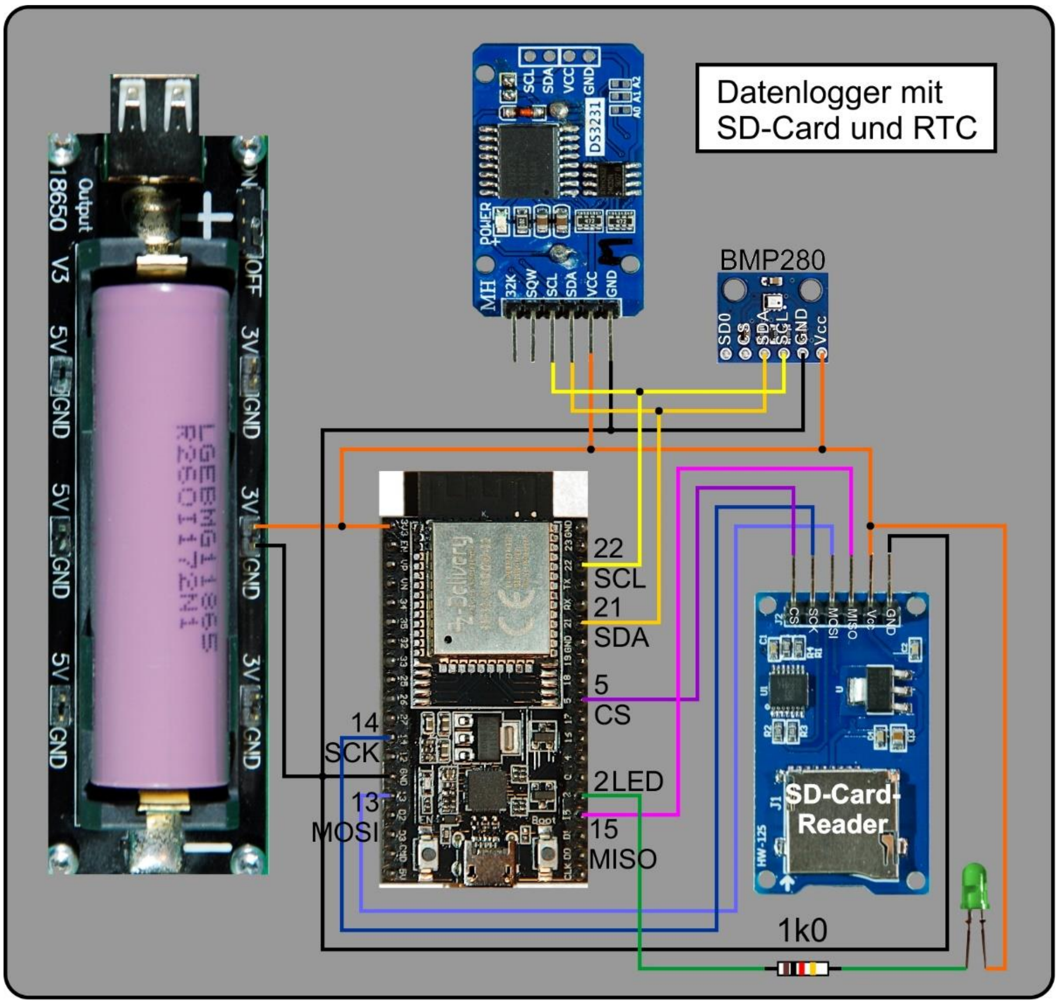


Abbildung 10: Datalogger_Schaltung mit SD-Card Reader

The core of the program remains the same. The preparations are extended to include the import of the SDCard class. This also requires advanced preparations for building the SD file system. Known danger spots are fixed by try-except constructs. As a contrast, I present the program variant with timer control. Therefore, the timer class is imported and after the instantiation of timer 0, it is also configured and started for the continuous run mode.

The timer ISR (aka Interrupt Service Routine) job () takes the mandatory parameter tim, quickly fetches the time data from the RTC and sets the flag to 1 for further processing.

The value of this flag is tested in the main loop and the response is already known by preparing the data for output. The hot area is secured again by the illuminated LED. At any time, however, it is possible to exit without any problems by pressing the flash button. If the LED is bright, it only needs to be pressed until the LED goes out. The while loop runs through very quickly, which is why the program can react quickly to the cancel key. When the program is ended by pressing a button, the file system of the SD card is unmounted so that the card can then be safely removed. A corresponding message appears in Thonny's terminal window. The ESP32 should not be de-energized while the program is running. The current consumption is 45mA when idle and approx. 65mA during the short measuring phase.

The syntax for saving on SD card is the same as when saving in a file in the flash of the ESP32. Only the path specification / sd has to be added.

```
# datalogger.py
# Autor: J. Grzesina
# Rev.: 1.0
# Date: 26.08.2021
# **********************************************************
from machine import Pin,SPI,I2C,Timer
from bmp280 import BMP280
import os, sys, sdcard
from ds3231 import DS3231
taste=Pin(0,Pin.IN,Pin.PULL_UP)
led=(2,Pin.OUT)
# ****************** Create Filesystem ******************
spi=SPI(1,baudrate=100000,sck=Pin(14),mosi=Pin(13),\
        miso=Pin(15),polarity=0,phase=0)
sd = sdcard.SDCard(spi, Pin(5))
try:
    os.mount(sd, '/sd')
    print("SD-Card is mounted on /sd")
except OSError as e:
    print(e)
    print("SD-Card previously mounted")

# ********************* I2C+ Sensors ********************
i2c=I2C(-1,scl=Pin(22),sda=Pin(21))

b=BMP280(i2c)
rtc=DS3231(i2c)
#eep=AT24C32(i2c)
d2="{:0>2}."
h3="{:0>2}:"
y5="{:>4} "
s3="{:0>2};"
p4="{:0>4}\n"
t6="{:> 5.1f};"
Y,M,D,h,m,s,flag=0,0,0,0,0,0,0

i2cDevices=i2c.scan()
print("Found I2C-Devices @:",end="")
for i in i2cDevices:
    print(hex(i),end=";")
    # 0x57: DS3231.EEPROM 32Kb
    # 0x68: DS3231.RTC
    # 0x76: BMP280
print("")

# ********************* Timer Stuff ********************
def job(tim):
    global Y,M,D,h,m,s,flag
    Y,M,D,_,h,m,s=rtc.DateTime()
    flag=1

t=Timer(0)
# Minutenabstand=10
# duration=1000*60*Minutenabstand
duration=5000
t.init(mode=Timer.PERIODIC,period= duration,callback=job)

# ********************** MAIN LOOP ***********************
while 1:
    if flag==1:
        led.value(0)
        T=b.calcTemperature()
        P=int(b.calcPressureNN(465,T))
        th=t6.format(T).replace(".",",")
        dtwString=d2.format(D)+d2.format(M)+y5.format(Y)+\
                  h3.format(h)+h3.format(m)+s3.format(s)+\
                  th+p4.format(P)
        print(dtwString)
        f=open("/sd/logging.txt","a")
        f.write(dtwString)
        f.close()
        sleep(2)
        led.value(1)
        flag=0

    if taste.value()==0:
        t.deinit()
        os.umount("/sd")
        print("Mit Flashtaste abgebrochen\nSD-Card wurde 
ausgehaengt und kann entnommen werden.")
        sys.exit()
```

For longer rest intervals, the program can of course also be written as a deep sleep variant datalogger_ds.py. In addition to saving energy (15mA quiescent current), the advantage is that the card can be removed from the file system after each measurement operation, so that the ESP32 can be switched off at any time when the LED is dark. With the restart everything will be rebooted again. The memory card is also attached to the file system again. The program text must be in the boot.py file again for the system to start automatically.

4. ESP8266 as Datalogger

Yes, the ESP8266 can also be used as a data logger. For the I2C bus, however, the GPIO pins have to be changed. SCL becomes GPIO5 = D1 and SDA comes to GPIO4 = D2. Everything else is the same as with the ESP32 Storage in the flash area. Here you can download the timer-controlled program for the ESP8266.

And the SD card, does it work on the ESP8266? Unfortunately no, for operation with DS3231 and BMP280 together - no chance, because 228 bytes of RAM are missing despite garbage collection. I had the same game a year ago when I tried to run three DS18B20s together with a DS3231 RTC and a TCP server on a D1 Mini. At that time I left out the RTC because the time stamps could be supplied by the Linux client. This is simply not possible with a stand-alone data logger. So let's be satisfied with the file system in the flash area, because measured values without a time stamp are not very meaningful in many cases.

Well, the matter of saving energy in deep sleep remains with the ESP8266. I had already indicated that it works very differently than with the ESP32. In the case of the little brother, only the internal RTC runs in deep sleep and no extra timer. Second, the IRQ that wakes the controller out of its slumber cannot do this internally, but only via the GPIO16 pin, which is to be connected to the RST pin. The IRQ sets the GPIO16 (= D0) to GND potential and thus triggers a hardware reset.

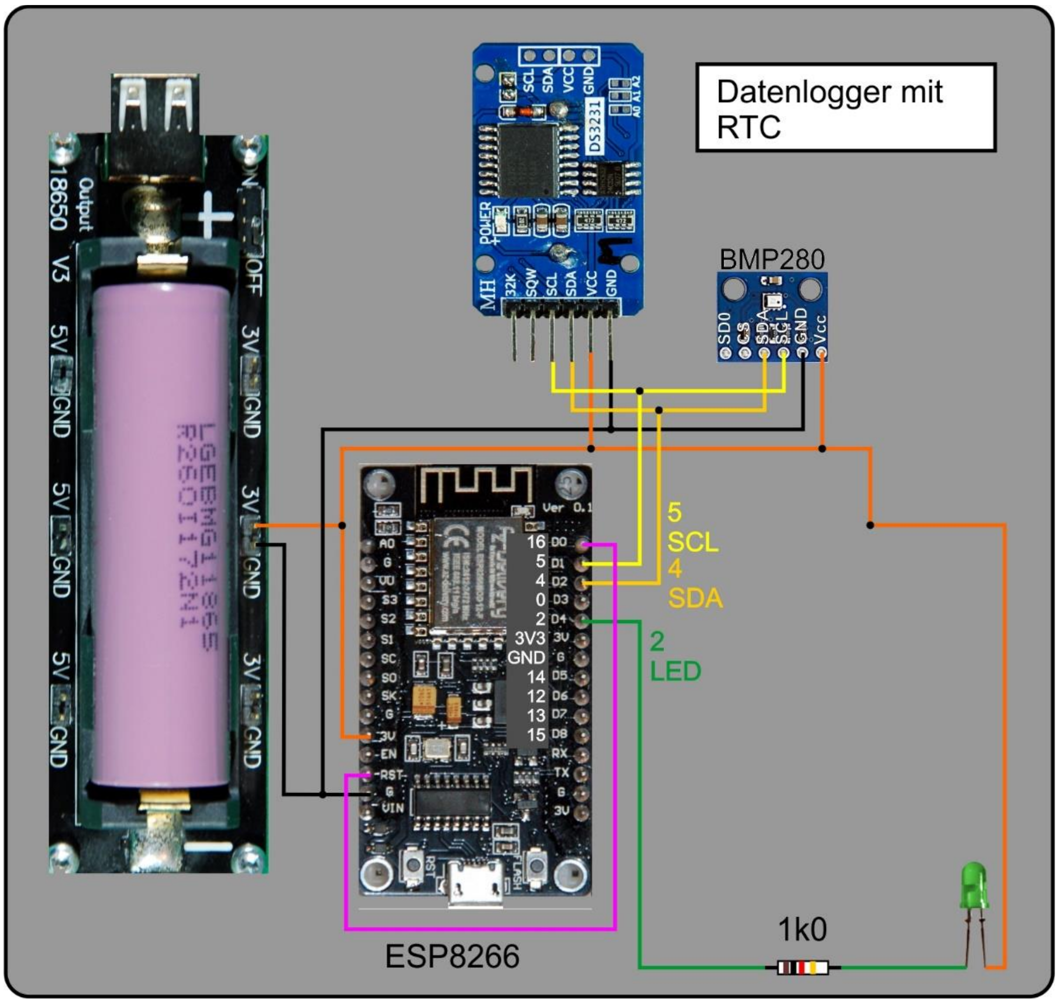


Abbildung 11: Datalogger_Schaltung mit ESP8266

And here is the program for it. For testing purposes it runs with a period of approx. 10 seconds.

```
# boot.py as datalogger8266_ds.py
# Autor: J. Grzesina
# Rev.: 1.0
# Date: 28.08.2021
# ***********************************************************
import esp
esp.osdebug(None)
import os,sys
import gc               # Platz fuer Variablen schaffen
gc.collect()
from machine import Pin,I2C,Timer,RTC,DEEPSLEEP,deepsleep
from time import sleep
from bmp280 import BMP280
from ds3231 import DS3231
# Pintranslator fuer ESP8266-Boards
# LUA-Pins     D0 D1 D2 D3 D4 D5 D6 D7 D8
# ESP8266 Pins 16  5  4  0  2 14 12 13 15
#                 SC SD
taste=Pin(0,Pin.IN,Pin.PULL_UP)
led=Pin(2,Pin.OUT,value=0)
sleep(2)
led.value(1)
if taste.value()==0:
    print("Mit Flashtaste beim Start abgebrochen")
    sys.exit()

# ************* On Board RTC configurieren ****************
dsl = RTC()
dsl.irq(trigger=dsl.ALARM0, wake=DEEPSLEEP)
# ******************* Filesystem Info   *******************
def df():
    s = os.statvfs('//sd')
    return ('{0} KB'.format((s[0]*s[3])/1024))
# ********************** I2C+ Sensors ********************
i2c=I2C(-1,scl=Pin(5),sda=Pin(4))
b=BMP280(i2c)
rtc=DS3231(i2c)
d2="{:0>2}."
h3="{:0>2}:"
y5="{:>4} "
s3="{:0>2};"
p4="{:0>4}\n"
t6="{:> 5.1f};"
Y,M,D,h,m,s,flag=0,0,0,0,0,0,0
i2cDevices=i2c.scan()
print("Found I2C-Devices @:",end="")
for i in i2cDevices:
    print(hex(i),end=";")
    # 0x57: DS3231.EEPROM 32Kb
    # 0x68: DS3231.RTC
    # 0x76: BMP280
print("")
# ********************* Timer Stuff ********************
Minutenabstand=10
StartUpTime=4300 #ms
duration=1000*60*Minutenabstand-StartUpTime
duration=10000 - StartUpTime
dsl.alarm(dsl.ALARM0, duration)
print("Freier Speicher:",df())
# ********************* MAIN LOOP **********************
led.value(0)
Y,M,D,_,h,m,s=rtc.DateTime()
T=b.calcTemperature()
P=int(b.calcPressureNN(465,T))
th=t6.format(T).replace(".",",")
dtwString=d2.format(D)+d2.format(M)+y5.format(Y)+\
          h3.format(h)+h3.format(m)+s3.format(s)+\
          th+p4.format(P)
print(dtwString, end="")
f=open("logging.txt","a")
f.write(dtwString)
f.close()
flag=0
sleep(3)
led.value(1)
if taste.value()==0:
    t.deinit()
    print("Mit Flashtaste abgebrochen")
    print("Freier Speicher:",df())
    sys.exit()
deepsleep()
```

So now nothing stands in the way of our long-term measurement series. With a lithium cell with 3600 mAh as in the circuit diagrams, we can collect data for over a week if we select deep sleep mode. If we assume 4 measurements per hour, the sum total in 7 days is 672 ridiculous data sets. And if MicroPython and the ESP32 / ESP8266 don't lie about the free, available memory size, then we can measure according to this scheme for 162 weeks. Only - certainly no battery charge lasts for three years.

I wish you lots of fun trying it out and good measurement results!