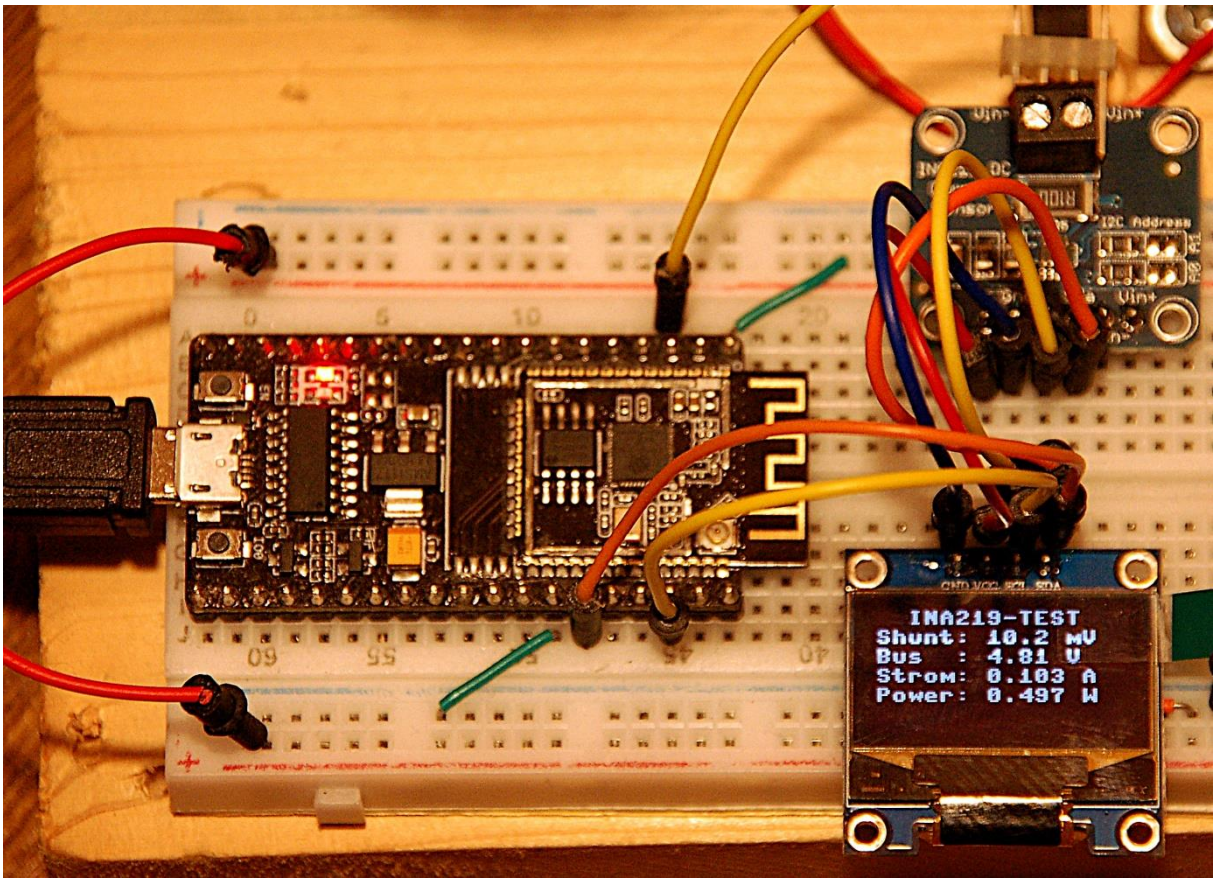

INA219 im Test

Diesen Beitrag gibt es auch als PDF-Datei. Er befasst sich mit dem Erstellen eines MicroPython-Moduls für den Baustein INA219.

Das IC INA219 ist ein Multitalent, wenn es um die Erfassung von Spannung, Stromstärke und Leistung geht. Diese Größen müssen erfasst werden, um den MPP, den Maximum-Power-Point, also die Stelle größter Leistung eines Solarpanels zu orten. Der INA219 arbeitet im Hintergrund und liefert die drei Werte über den I2C-Bus. Zur Ansteuerung des ICs werde ich Ihnen in diesem Beitrag ein MicroPython-Modul vorstellen, und dessen Anwendung anhand eines Beispielprogramms demonstrieren. Willkommen also zu einer neuen Folge aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Der ESP32 und das Modul ina219a

Zum Testen wird die Schaltung des Buckconverters aus dem letzten Beitrag verwendet. In die Liste der Hardware reiht sich daher nur ein INA219 zusätzlich ein. Die Eckdaten aus dem Datasheet:

Betriebsspannung	3,3V bis 5V
Leerlauf-Stromstärke	0,7mA
Busspannung	-26V bis 26V
Stromstärke	-3,2A bis +3,2A

Das IC verwende ich in Form eines Break-Out-Boards (aka BOB), auf dem alle nötigen weiteren Bauelemente bereits vorhanden sind. Die Links dazu findet man reichlich mit Hilfe von Onkel Google.

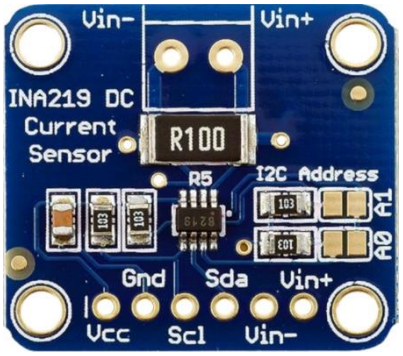

Abbildung 1: Das Modul INA219

Aus der Tabelle kann man schon erkennen, dass das Modul bidirektional arbeitet. Positive Werte für Strom und Spannung ergeben sich dann, wenn der Pluspol der Busspannung an den Anschluss Vin+ gelegt wird. Der Minuspol der Busspannung liegt aber nicht an Vin-, sondern an GND, wie die Versorgungsspannung auch. Unter Busspannung ist auch nicht der I2C-Bus gemeint, sondern eben die Plusleitung der Eingangsspannung. Die Schaltskizze macht das deutlich.

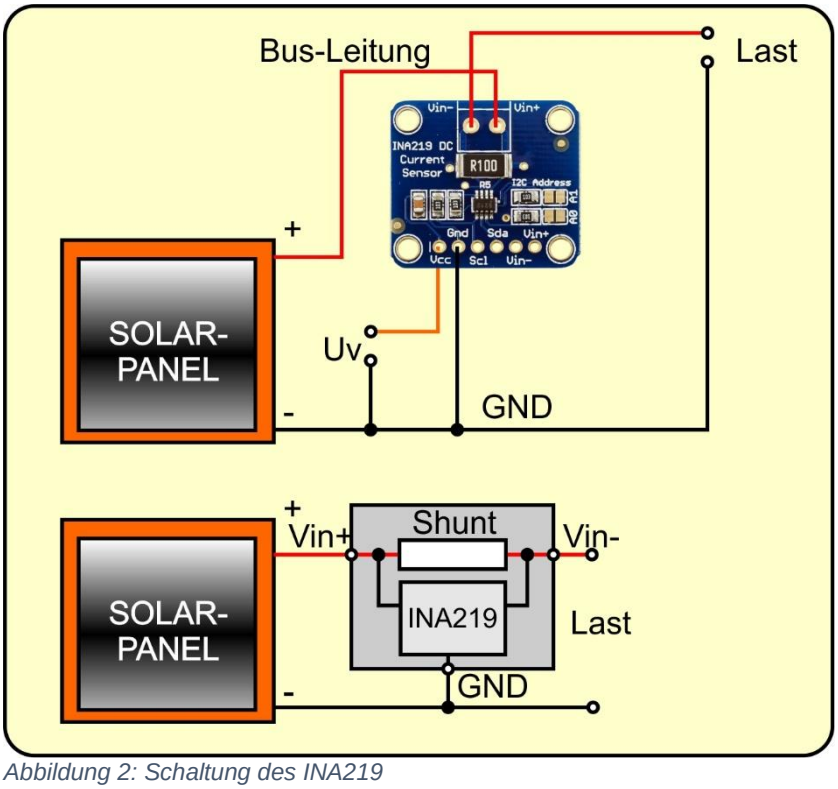

Abbildung 2: Schaltung des INA219

Das Messprinzip für die Stromstärke beruht auf der Messung des Spannungsabfalls am Shunt-Widerstand, der einen typischen Wert von 100mΩ hat. Als Spannungsquelle ist hier ein Solarpanel eingezeichnet. Der Messwiderstand liegt also in der Busleitung zwischen dem Pluspol der Spannungsquelle und dem Pluspol der Last. Die Stromstärke durch den Shunt darf bis zu 3,2A betragen. Der Spannungsabfall beträgt dann 320mV, die verbratene Leistung ca. 1W.

Der Wert für die Busspannung wird an Vin- gegen GND gemessen. Den Spannungswert an der Spannungsquelle erhält man durch Addition der Busspannung an Vin- mit dem Spannungsabfall am Shunt. Beide Werte können über entsprechende Register des ICs abgefragt werden. Den Zusammenhang zeigt die Abbildung 3, die aus dem Datenblatt, Seite 10, stammt.

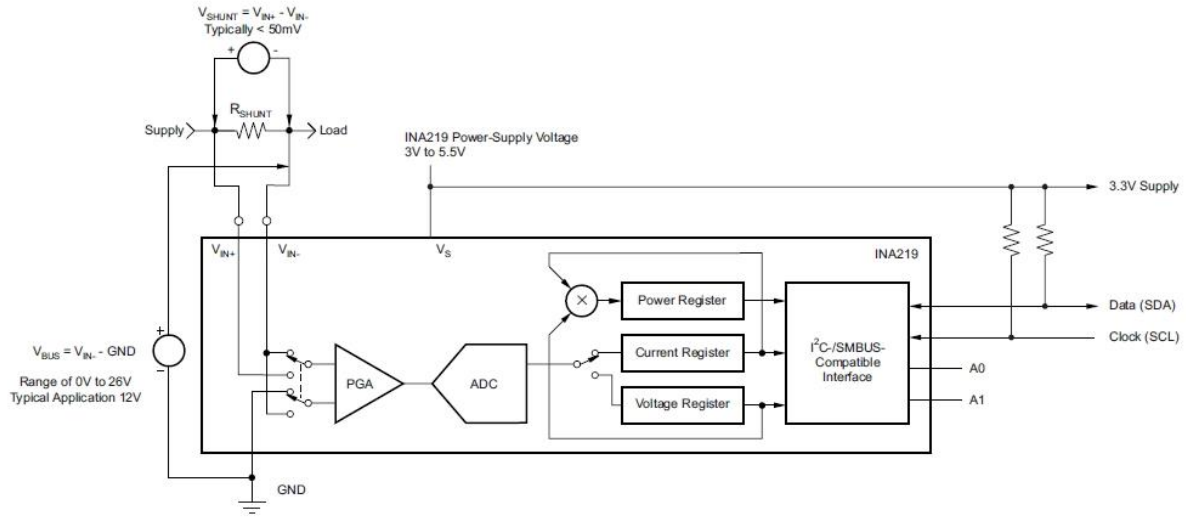

Abbildung 3: Blockschaltbild des INA219

Hardware

1	ESP32 Dev Kit C unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board oder NodeMCU-ESP-32S-Kit
1	0,96 Zoll OLED SSD1306 Display I2C 128 x 64 Pixel
1	INA219
2	Widerstand 47 Ω
diverse	Jumperkabel
1	MB-102 Breadboard Steckbrett mit 830 Kontakten 3er

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

Verwendete Firmware für den ESP8266/ESP32:

v1.19.1 (2022-06-18) .bin

Die MicroPython-Programme zum Projekt:

ssd1306.py Hardwaretreiber zum OLED-Display
oled.py API für das OLED-Display
ina219.py Treiber-Modul für den INA219

MicroPython - Sprache - Module und Programme

Zur Installation von Thonny finden Sie hier eine ausführliche Anleitung (english version). Darin gibt es auch eine Beschreibung, wie die Micropython-Firmware (Stand 05.02.2022) auf den ESP-Chip gebrannt wird.

MicroPython ist eine Interpretersprache. Der Hauptunterschied zur Arduino-IDE, wo Sie stets und ausschließlich ganze Programme flashen, ist der, dass Sie die MicroPython-Firmware nur einmal zu Beginn auf den ESP32 flashen müssen, damit der Controller MicroPython-Anweisungen versteht. Sie können dazu Thonny, µPyCraft oder esptool.py benutzen. Für Thonny habe ich den Vorgang hier beschrieben.

Sobald die Firmware geflasht ist, können Sie sich zwanglos mit Ihrem Controller im Zwiegespräch unterhalten, einzelne Befehle testen und sofort die Antwort sehen, ohne vorher ein ganzes Programm kompilieren und übertragen zu müssen. Genau das stört mich nämlich an der Arduino-IDE. Man spart einfach enorm Zeit, wenn man einfache Tests der Syntax und der Hardware bis hin zum Ausprobieren und Verfeinern von Funktionen und ganzen Programmteilen über die Kommandozeile vorab prüfen kann, bevor man ein Programm daraus strickt. Zu diesem Zweck erstelle ich auch gerne immer wieder kleine Testprogramme. Als eine Art Makro fassen sie wiederkehrende Befehle zusammen. Aus solchen Programmfragmenten entwickeln sich dann mitunter ganze Anwendungen.

Autostart

Soll das Programm autonom mit dem Einschalten des Controllers starten, kopieren Sie den Programmtext in eine neu angelegte Blankodatei. Speichern Sie diese Datei unter boot.py im Workspace ab und laden Sie sie zum ESP-Chip hoch. Beim nächsten Reset oder Einschalten startet das Programm automatisch.

Programme testen

Manuell werden Programme aus dem aktuellen Editorfenster in der Thonny-IDE über die Taste F5 gestartet. Das geht schneller als der Mausklick auf den Startbutton, oder über das Menü **Run**. Lediglich die im Programm verwendeten Module müssen sich im Flash des ESP32 befinden.

Zwischendurch doch mal wieder Arduino-IDE?

Sollten Sie den Controller später wieder zusammen mit der Arduino-IDE verwenden wollen, flashen Sie das Programm einfach in gewohnter Weise. Allerdings hat der ESP32/ESP8266 dann vergessen, dass er jemals MicroPython gesprochen hat. Umgekehrt kann jeder Espressif-Chip, der ein kompiliertes Programm aus der Arduino-IDE oder die AT-Firmware oder LUA oder … enthält, problemlos mit der MicroPython-Firmware versehen werden. Der Vorgang ist immer so, wie hier beschrieben.

Signale auf dem I2C-Bus

Wie eine Übertragung auf dem I2C-Bus abläuft und wie die Signalfolge ausschaut, das können Sie in meinem Beitrag mammutmatrix_2_ger.pdf nachlesen. Ich verwende dort ein interessantes kleines Tool, mit dem Sie die I2C-Bus-Signale auf Ihren PC holen und analysieren können.

Das Modul ina219.py

Wie bereits erwähnt, wird das INA219 BOB über I2C angesprochen. Weil ich neben diesem Modul auch ein OLED-Display am I2C-Bus betreibe, habe ich das MicroPython-Modul so geschrieben, dass dem Konstruktor eines INA219-Objekts eine vorab zu definierende I2C-Instanz zu übergeben ist, die dann auch für die Anzeige Verwendung findet. Programme, die beide Features bedienen sollen, starten also immer folgendermaßen.

```
from machine import Pin,SoftI2C
i2c=SoftI2C(Pin(22),Pin(21))
```

Im Modul ina219.py definiere ich eine Klasse INA219, die, der besseren Lesbarkeit wegen, mit der Festlegung diverser Konstanten und Variablen beginnt, damit die Register und Bits mit Namen angesprochen werden können, anstatt mit Nummern. Am Anfang steht die 7-Bit-Basis-Hardware-Adresse (HWADR) des INA219, 0x40 oder 64 dezimal. An diese Adresse wird durch die I2C-Treiberroutinen ein achtes Bit als LSBit (Least Significant Bit = niederwertigstes Bit) angehängt, 0 für Schreib- und 1 für Lesevorgänge. Laut Datenblatt könnte man den INA219-Chip auf eine von 15 weiteren Hardwareadressen einstellen. Allerdings ist die 0x40 auf dem Modul fest verdrahtet.

```
class INA219:
    HWADR=const(0x40)
```

Von den sechs Registern des INA219 dienen vier der Abfrage von Shuntspannung, Busspannung, Stromstärke und Leistung. Das Register 0x00 enthält die Konfigurationsbits des Bausteins. Hier werden im Wesentlichen der Betriebsmodus, die Auflösung für die Messung der Busspannung und der Spannung am Shunt sowie der Faktor für den Messbereich der Shunt-Spannungsmessung eingestellt. Ferner kann einer von zwei Bereichen für die Busspannung angegeben werden. Das MSBit

des Registers ist das Reset-Bit, das den Chip in denselben Zustand wie nach dem Booten versetzt, wenn es auf 1 gesetzt wird.

```
    # Register
    RegConfig=const(0x00)
    RegUShunt=const(0x01)
    RegUBus=const(0x02)
    RegPower=const(0x03)
    RegCurrent=const(0x04)
    RegCalibrate=const(0x05)

    # RST-Bit
    RST=const(1<<15)

    # Umax am Bus
    Ubus16V=const(0)
    Ubus32V=const(1<<13)
    ubus=["16V","32V"]
```

Die Liste **ubus** enthält die Klartextbezeichnungen für Rückmeldungen des Konfigurationszustands.

```
    # Abschwaecher
    PGA1=const(0) # 40mV
    PGA2=const(1) # 80mV
    PGA4=const(2) # 160mV
    PGA8=const(3) # 320mV
    PGAShift=const(11)
    PGAMask = const(3<<11)
    pga=["PGA1: 40mV","PGA2: 80mV","PGA4: 160mV","PGA8:
320mV",]
```

Die Konfigurationsgruppe für den Abschwächer besteht aus einer Reihe von Bits, die zusammen die Werte 0 bis 3 ergeben. **PGAShift** gibt an, um wie viele Stellen dieser Wert nach links geschoben werden muss. PGAMask dient zum maskieren der Bits in dem 16-Bit Registerwert. Analog verhält sich das auch für die Auflösung und den Modus.

```
    # Aufloesung
    Res9=const(0)  # 9-Bit @ 84µs
    Res10=const(1) # 10-Bit @ 148µs
    Res11=const(2) # 11-Bit @ 276µs
    Res12=const(3) # 12-Bit @ 532µs
    Samp2=const(9) # 12-Bit 2x @ 1060µs
    Samp4=const(10) # 12-Bit 4x @ 2,13ms
    Samp8=const(11) # 12-Bit 8x @ 4,26ms
    Samp16=const(12) # 12-Bit 16x @ 8,51ms
    Samp32=const(13) # 12-Bit 32x @ 17,02ms
    Samp64=const(14) # 12-Bit 64x @ 34,05ms
```

```
    Samp128=const(15) # 12-Bit 128x @ 68,10ms
    BusShift=const(7)
    ShuntShift=const(3)
    BusMask=const(15<<7)
    ShuntMask=const(15<<3)
    res={
        0 :"Res9: 9-Bit @ 84µs",
        1 :"Res10: 10-Bit @ 148us",
        2 :"Res11: 11-Bit @ 276us",
        3 :"Res12: 12-Bit @ 532us",
        9 :"Samp2: 12-Bit 2x @ 1060us",
        10 :"Samp4: 12-Bit 4x @ 2,13ms",
        11 :"Samp8: 12-Bit 8x @ 4,26ms",
        12 :"Samp16: 12-Bit 16x @ 8,51ms",
        13 :"Samp32: 12-Bit 32x @ 17,02ms",
        14 :"Samp64: 12-Bit 64x @ 34,05ms",
        15 :"Samp128: 12-Bit 128x @ 68,10ms",
        }
```

Die Klartextmeldungen sind bei der Auflösung und dem Modus als Dictionary (Dict) kodiert, weil nicht alle Bitmuster durchgängig vorkommen.

```
    ModePowerDown=const(0)
    ModeADCoff=const(4)
    ModeUShunt=const(5)
    ModeUBus=const(6)
    ModeBoth=const(7)
    ModeMask=const(7)
    mode={
        0:"Power down",
        4:"ADC off",
        5:"Shunt-Spannung",
        6:"Bus-Spannung",
        7:"Bus- und Shunt-Spannung",
        }

    CnvRdy=const(1)
    Ovfl=const(0)
```

Das Register 0x02 enthält neben dem Wert der Busspannung die Statusbits 0 und 1. Letzteres ist 1, wenn eine Messung fertig ist, Bit 0 zeigt einen Überlauf der Strommessung an.

Die Methode **__init__()** stellt den Konstruktor einer INA219-Instanz dar. Bis auf den Positionsparameter **i2c**, dem als Argument ein I2C-Objekt übergeben werden muss, sind alle weiteren Parameter Schlüsselwortparameter und damit optional und können bei der Instanziierung auch weggelassen werden. In diesem Fall wird der jeweilige Vorgabewert verwendet.

```
    def __init__(self,
                 i2c,
                 mode=ModeBoth,
                 busres=Samp4,
                 shuntres=Samp4,
                 shuntpga=PGA1,
                 ubus= Ubus16V,
                 Imax=0.4,
                 Rshunt=0.1):
        self.i2c=i2c
        print("Constructor of INA219")
        self.imax=Imax
        self.rshunt=Rshunt
        self.configure(mode=mode,
                       busres=busres,
                       shuntres=shuntres,
                       shuntpga=shuntpga,
                       ubus=ubus,
                       Imax=Imax,
                       Rshunt=Rshunt)
```

Damit während des Programmlaufs die Konfiguration geändert werden kann, gibt es die Methode **configure**(), die auch der Konstruktor abschließend aufruft.

Vom Konstruktor aus werden für alle Parameter Argumente übergeben. Das ist wichtig, damit auch alle Properties (aka Eigenschaften oder Attribute) der INA219-Instanz gesetzt werden können.

Andererseits ist es lästig, wenn man zur Änderung nur eines Parameters die ganze Liste von Argumenten übergeben muss. Daher habe ich alle diese Parameter optional kodiert und mit None vorbelegt. Jetzt kann ich in der Methode **configure**() darauf abfragen und nur auf wirklich vorhandene Parameter reagieren.

Als Erstes wird die bestehende Konfiguration eingelesen und die lokale Variable **aenderung** auf **False** gesetzt. Für jedes übergebene Argument ist der entsprechende Parameter nicht None, somit wird die zuständige Methode ausgeführt, welche die gewünschte Änderung der Eigenschaft durchführt.

```
    def configure(self,
                  mode=None,
                  busres=None,
                  shuntres=None,
                  shuntpga=None,
                  ubus=None,
                  Imax=None,
                  Rshunt=None):
        self.readConfig()
        aenderung=False
        if ubus is not None:
            self.setBusrange(ubus)
        if shuntpga is not None:
            self.setPGAShunt(shuntpga)
```

```
        if busres is not None:
            self.setBusResolution(busres)
        if shuntres is not None:
            self.setShuntResolution(shuntres)
        if mode is not None:
            self.setMode(mode)
        if Imax is not None:
            self.imax=Imax
            aenderung=True
        if Rshunt is not None:
            self.rshunt=Rshunt
            aenderung=True
        if aenderung:
            self.currentLSB=self.imax/32768
            self.cal=int(0.04096/  \
            (self.currentLSB*self.rshunt))
            self.writeReg(RegCalibrate,self.cal)
            print("Kalibrierfaktor: {}".format(self.cal))
        print("Konfigurationsbits:")
        self.printData(self.config)
```

Weil sowohl eine Änderung von **Rshunt** als auch **Imax** eine Änderung des Kalibrierfaktors bewirkt, wird neben dem Eigenschaftswert **self.rshunt** oder **self.imax** auch **aenderung** auf True gesetzt. Das führt zu einer Neuberechnung des Kalibrierfaktors. Zur Kontrolle werden die Konfigurationsbits im Terminal ausgegeben.

Zum Auslesen eines Registers dient die Methode **readReg**(), der die Registernummer übergeben wird. Alle Register haben 16-Bit Breite. Daher müssen 2 Bytes eingelesen und anschließend zu einem 16-Bit-Wert zusammengesetzt werden.

```
    def readReg(self, regnum):
        buf=bytearray(2)
        buf[0]=regnum
        self.i2c.writeto(HWADR,buf[0:1])
        buf=self.i2c.readfrom(HWADR,2)
        return buf[0]<<8 | buf[1]
```

Die Kommunikation über den I2C-Bus erfolgt mittels Variablen, die das sogenannte Buffer-Protokoll erfüllen. Deshalb verwende ich hier das Bytearray **buf** zum Senden der Registeradresse und zum Empfang des Registerwerts. Der Integerwert der Registernummer wird zunächst an die Position 0 des Bytearrays geschrieben. Dann sende ich die nullte Position des Buffers zum INA219. Dieser sendet die beiden Bytes des Registerinhalts in der Reihenfolge MSB, LSB, welche ich demselben Bytearray zuweise. Das MSB in Position 0 schiebe ich um 8 Bit nach links, das entspricht einer Multiplikation mit 256. Die unteren 8 Bit dieses Werts wurden beim Schieben mit Nullen aufgefüllt. Dazu oderiere ich das LSB in Bufferposition 1 und erhalte so den Registerinhalt, den die Methode zurückgibt.

Das Gegenstück zu **readReg**() ist **writeReg**(). Der Methode werden Registernummer und Registerinhalt übergeben. Damit wird das Bytearray **buf** gefüllt. Das LSB des

Registerinhalts wird als letztes gesendet. Ich bekomme es durch Undieren des Werts mit 0xFF = 0b11111111. Das MSB bekomme ich durch Rechtsschieben um 8 Bitpositionen, was einer Division durch 256 entspricht. Die Registernummer kommt in Bufferposition 0, weil sie vor den Daten gesendet werden muss.

```
    def writeReg(self, regnum, data):
        buf=bytearray(3)
        buf[2]=data & 0xff #
        buf[1]=data >> 8   # HIGH-Byte first (big endian)
        buf[0]=regnum
        self.i2c.writeto(HWADR,buf)
```

Alle anderen Methoden des Moduls greifen zur Kommunikation mit dem INA219 auf **readReg**() und **writeReg**() zurück.

Nach den Regeln der OOP (**O**bjekt **O**rientierte **P**rogrammierung) soll auf die Attribute eines Objekts nicht direkt lesend oder gar schreibend von außen zugegriffen werden. Daher gibt es in der Klasse INA219 entsprechende Routinen, welche Attributwerte auslesen oder setzen. Ersteres sieht dann so aus.

```
    def readConfig(self):
        self.config = self.readReg(RegConfig)
```

```
    def getPGAShunt(self):
        self.readConfig()
        c=(self.config & PGAMask) >> PGAShift
        return c
```

Die **getXYZ**()-Methoden lesen zuerst das Konfigurationsregister vom INA219 ein. Der Wert landet in dem Attribut **self.config**. Der Gruppenwert ergibt sich durch Undieren der Konfiguration mit der Gruppenmaske (PGAMask) und anschließendem Rechtschieben um den Positionswert (PGAShift). Der Gruppenwert wird zurückgegeben. Er dient auch als Index in die Liste oder in das Dictionary mit den Klartextmeldungen. Diese werden durch die **tellXYZ**()-Methoden zurückgegeben.

```
    def tellPGA(self,val):
        return self.pga[val]
```

getXYZ()- und **tellXYZ**()-Methoden gibt es für jede der Gruppen PGA, Busspannungsbereich, Busspannungsauflösung, Shuntspannungsauflösung und Mode.

Eine Zusammenfassung der gesamten Konfiguration gibt **tellConfig**() im Terminal aus.

```
    def tellConfig(self):
        print("Spannungsbereich:",
              self.tellUbus(self.getBusrange()))
        print("PGA Abschwaechung:",
              self.tellPGA(self.getPGAShunt()))
        print("Bus Aufloesung:",
              self.tellResolution(self.getBusResolution()))
        print("Shunt Aufloesung:",
              self.tellResolution(self.getShuntResolution()))
        print("Modus:",self.tellMode(self.getMode()))
```

Jeder **getXYZ**()-Methode entspricht auch eine **setXYZ**()-Methode. Sie überprüft das übergebene Argument auf Gültigkeit und wirft eine **Assertation-Exception**, falls ein ungültiger Wert übergeben wurde.

```
    def setBusrange(self,data=Ubus16V):
        assert data in [0,1,16,32]
        c=self.config & (0xFFFF - Ubus32V)
        if data in [1,32]:
            c=c | Ubus32V
        self.config=c
        self.writeConfig()
```

Im Attribut **self.config** muss zunächst das Bit für den Busspannungsbereich gelöscht werden. Dafür benötige ich die negierte Konstante Ubus32V=const(1<<13). Da aber in MicroPython kein Befehl dafür vorhanden ist, muss ich in die Trickkiste greifen.

```
   Ubus32V  = 0b0010000000000000
  ~Ubus32V  = 0b1101111111111111

    0xFFFF  = 0b1111111111111111
 - Ubus32V  = 0b0010000000000000
= ~Ubus32V  = 0b1101111111111111
```

Falls als Argument 1 oder Ubus32V übergeben wurde wird das Bereichsbit durch Oderieren mit Ubus32V gesetzt, dem Attribut **self.config** zugewiesen und zum INA219 gesendet.

```
    def writeConfig(self):
        self.writeReg(RegConfig,self.config)
```

Die anderen **setXYZ**()-Methoden arbeiten ähnlich. Allerdings müssen dort mehrere Bits gesetzt und an die richtige Position geschoben werden. Bei einem durchgehenden Bereich kann man die gültigen Werte mit **range**() überprüfen. Lückende Bereiche prüfe ich mit Hilfe einer Liste.

```
    def setPGAShunt(self,data=PGA1):
        assert data in range(0,4)
        self.readConfig()
        c=self.config & (0xFFFF - PGAMask)
        c=c | (data << PGAShift)
        self.config=c
        self.writeConfig()
```

```
    def setBusResolution(self,data=Samp4):
        assert data in [0,1,2,3,9,10,11,12,13,14,15]
        self.readConfig()
        c=self.config & (0xFFFF - BusMask)
        c=c | (data << BusShift)
        self.config=c
        self.writeConfig()
```

Das Rücksetzen der Gruppenbits geschieht wieder durch Differenzbildung. Der entstandene Wert wird mit den Konfigurationsbits oderiert nachdem diese an die korrekte Position geschoben wurden.

Bisher ging es um die Konfiguration des INA219. Jetzt wollen wir die Messwerte auslesen und aufarbeiten. Ich beginne mit dem Spannungsabfall über dem Messwiderstand von 100mΩ.

```
    def getShuntVoltage(self):
        raw=self.readReg(RegUShunt)
        if raw & 1<<15:
            raw = -(65536 - raw)
        return raw / 100 # mV
```

Ich hole den Wert des Registers 0x01, das den Rohwert der Shuntspannung in der Zweierkomplement-Darstellung enthält. Ist Bit 15 gesetzt, dann handelt es sich um einen negativen Wert, der einer speziellen Behandlung bedarf. Von 0xFFFF wird in diesem Fall der eingelesene Wert abgezogen und 1 addiert, 0xFFFF+1 = 0x10000 = 65536. Das vorangestellte Minuszeichen macht daraus den korrekten negativen Spannungswert. Dessen LSBit entspricht laut Datenblatt 10µV. Teile ich den Wert vor der Rückgabe durch 100, dann erhalte ich die Shuntspannung in Millivolt.

```
    def getBusVoltage(self):
        return (self.readReg(RegUBus) >>3) * 4 / 1000# V
```

Vom Inhalt des Registers 0x02 brauchen wir die Bits 3 bis 15, die erst einmal um 3 Bitpositionen nach rechts geschoben werden müssen. Das LSBit hat nun den Wert von 4mV. Der Rückgabewert der Methode ist mit der Division durch 1000 die Busspannung in Volt.

Der Wert für die Stromstärke ist nur brauchbar, wenn es bei der Messung und Berechnung im INA219 keinen Überlauffehler gegeben hat. Deshalb lese ich erst

einmal das Busspannungsregister und isoliere Bit 0, das Überlaufflag **Ovfl**. Ist es gesetzt, ist der Stromwert ohne Bedeutung und es wird eine Überlauf-Exception geworfen, die das aufrufende Programm abfangen sollte. Ist Ovfl = 0, dann muss der Inhalt des Stromstärkeregisters mit dem LSB für die Stromstärke multipliziert werden. Der Wert für das LSB wurde in der Methode **configure**() berechnet. Die Formel dafür findet man im Datenblatt:

currentLSB = **maximal zu erwartende Stromstärke / 32768**.

Dieser Wert wird ebenfalls für eine weitere Größe, den Kalibrierfaktor **cal**, benötigt. Auch dafür findet man eine Formel im Datenblatt (Seite 12).

Cal = 0,04096 / (currentLSB * rshunt)

Für das Feintuning wird jetzt die Stromstärke mit einem Amperemeter (A_Meterwert, zum Beispiel 91,7 mA) gemessen und zugleich die Methode **getCurrent**() (INA219_Wert zum Beispiel 90,69824 mA) aufgerufen.

```
            self.currentLSB=self.imax/32768
            self.cal=int(0.04096/  \
                     (self.currentLSB*self.rshunt)
            self.writeReg(RegCalibrate,self.cal)
```

Der Kalibrierfaktor **cal** wird mit Hilfe dieser Werte korrigiert und in der Methode **configure**() wie folgt eingebunden.

Cal = 0,04096 / (currentLSB * rshunt) * A_Meterwert / INA219_Wert

```
            self.currentLSB=self.imax/32768
            self.cal=int(0.04096/  \
                     (self.currentLSB*self.rshunt)*\
                        91.7/90.69824)
            self.writeReg(RegCalibrate,self.cal)
```

Nun sollte bei weiteren Messungen der A_Meterwert mit dem INA219_Wert übereinstimmen.

Jetzt fehlt nur noch die Ermittlung der Leistung. Hier kommt die Methode **getPower**(), welche diese Lücke schließt.

```
    def getPower(self):
        if not (self.readReg(RegUBus) & (1<<Ovfl)):
            return self.readReg(RegPower)*20*self.currentLSB
        else:
            raise OverflowError
```

Auch in diesem Fall wird das Überlaufflag geprüft. Ist es nicht gesetzt, kann der Wert für die Leistung berechnet werden. Der Inhalt des Power-Registers 0x03 wird mit dem 20-fachen Wert des **currentLSB**s multipliziert, so will es das Datenblatt.

Wie sagte schon Giovanni Trapattoni 1998? "Ich habe fertig". Sie kennen jetzt das Modul **ina219**.py, und es wird Zeit, damit einen Test zu starten. Das Programm **ina219_test.py** wird die Ergebnisse der vier markanten Größen Shunt-Spannung, Bus-Spannung, Stromstärke und Leistung abfragen und im OLED-Display darstellen. Im Terminal wird zu Beginn die Konfiguration im Klartext ausgegeben. Natürlich brauchen wir dazu eine Testschaltung. Hier ist das Schaltbild.

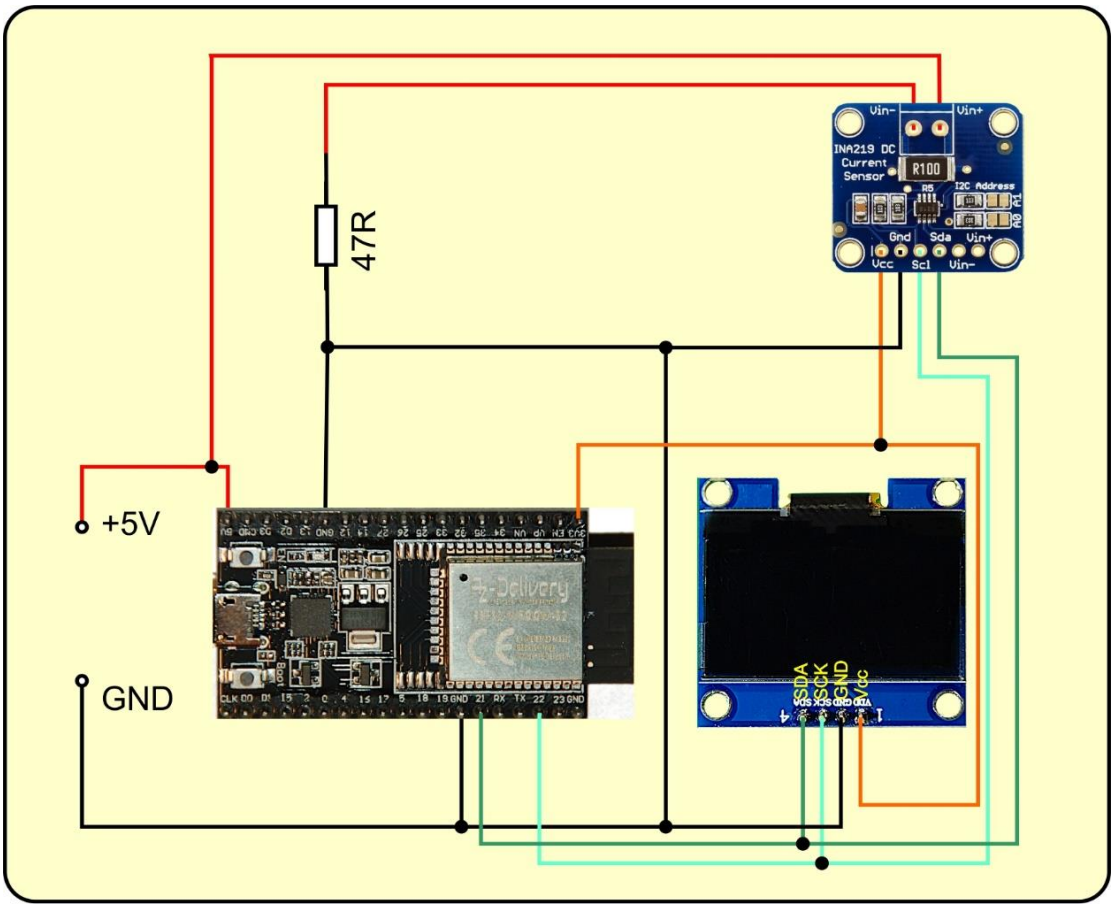

Abbildung 4: INA219 - Testschaltung

Und so schaut die Umsetzung auf dem Testboard aus.

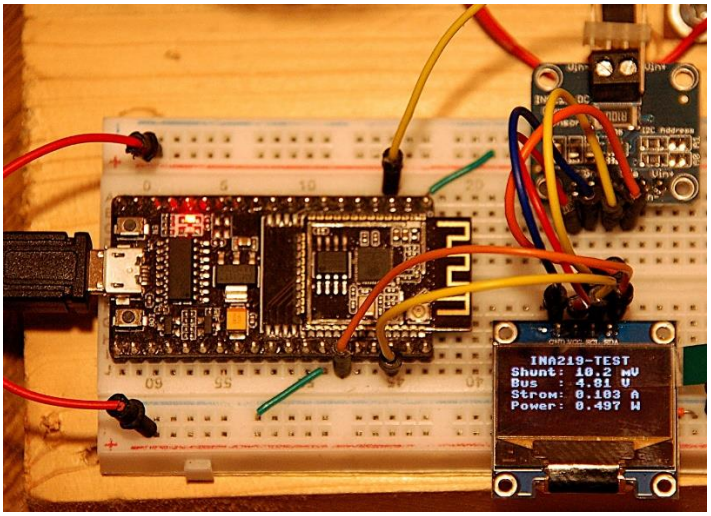

Abbildung 5: INA219 im Test

Nun zum Testprogramm. Es ist sehr überschaubar, sodass man nicht viel darüber erzählen muss. Mit der Flashtaste auf dem ESP32-Board kann man das Programm, das in einer Endlosschleife läuft, beenden. Die Taste schließt nach GND, deshalb wird auf 0 abgefragt. Um bei verschiedenen Stromstärken zu messen, können die beiden 47Ohm-Widerstände einzeln, in Serie und parallel in die Schaltung eingebaut werden. Natürlich lässt sich auch die Schaltung des Buck-Konverters aus der vorherigen Blogfolge verwenden, um zum Beispiel bei verschiedenen Spannungen zu messen.

```
# ina219_test.py
#
from machine import SoftI2C,Pin
from time import sleep
from oled import OLED
from ina219 import INA219
from sys import exit

i2c=SoftI2C(Pin(22),Pin(21))

d=OLED(i2c)
d.clearAll()
d.setContrast(255)
d.writeAt("INA219-TEST",2,0)

shunt = 0.1  # Ohm
imax=0.8 # Ampere

ina=INA219(i2c,mode=INA219.ModeBoth,
           busres=INA219.Samp4,
           shuntres=INA219.Samp4,
           shuntpga=INA219.PGA2,
           ubus= INA219.Ubus16V,
           Imax=imax,
           Rshunt=shunt)

taste=Pin(0,Pin.IN,Pin.PULL_UP)
sleep(0.1)
ina.tellConfig()

while 1:
    shuntSpannung=ina.getShuntVoltage()
    busSpannung=ina.getBusVoltage()
    strom=ina.getCurrent()
    leistung=ina.getPower()
    d.writeAt("Shunt: {:.1f} mV  ".format(shuntSpannung),0,1)
    d.writeAt("Bus  : {:.2f} V  ".format(busSpannung),0,2)
    d.writeAt("Strom: {:.3f} A  ".format(strom),0,3)
    d.writeAt("Power: {:.3f} W  ".format(leistung),0,4)
    sleep(1)
    if taste.value()==0:
        d.writeAt("PROG CANCELLED",1,5)
        exit()
```

Ausblick

Mit dem vom ESP32 gesteuerten Step-Down-Wandler vom letzten Beitrag und der Verwendung eines INA219 als Messknecht sind wir nun in der Lage, automatisch die Leistungskennlinie einer elektrischen Energiequelle zu erfassen. In der nächsten Folge tun wir das für eine Solarzelle. Sie können dann die Parallel- und Serienschaltung mehrerer Solar-Panele unter die Lupe nehmen und bei verschiedener Bestrahlungsstärke untersuchen. Bis dann!