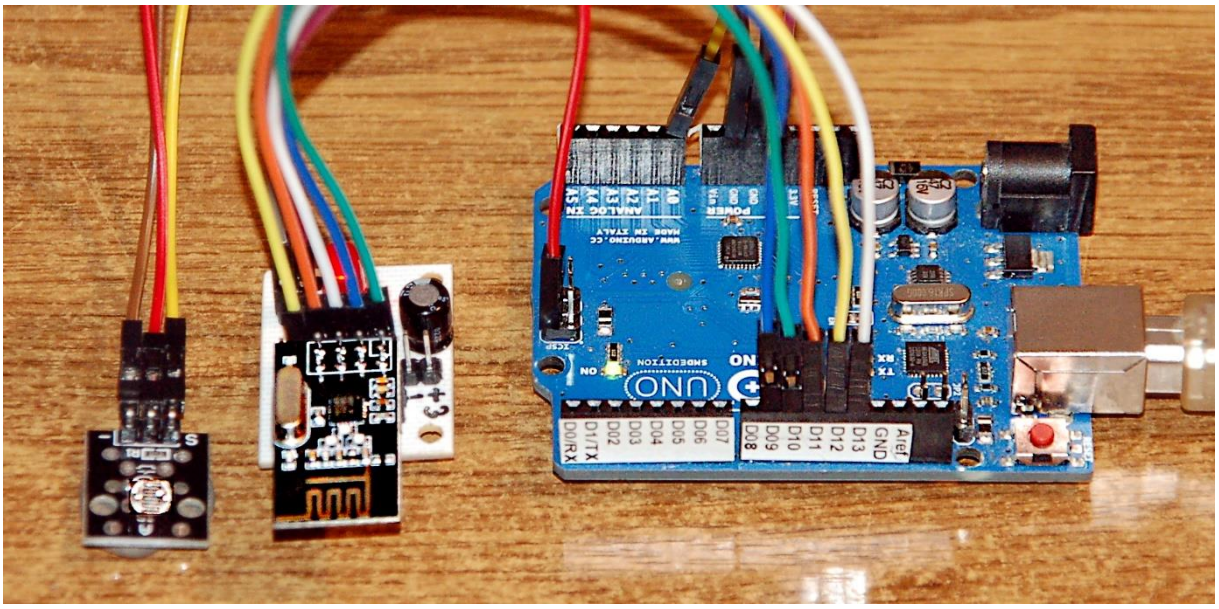

Abbildung 1: LDR, nRF24L01 und Arduino UNO

Diesen Beitrag gibt es auch als PDF-Datei zum Download.

Willkommen zu einem weiteren Beitrag zur Reihe

MicroPython auf dem ESP32 und ESP8266

Vermutlich fragen Sie sich jetzt: "Oh, dann ist sicher das falsche Titelbild hier reingerutscht?" Aber – nein - das Titelbild ist schon richtig hier, es hat nur eine unvollständige Bildunterschrift, man ergänzt besser "**go ESP8266**".

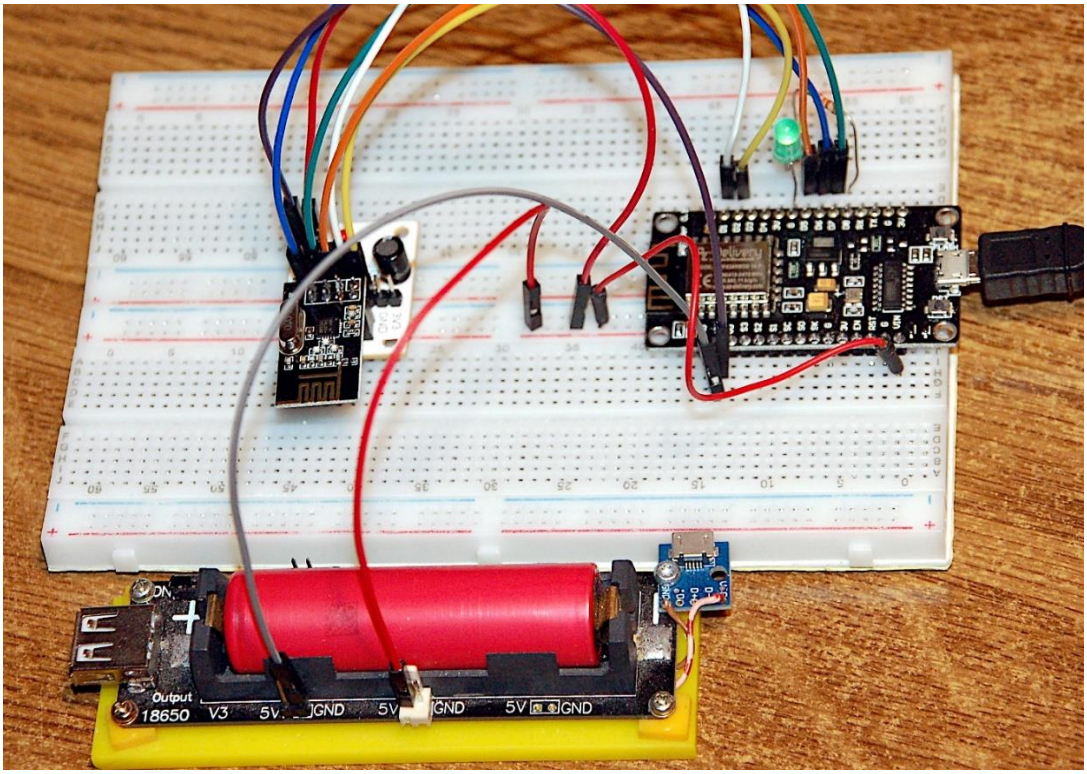

Abbildung 2: nRF24L01 mit ESP8266-12F

Der Titel der aktuellen Episode lautet nämlich:

Arduino goes ESP8266/32 (Teil 1)

Ein trautes Zusammenleben von Controllern der AVR-Familie, wie ATTinyXX, ATMega328 alias Arduino UNO mit deren vielfältigen weiteren Verwandten einerseits und dem Clan der Espressif-Familie, ESP8266 und ESP32, andererseits ist sehr gut möglich. Selbstverständlich klappt die angestrebte Funk-Verbindung auch jeweils familienintern. Natürlich bedarf es, wenn es um Funkverbindungen gehen soll, bei den Arduinos einer kleinen Nachhilfe und die heißt nRF24L01+.

Ein Vorteil des Arduino-Clans ist zum Beispiel dessen einfache Programmierung in Assembler, was zu kleinen, schlanken Programmen führt. AVR-Controller lassen sich auch ganz ohne Arduino-Hardwareumgebung einfach programmieren. Der gravierende Nachteil ist die fehlende WLAN-Fähigkeit. Deshalb soll es in diesem Beitrag darum gehen, wie man einen Arduino-Clone per Funk an einen ESP8266 oder ESP32 andocken kann.

Unter Arduino-Clone verstehe ich die gesamte AVR-Familie, deren Mitglieder eines gemeinsam haben, den SPI-Bus. Weit verbreitet ist die Programmierung eines AVR-Controllers über die Arduino-IDE und den RS232-Port der AVRs. Dann ist jeder dieser Chips aber auch noch mit einer SPI-Schnittstelle ausgerüstet. Über diese Schnittstelle erhält der Controller die Programmierung mit einem Bootloader, wodurch er überhaupt erst in die Lage versetzt wird, auf eine Programmierung via RS232 zu reagieren. Natürlich lässt sich diese SPI-Schnittstelle auch anderweitig einsetzen, zum Beispiel für einen Anschluss an einem ESP32 oder ESP8266. Hierin unterscheidet sich ein AVR-Controller ganz wesentlich von einem ESP-Chip, der seine Daten und Programme grundsätzlich über die USB/RS232-Verbindung erhält.

Ein ESP8266 oder ESP32 besitzt zwei solcher SPI-Interfaces. Eines zur Interaktion mit dem onBoard-EEPROM und eine zweite zur Anbindung von externen SPI-Bausteinen. Ein AVR-Controller benötigt das SPI-Interface nur während der Programmierung über diese Schnittstelle, danach ist sie frei verfügbar. In diesem Projekt machen wir uns das SPI-Interface beider Systeme zu Nutze.

Man könnte also einfach das SPI-Interface zur Kommunikation zwischen Arduino-Clones und einem ESP-Controller direkt nutzen. Dafür wäre dann allerdings eine Kabelverbindung nötig. Vielleicht komme ich in einem Folgebeitrag darauf zurück.

Heute werde ich den ersten Teil eines Projekts vorstellen, das diese Schnittstelle bei beiden Controllern, Arduino und ESP8266, verwendet, um eine Verbindung via einer anderen Funk-Einheit herzustellen, deren Reichweite von unterschiedlichen Quellen zwischen 250m und 1000m angegeben wird. Die Rede ist von einem Modul nRF24L10+, das ebenfalls auf dem Frequenzband von 2,4GHz arbeitet, wie unser bekanntes WLAN. Arduino und ESP8266 werden sich also kabellos über Funk verständigen.

Dabei liegt keines der üblichen Übertragungsprotokolle wie TCP/IP oder UDP zugrunde. Wir müssen uns also großenteils selbst, falls erforderlich, um die Integrität der Daten kümmern.

Im heutigen Beitrag schauen wir uns das nRF24L01+ aus dem Blickwinkel von MicroPython näher an. Wir werden auszugsweise das MicroPython-Modul mit der Klasse NRF24L01 unter die Lupe nehmen und ein Programm entwickeln, auf dessen Grundlage sich zwei ESP8266/32, am WLAN und den Protokollen TCP und UDP vorbei, untereinander verständigen können.

Grundlagen

Das nRF24L01+ - Modul, welches hier zum Einsatz kommt, ist über den SPI-Bus ansprechbar und verfügt darüber hinaus über zwei weitere Steuerleitungen, CE und CSN. Der Anschluss IRQ wird nicht verwendet.

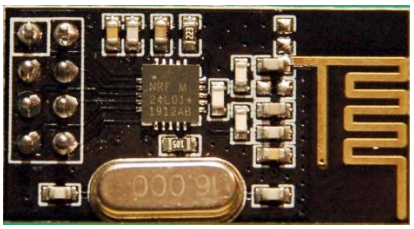

Abbildung 3: nRF24L01

Die Form der Antenne auf dem Board erinnert an den ESP8266-01. Tatsächlich verwendet der nRF24L01 dasselbe Frequenzband auf 2,4GHz. Das kann leider auch zu gegenseitigen Störungen führen, doch dazu später mehr.

Die Schaltung für einen ESP8266 zeigt die folgende Abbildung. Vier digitale Pins und der analoge Eingang bleiben noch verfügbar. Die Pinbezeichnungen auf dem Board des ESP8266 D1 mini sind an der Arduino-IDE orientiert. Die Pinnummern für die Verwendung unter MicroPython wurden, grau unterlegt, hinzugefügt.

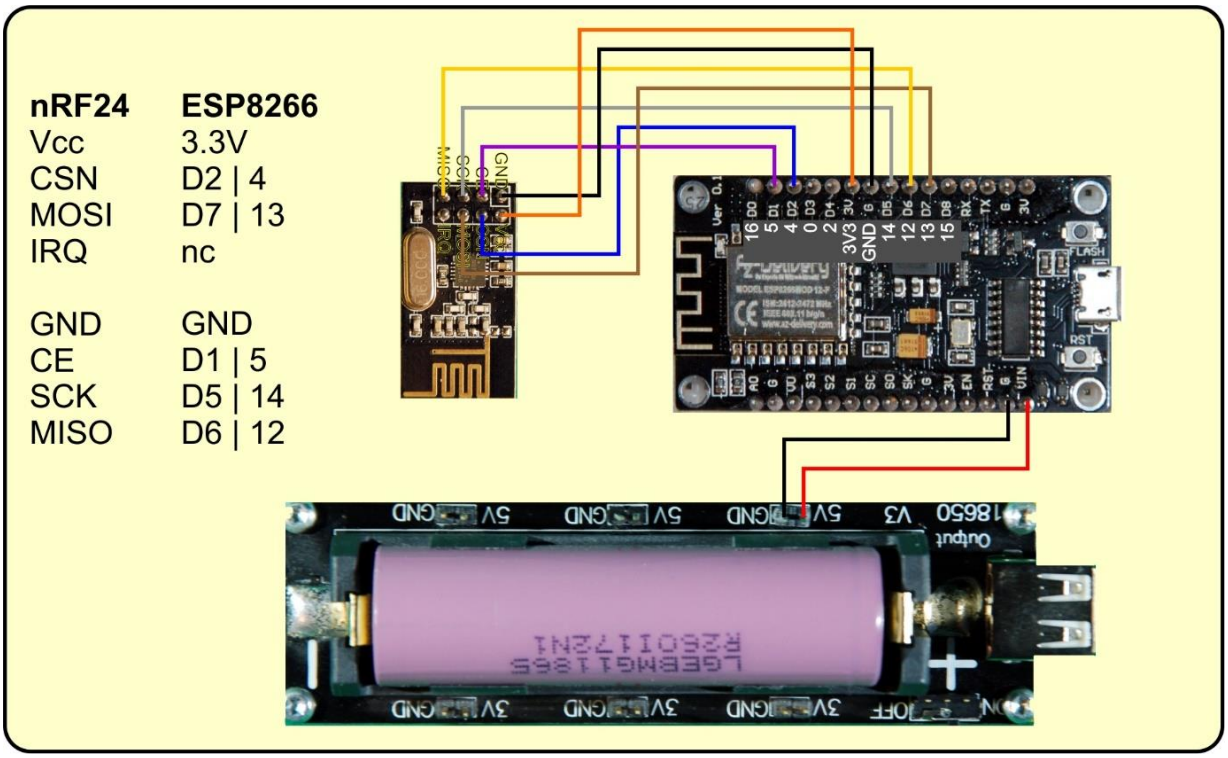

Abbildung 4: nRF24L01 am ESP8266

Natürlich kann auch ein ESP32 verwendet werden. Dessen Anschlüsse sind dann wie folgt. CE und CSN sind beim ESP32 wie auch beim ESP8266 mit den Pins 5 und 4 verbunden. Im Programm sieht das zum Beispiel so aus.

MISO= Pin(15)
MOSI= Pin(13)
SCK = Pin(14)
CSN = Pin(4, mode=Pin.OUT, value=1)
CE   = Pin(5, mode=Pin.OUT, value=0)

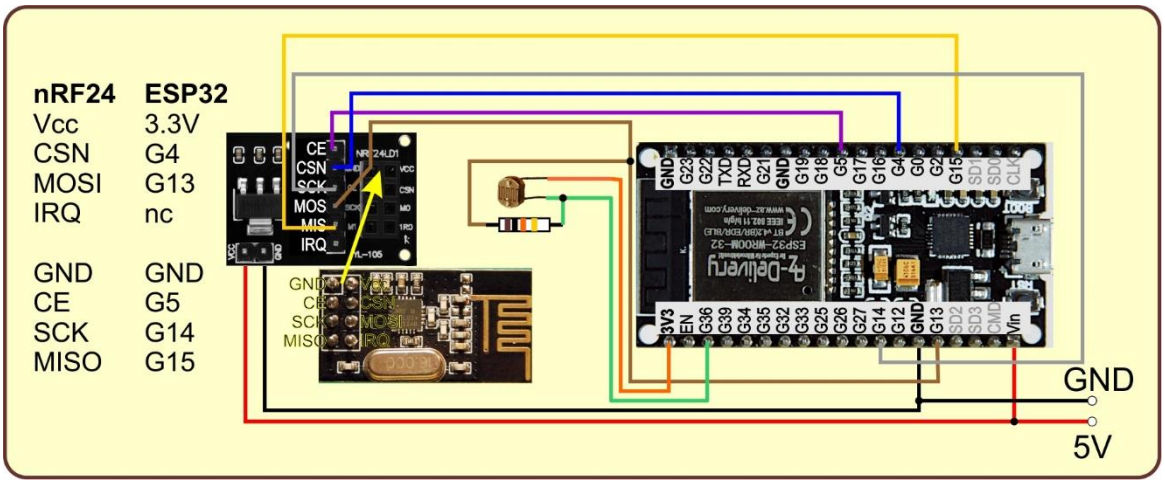

Abbildung 5: nRF24L01 mit Adapter am ESP32 und losem LDR

Das **nRF24L01-Board darf nur mit einer Spannung von maximal 3,3V** betrieben werden, obwohl die Logikleitungen 5V-verträglich sind. Für das Board gibt es einen Breadboardadapter mit integriertem 3,3V-Spannungsregler (AMS1117 3V3), der dann allerdings mit 5V zu versorgen ist (Abbildung 5).

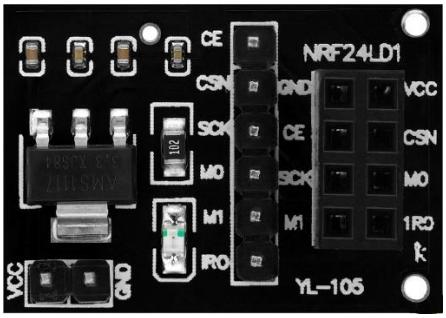

Abbildung 6: nRF24L10_Adapter

Für diesen Fall ist im Schaltplan mit dem ESP8266 (Abbildung 4) eine externe 5V-Versorgung eingezeichnet, die dann auch mit dem **Breadboardadapter** des nRF24L01 zu verbinden ist, aber **niemals direkt mit dem Vcc-Pin des nRF24L01!**

Der abgebildete ESP8266 Node-MCU V3 ist anschlusstechnisch ein Sonderfall unter den ESP-Boards. Bei diesem Board ist die 5V-Zuführung des USB-Kabels nämlich nicht am Pin **Vin** verfügbar, bei den anderen Boards der Familie schon. Für den **Adapter** des nRF24L01 ist deshalb zwingend eine externe Spannungsquelle erforderlich, bei den anderen ESP8266-Boards kann die Versorgung des nRF24L01-Adapters über den Pin **Vcc** durch den USB-Anschluss erfolgen.

Ohne den Breadboardadapter kann der nRF24L01 natürlich jederzeit direkt aus dem 3,3-V-Pin des ESP8266/ESP32 versorgt werden, wie es in den Abbildung 4 zu sehen ist. Abbildung 5 zeigt das Anschlussschema mit Adapter.

Auf die Funktionsweise und Programmierung des nRF24L01+ komme ich später zurück. Beschäftigen wir uns zunächst mit der, im Projekt verwendeten Hardware. Aus den Schaltbildern sind die benötigten Teile für dieses Projekt schnell ersichtlich.

Hardware

1	ESP8266 Node-MCU
1	ESP32 Node-MCU oder ESP32 D1 mini
2	nRF24L01+
2	nRF24L10+ Breadboard-Adapter
1	LDR mit 10kΩ Trimmpoti oder
1	LDR-Modul
2	Minibreadboards
diverse	Jumperkabel
2	passende USB-Kabel
1	Batterie 4,5V oder 5V-Steckernetzteil

Die Verdrahtung der beiden Baugruppen mit dem Arduino und dem ESP8266 stellt sicher kein Problem dar. Den Aufbau des Lichtsensors mit dem LDR schauen wir uns aber doch noch etwas näher an. Die beiden Varianten unterscheiden sich in der Schaltung und im eingesetzten Widerstand.

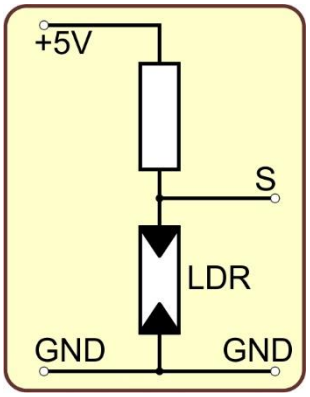

Abbildung 7: Schaltung des LDR-Moduls

Abbildung 8: LDR-Modul

Beim LDR-Modul liegt der Fotowiderstand gegen Masse (GND), der Festwiderstand an der positiven Versorgungsspannung. Beide Widerstände bilden einen sogenannten Spannungsteiler. Bei Belichtung des LDR sinkt dessen Widerstandswert, wodurch die Spannung am Signalausgang S abnimmt. Wir erhalten also umso weniger an Ausgangsspannung, je stärker der LDR beleuchtet wird. Dadurch sinkt auch der Wert des AD-Wandlers beim ESP8266 oder beim Arduino, den wir als Messwertaufnehmer verwenden wollen. Ich beschreibe hier übrigens den Einsatz eines LDR als Sensor, weil die Umsetzung sehr einfach ist. Natürlich können auch beliebige andere Sensoren zum Beispiel am I2C-Bus zur Verwendung

kommen. In der Hauptsache geht es in diesem Beitrag ja um das Zusammenspiel zwischen nRF24L01 und ESP8266.

Der Aufbau mit LDR und Trimm-Potentiometer arbeitet genau anders herum.

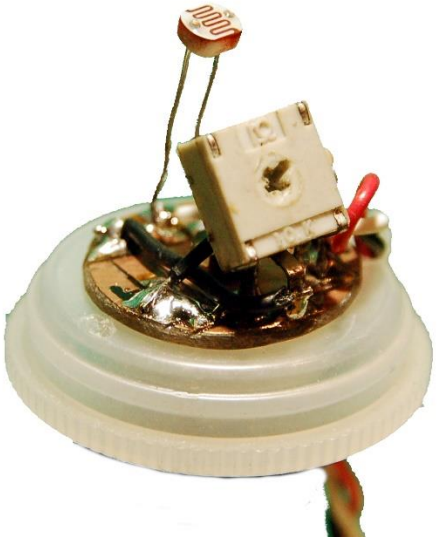

Abbildung 9: im Filmbox-Deckel

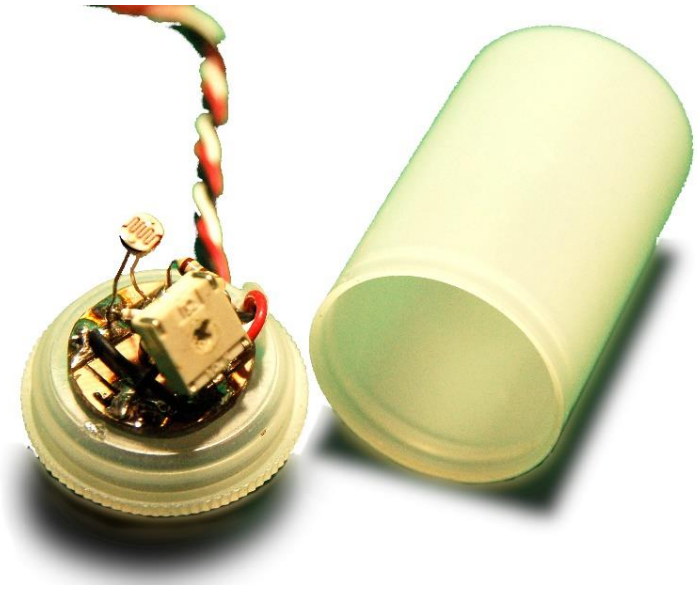

Abbildung 10: Helligkeitssensor

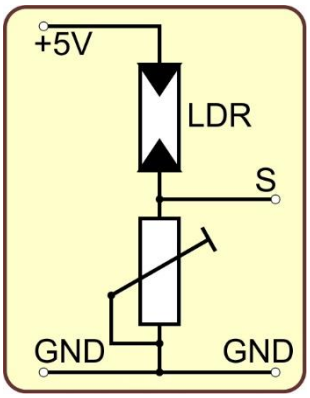

Abbildung 11: mit Trimm-Potentiometer zur Helligkeitsanpassung

Die Schaltung ist in einer Filmdose aus durchscheinendem Material untergebracht, wodurch das einfallende Licht gestreut wird. Der LDR wird also aus verschiedenen Richtungen gleichmäßiger belichtet. Weil er jetzt gegen Vcc = 5V geschaltet ist, steigt bei zunehmender Beleuchtungsstärke die Spannung am Punkt S, mehr Licht, geringerer Widerstand, höhere Spannung, höherer ADC-Wert beim Controller. Weil der Festwiderstand durch einen Trimmer ersetzt wurde, lässt sich der Ausgangsspannungsbereich zusätzlich an den Bereich der eintreffenden Lichtmenge anpassen. Zur Abdeckung eignet sich neben der Filmdose übrigens auch die abgesägte Kunststoffkugel einer LED-Lampe. Sie verteilt das Licht noch homogener.

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft
Putty als zweites Terminal neben Thonny/µPyCraft

Verwendete Firmware für den ESP8266/ESP32:

MicropythonFirmware
Bitte eine Stable-Version aussuchen

Die MicroPython-Programme zum Projekt:

nRF24simple.py Modul für den ESP8266/ESP32
master+slave.py: Demoprogramm für die ESP8266/32 TX+RX-Funktion des nRF24L01
startnrf24.py: Testprogramm für die ESP8266/32 TX+RX-Funktion des nRF24L01

MicroPython - Sprache - Module und Programme

Zur Installation von Thonny finden Sie hier eine ausführliche Anleitung. Darin gibt es auch eine Beschreibung, wie die MicropythonFirmware (Stand 26.01.2022) 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.

Die Programmierung des nRF24L01(+)

Wie alle Peripherie-Bausteine, mit mehr oder weniger komplexem Innenleben, verfügt auch der nRF24L01 über diverse Register, also interne Speicherplätze, über welche die Konfiguration und Kommunikation erfolgen. Darüber hinaus gibt es 11 Befehlscodes über welche die Funktion des Bausteins gesteuert werden kann. Das MicroPython-Modul nrf24simple.py lehnt sich nah an die Registerbezeichnungen und Befehlsnamen aus dem Datenblatt zum nRF24L01 an. Die Methoden der Klasse NFR24L01 aus diesem Modul sind zum Teil der Arduino-Bibliothek für den nRF24L01+ nachempfunden, gestalten sich aber meist kürzer und übersichtlicher.

Der SPI-Bus

Starten wir aber zunächst beim SPI-Bus. Gibt es beim I2C-Bus nur zwei Leitungen, so sind es beim SPI-Bus deren vier, SCK, MISO, MOSI und -CSN. Beim nRF24L01+ kommt eine fünfte Leitung dazu. Über CE wird das Funkmodul aktiviert. Der Anschluss ist nur der Vollständigkeit wegen eingezeichnet und hat nichts mit der SPI-Schnittstelle zu tun.

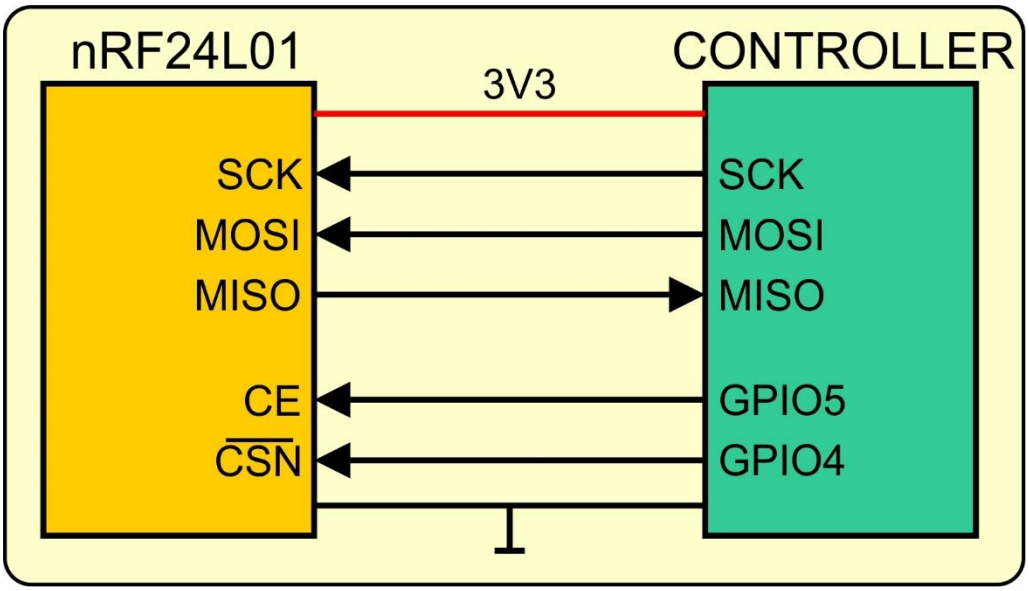

Abbildung 12: SPI-Bus-Leitungen

Auf **SCK** (Serial Clock) gibt der Controller als Master den Takt vor, mit dem Bit für Bit über die Datenleitungen geschoben wird.

Auf **MOSI** (Master Out Slave In) werden Daten vom Master zum Slave (nRF24L01) geschoben.

Auf der **MISO**-Leitung (Master In Slave Out) kommen simultan die Bits vom Slave zum Master.

Die Chip Enable-Leitung **CE**, ich erwähnte es bereits, hat nichts mit dem Betrieb auf dem SPI-Bus zu tun, sondern muss nur während des gesamten Sende- oder Empfangsvorgangs des nRF24L01 auf HIGH-Potenzial liegen. Sie schaltet also das "Radiogerät" an. Beim Senden wird zuerst über SPI der Sende-Puffer gefüllt, dann geht CE auf 1, wir warten kurz und schalten dann CE wieder auf 0. Die Methode **transmit**() arbeitet genau nach diesem Schema. Das Lauschen am Radio beginnt ebenfalls mit CE=1 und endet auch mit CE=0. Sie können das anhand der Methoden **startListening**(), **stopListening**() studieren. Das Abholen der Daten erledigt die Methode **getData**(), die immer dann aufgerufen wird, wenn **bytesAvailable**() **True** zurückgibt, wenn also Zeichen eingetroffen sind und im Empfangspuffer bereit liegen.

-CSN aktiviert die SPI-Schnittstelle, wenn dort ein LOW-Signal anliegt. Das entspricht dem üblichen Schnittstellenprotokoll. -CSN geht auf LOW, Zeichen wandern simultan über MOSI und MISO gleich danach geht -CSN wieder auf HIGH.

Gegenüber dem I2C-Bus ist bemerkenswert, dass der Datenaustauch, wie schon angedeutet, bei SPI simultan erfolgt. Es wird also stets mit einem vom Master auf MOSI gesendeten Bit auch eines vom Slave gesendet und zwar auf MISO. Im Fall des nRF24L01 beginnt der Transfer eines Bytes mit dem MSBit, dem Most Significant Bit, also dem Bit mit der höchsten Wertigkeit. Es wird aber bei der Übermittlung mehrerer Bytes über die SPI-Schnittstelle stets das LSByte, also das Byte mit der niedrigsten Wertigkeit zuerst übertragen, zum Beispiel die Pipe-Adresse.

Das Senden einer Hardwareadresse seitens des Masters wie beim I2C-Bus ist nicht nötig, da die Chipauswahl über die -CSN-Leitung erfolgt. Bei jeder negativen Flanke auf dieser Leitung, also jedem HIGH-LOW-Wechsel, sendet der nRF24L01 auf der MISO-Leitung stets mit den ersten 8 Takten automatisch den Inhalt seines Statusregisters, während der Controller gleichzeitig ein Befehlsbyte auf die MOSI-Leitung taktet. Der Pegel auf den Daten-Leitungen MISO und MOSI wird jeweils mit der steigenden Flanke auf der Taktleitung SCK übernommen. Die schwarzen Pfeile in Abbildung 13 geben die Richtung des Datenflusses an, die Zeitachse läuft immer von links nach rechts. Die Taktfrequenz beträgt 1 MHz.

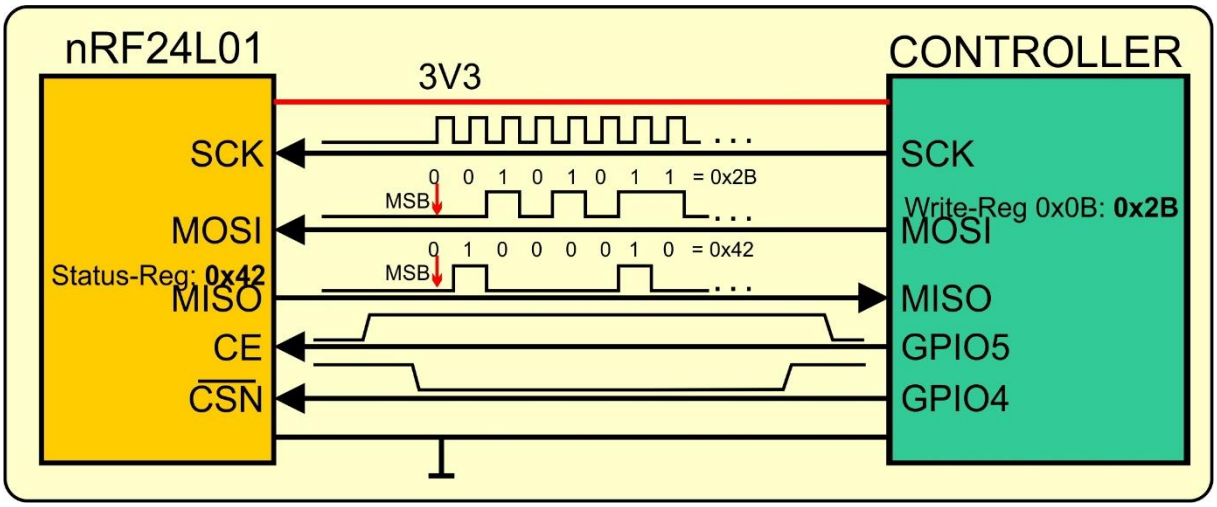

Abbildung 13: SPI-Datentransfer

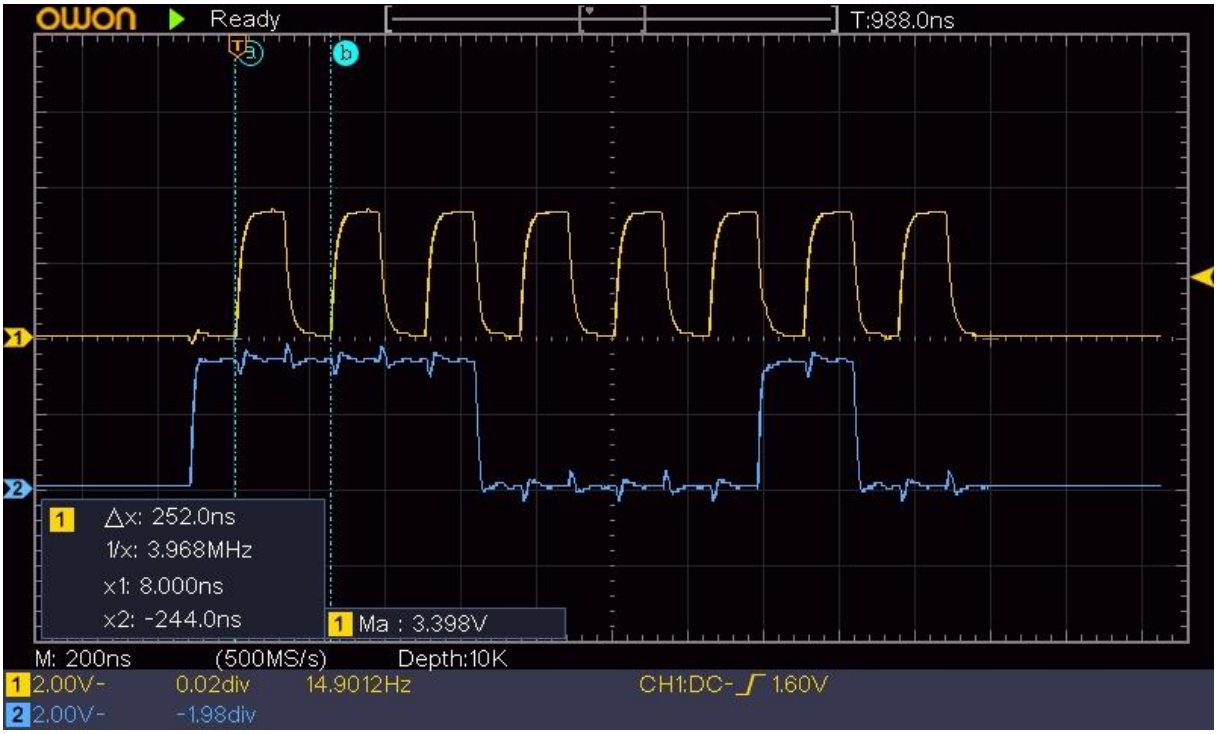

Abbildung 14: Der Befehl flushRX auf dem DSO

Aus dem DSO-Plot lesen wir heraus, dass die Spur 1 die Taktleitung mit 4MHz sein muss und gerade der Befehl flushRX (lösche den Empfangspuffer) auf Kanal 2 über die Leitung MOSI wandert. Das Befehlsbyte dafür ist 0xE2 = 0b11100010. Der Puls bei der Wertigkeit 2 zeigt deutlich, dass der Pegelwechsel auf MOSI mit der fallenden

Taktflanke und die Abtastung mit der steigenden Flanke auf SCK erfolgen. Außerdem startet die Übertragung mit einer 1, dem MSB.

Einige nRF24L01-Befehle und Register

Befehle an den nRF24L01 sind durch die 11 Kommandobytes codiert, von denen in der Klasse NRF24L01 neun als eigene Methoden definiert wurden. Sie benutzen ihrerseits weitere Methoden der Klasse für die Bedienung des SPI-Busses zum Schreiben und Lesen. Grundlage für das Umsetzen der Befehle in MicroPython-Code ist das Datenblatt des Moduls. Dort finden wir auf Seite 39 die Liste der Befehls-Codes mit Erläuterung, auf den Seiten 45ff folgt analog die Darstellung der Register.

Die Instanziierung des benötigten SPI-Objekts erfolgt im Hauptprogramm, passend zum verwendeten Controller, ESP8266 oder ESP32. Der Bus ist daher parallel auch für weitere SPI-Bausteine mit eigenem CS-Pin nutzbar. Die Variable sys.platform liefert den Typ des Controllers.

Beim Aufruf des Konstruktors der Klasse NRF24L01 wird das SPI-Objekt als erster Parameter übergeben. Es folgen die Referenzen auf die Pin-Objekte für die -CSN- und die CE-Leitung. Optional können die Kanalnummer (default: channel=50) und die Anzahl an Bytes für die Nutzlast (default: payloadSize=8) angegeben werden.

Der Konstruktor setzt automatisch die Anzahl der Adressbytes für Pipe 0 und 1 auf 5, die Sendestärke auf Minimum (-18dBm) und die Bitrate für den Funk auf 250kB/s. Letztere beschert uns die höchste Empfangsempfindlichkeit. Eine Instanzvariable buf wird als bytearray der Länge 1 deklariert und übernimmt bei den Schreib-Lese-Befehlen des SPI-Objekts die Rolle des Empfangspuffers beim Senden eines Befehlsbytes.

Wir sehen uns jetzt stellvertretend einige der Methoden näher an. Der Rest benutzt diese Methoden oder ähnlich aufgebaut.

```
    def readReg(self,reg):
        self.csn(0)
        self.spi.readinto(self.buf,reg | READ_REG_CMD)
        self.spi.readinto(self.buf)
        self.csn(1)
        return self.buf[0]
```

readReg() liest den Inhalt eines der 8-Bit-Register des nRF24L01 ein. Hier spiegelt sich der Ablauf des Datenverkehrs aus Abbildung 13 wider. -CSN wird für den Datentransfer über den SPI-Bus LOW gesetzt. Die Registernummer wird mit dem Lesebefehl für Register oderiert.

Beispiel:
Zu lesendes Register: 0x0B
Lesebefehl: 0x20
Befehlsbyte: 0x20 | 0x0B = 0x2B

Der erste SPI-Befehl **readinto**() liest das Status-Byte von MISO ein und gibt simultan das Byte 0x2B auf MOSI aus. Weil kurz zuvor die -CSN-Leitung LOW gelegt wurde, sendet der nRF24L01 den Inhalt des Status-Registers, der hier verworfen wird. Wir sind ja nur am Inhalt des in **reg** übergebenen Registers interessiert und den liest der nächste **readinto**()-Befehl ein. -CSN auf HIGH-Pegel setzen und den Inhalt des Buffers **buf** an der Stelle 0 (also das eine Byte) als Zahl zurückgeben – fertig.

```
    def writeReg(self,reg,val):
        self.csn(0) # Befehl einleiten
        self.spi.readinto(self.buf,reg | WRIT_REG_CMD)
        state=self.buf[0]
        self.spi.readinto(self.buf, val)
        self.csn(1) # SPI-Transfer beendet
        return state
```

writeReg() beginnt ähnlich, jedoch speichern wir jetzt den Status in **state** zwischen. Interessant ist, dass wir mit einem weiteren **readinto**() den Inhalt des angepeilten Registers im nRF24L01 dorthin **schreiben**. Es liegt daran, dass lesen und schreiben über den SPI-Bus simultan erfolgen. Das heißt, dass durch den Lesebefehl der Inhalt von **val** auf MOSI zum nRF24L01 wandert. Was jetzt über MISO ankommt verschwindet im Nirvana. Stattdessen geben wir den zwischengespeicherten Status zurück.

```
    def writeBuffer2Reg(self,reg,buffer):
        # schreibt den Inhalt von buffer an Register reg
        self.csn(0)
        self.spi.readinto(self.buf, WRIT_REG_CMD | reg)
        self.spi.write(buffer)
        self.csn(1)
        return self.buf[0]
```

Nur wenn mehr als ein Byte geschrieben werden soll, verwenden wir den **spi.write**()-Befehl. Er sendet jedes Byte in **buffer** an den nRF24L01. Das bedeutet aber auch, dass der Buffer nicht länger sein darf wie die von ihm transportierte Nutzlast.

Diese drei Methoden werden von fast allen anderen Methoden zum Datentransfer genutzt. **setChannel**() grenzt den übergebenen Wert der Kanalnummer auf den gültigen Bereich ein und überträgt das Ergebnis an das Register RF-CH (=0x05).

```
    def setChannel(self,kanal):
        # Kanalnummer setzen 0..125
        self.writeReg(RF_CH, max(0,min(kanal, 125)))
```

Es gibt drei von uns benutzte Methoden, die außer dem Kommando-Byte kein weiteres Byte senden. Diese nutzen selbst auch die **readinto**()-Methode, wie beispielsweise **flushTX**().

```
    def flushTX(self):
        # Sende-Puffer leeren, Kommando parameterlos
```

```
        self.csn(0)
        self.spi.readinto(self.buf, FLUSH_TX)
        self.csn(1)
```

Durch das Abfragen und Setzen von Registerinhalten wird der nRF24L01 konfiguriert.

```
    def setTXConfig(self,baud,power):
        # setzt Leistung und Geschwindigkeit
        val=(self.readReg(RF_SETUP)&0b11010001)|baud|power
        self.writeReg(RF_SETUP,val)
```

RF_SETUP-Register einlesen, die relevanten Bits durch undieren rücksetzen und durch oderieren die neuen Werte setzen und dann in das Register zurückschreiben. Hier werden die Baudrate und die Sendestärke für den Funk gesetzt.

Register spielen auch für den Datenverkehr selbst eine Rolle.

```
    def getData(self):
        self.csn(0)
        self.spi.readinto(self.buf, R_RX_PAYLOAD)
        buffer = self.spi.read(self.payloadSize)
        self.csn(1)
        self.writeReg(STATUS, RX_DR)
        return buffer
```

Wenn Daten per Funk eingetroffen sind, holen wir sie ab, **getData**(). Dazu senden wir den Befehl R_RX_PAYLOAD = lies die empfangenen Daten. Dann holen wir so viele Bytes ab, wie in der Instanz-Variablen **PayloadSize** festgesetzt wurden. -CSN geht wieder auf 1 und nach dem Löschen des Empfangsflags **RX_DR** im Register **STATUS** geben wir den Bufferinhalt zurück.

Eine bemerkenswerte Methode, die nicht direkt etwas mit den nRF24L01-Registern zu tun hat ist TimeOut(). Sie definiert mit **compare**() eine sogenannte Closure.

```
    def TimeOut(self,t):
        start=ticks_ms()
        def compare():
            return int(ticks_ms()-start) >= t
        return compare
```

In **t** übergeben wir eine Zeitdauer in Millisekunden. Innerhalb **TimeOut**() definieren wir die Funktion **compare**(), auf die **TimeOut**() eine Referenz zurückgibt. **compare**() vergleicht die Differenz zwischen der aktuellen Zeit und der Startzeit mit der übergebenen Zeitspanne in **t** und gibt **True** zurück, wenn **t** ms abgelaufen sind. Mit dem Aufruf von TimeOut() setzen wir zum Beispiel t auf 10000.

>>> timer=TimeOut(10000)

timer() vertritt jetzt selbst eine Funktion, weil wir der Variablen die Referenz auf **compare**() zugewiesen haben. Wir können nun **timer**() im Thonny-Terminal aufrufen und erhalten **False** als Antwort, bis 10 Sekunden vorüber sind. **timer**() ist eine Closure, deren Angewohnheit es ist, sich beim nächsten Aufruf an den Inhalt lokaler Variablen zu erinnern, auch wenn die Funktion zwischendurch verlassen wurde.

Mit **TimeOut**() lassen sich (fast) beliebig viele Timer mit (fast) beliebigen Ablaufzeiten einrichten. Eine weitere Besonderheit daran ist, dass diese Timer den Programmablauf nicht blockieren wie zum Beispiel **sleep**() oder **sleep_ms**().

Alle Befehle, die für unser Projekt benötigt werden, wurden in entsprechende Methoden umgesetzt. Sie finden diese und noch ein paar weitere in der Datei nrf24simple.py, die wir als Modul in unser Programm einbauen. Damit wir die Methoden dieses Moduls von Hand auf der Kommandozeile von Thonny testen können, habe ich den Definitionsteil des nachfolgend besprochenen Programms **master+slave.**py in das Testprogramm **startnrf24**.py kopiert. Dieses Programm können wir vom Editorfenster aus mit **F5** starten. Im Terminal sind nun manuell alle Methoden der Klasse NRF24L01 im Test überprüfbar.

Master und Slave

Wesentliche Punkte des Datenverkehrs vom und zum nRF24L01 via SPI und die Funkdatenübermitlung sind behandelt, dann geht's jetzt in den Endspurt. Zum Testen der gesamten Anlage brauchen wir einen Sender und einen Empfänger. Damit man nicht zwei Programme pflegen muss, habe ich die beiden Einheiten in ein Programm zusammengepackt. Eine Zeile entscheidet über Sender (master = True; Slave=False) oder Empfänger (master=False; slave=True). Das finden wir im Listing Zeile 8 respektive 9, je nachdem welche von beiden entkommentiert ist. Ein paar weitere Klassen und Methoden sind neben NRF24L01 für unser Vorhaben zu importieren.

```
# master+slave.py
import sys, os
from time import sleep_ms,ticks_ms
from machine import Pin, SPI
from nrf24simple import NRF24L01
# -------------------------------------------------------
#master=False; slave=True
master=True; slave=False
# -------------------------------------------------------
```

```
led=Pin(2,Pin.OUT,value=1)
def blink(led,pulse,wait,inverted=False,repeat=1):
    for i in range(repeat):
        if inverted:
            led.off()
```

```
            sleep_ms(pulse)
            led.on()
            sleep_ms(wait)
        else:
            led.on()
            sleep_ms(pulse)
            led.off()
            sleep_ms(wait)
```

Die onBoard-LED, soweit vorhanden, benutzen wir mangels Displays für die Rückmeldung von Programmzuständen, denn beim Versuch im Freien haben wir dafür ja auch kein Terminal verfügbar. Die Funktion **blink**() hilft uns wie üblich dabei. Sie kann Blinkpulse einzeln oder als Folge mit variabler Pausendauer erzeugen und berücksichtigt durch den optionalen Parameter **inverted** sowohl LEDs, die gegen Masse geschaltet werden (**True**), als auch solche, die durch den HIGH-Pegel am Ausgangspin aktiviert werden (**False**). Das nutzen wir auch gleich für die erste Fehlermeldung, wenn **master** und **slave** beide versehentlich auf **True** gesetzt wurden.

```
if master and slave:
    blink(led,500,100,inverted=True,repeat=5)
    raise OSError ("ENTWEDER slave ODER master!")
```

```
chip=sys.platform
taste=Pin(0,Pin.IN,Pin.PULL_UP)
if chip == 'esp8266':
    # 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
    bus = 1
    MISOp = Pin(12)
    MOSIp = Pin(13)
    SCKp  = Pin(14)
    spi=SPI(1,baudrate=4000000)   #ESP8266
    # # alternativ virtuell mit bitbanging
    # spi=SPI(-1,baudrate=4000000,sck=SCK,mosi=MOSI,\
    #         miso=MISO,polarity=0,phase=0)  #ESP8266
    if slave:
        adc=ADC(0)

elif chip == 'esp32':
    bus = 1
    MISOp= Pin(15)
    MOSIp= Pin(13)
    SCKp = Pin(14)
    spi=SPI(1,baudrate=4000000,sck=Pin(14),mosi=Pin(13),\
            miso=Pin(15),polarity=0,phase=0)  # ESP32
    if slave:
        adc=ADC(Pin(39)) # Pin SP
```

```
        adc.atten(ADC.ATTN_11DB)
        adc.width(ADC.WIDTH_12BIT)
else:
    blink(led,800,100,inverted=True,repeat=5)
    raise OSError ("Unbekannter Port")
    at(MISOp,MOSIp,SCKp))

print("Hardware-Bus {}: Pins fest vorgegeben".format(bus))
print("MISO {}, MOSI {}, SCK {}\n".format(MISOp,MOSIp,SCKp))
```

Wir haben das Modul **sys** importiert, weil uns der String **sys.platform** den Typ des Ports verrät. Je nachdem können wir individuell auf die Eigenheiten des SPI-Busses und auf das Einrichten des Analogeingangs reagieren. Die beiden **print**-Befehle informieren darüber, was letztlich eingestellt wurde.

```
CSN = Pin(4, mode=Pin.OUT, value=1)
CE =  Pin(5, mode=Pin.OUT, value=0)
nrf = NRF24L01(spi, CSN, CE, payloadSize=8)
kanal=50
pipeAdr=[0x5A5A5A5A54,0x5A5A5A5A52]
```

CSN und CE liegen bei ESP32 und ESP8266 auf denselben GPIO-Pins. **nrf** ist die Instanz der Klasse **NRF24L01**, die mit den standardmäßigen 5 Bytes Adressbreite und einer Payload-Länge von 8 Bytes definiert wird. Die Variable für die Kanalnummer wird auf 50 gesetzt. Jeder Funk-Kanal kann über bis zu 6 sogenannte Pipes Nachrichten von Slave-Einheiten erhalten. Dafür braucht jede Pipe eine eindeutige Nummer, die Pipe-Adresse. Die Adresse, für die Empfangs-Pipe 0 (RX) wird auch für die Sende-Pipe (TX) verwendet. Sie wird ebenso wie die Pipe 1 durch die vereinbarten 5 Bytes definiert. Es genügt, wenn sich die Adressen der Empfangs-Pipes durch ein Byte unterscheiden. Das ist in der Regel das LSByte. Die zwei bis vier höherwertigen Bytes stellen quasi die Gruppenadresse des Kanals dar. Diese Bytes werden auch transparent zur Adressierung der Pipes 2 bis 5 verwendet, für die nur ein LSByte als Adresse im nRF24L01 gesetzt werden kann. Die restlichen Adressbytes werden von denen der Pipe 1 genommen. Die Übermittlung der Adresse an den nRF24L01 via SPI beginnt mit dem LSByte. Diese Tatsache wird durch Methoden openTXPipe() und openRXPipe() automatisch berücksichtigt.

Die Sendefrequenz des Kanals ergibt sich übrigens, indem man die Kanalnummer mit der Einheit MHz zur Basis des Frequenzbandes 2400MHZ addiert. Kanal 50 sendet und empfängt somit auf 2450MHz oder 2,450GHz.

Der nRF24L01 kann zweierlei Jobs erledigen.

a) Zwei Stationen bidirektional im Halbduplex verbinden
Beide Stationen, Master und Slave, können abwechselnd auf derselben Frequenz (Kanal) senden. Der Slave ist zunächst Listener (Zuhörer, Receiver) und wartet auf den Eingang einer Nachricht vom Master (Transmitter, Sender). Dann erledigt der Slave seinen Job aus der Anfrage und sendet das Ergebnis zurück.

b) Ein Receiver kann bis zu sechs logische Verbindungen zu Transmittern, die sogenannten Pipes, auf einem Kanal bedienen. Halbduplex ist möglich, aber aufwendig zu programmieren. Die sechs Eingangs-Pipes des Receivers werden auf die Adressen der sechs Transmitter-Einheiten gesetzt.

Der Master

Das folgende Programm nutzt Version a). Schauen wir uns zuerst das Programm für den Master genauer an.

```
if master:
    versucheMax = 5
    erfolgreich = 0
    fehler  = 0
    durchgang = 0
    nrf.setChannel(kanal)
    print("MASTER-Modus: Sending on channel ",kanal)
    nrf.openTXPipe(pipeAdr[1])
    nrf.openRXPipe(1, pipeAdr[0])
    print("MASTER-Modus, sende {} Pakete".format(versucheMax))
    nrf.info()
```

Mit **versuchemax**=5 geben wir die Anzahl von Verbindungsversuchen vor. Wir lassen die Durchgänge, Treffer und Fehlanzeigen mitzählen. Der vorgegebene Kanal wird eingestellt, dann öffnen wir die Sende- und Empfangs-Pipes, indem wir den Verbindungen die eindeutigen Adressen zuordnen. **nrf.info**() liefert uns eine ganze Reihe von Informationen zur Konfiguration des nRF24L01. Folgende Punkte müssen bei der Adresszuordnung beachtet werden.

a) Jede Einheit hat eine TX-Pipe deren Adresse gleichzeitig auch der RX-Pipe 0 zugeordnet wird. Das macht die Methode **nrf.openTXPipe**().

b) Die Methode **nrf.openRXPipe**(nr,adr) ordnet der RX-Pipe nr die Adresse adr zu.

c) Von der RX-Adresse adr wird nur das LSByte herausgepickt, falls nr >= 2 ist.

d) Die MSBytes der Pipes 2 bis 5 werden von der Pipe 1 übernommen (grau in Abb. 15)

e) Die Adresse von TX- und somit RX-Pipe 0 darf sich von den anderen Adressen in jedem Byte unterscheiden.

f) Die Adressen für eine Pipe müssen auf der Sende- und Empfangsseite übereinstimmen.

g) Ähnlich wie die Kabelanschlüsse bei einer RS232-Verbindung müssen die Zuordnungen gekreuzt werden.

h) Nach dem Kaltstart des nRF24L01 sind bereits Adressen nach diesen Vorgaben eingerichtet.

Nach dem Start von startnrf24.py im Editorfenster mit F5:

>>> nrf.info()

Ausgabe:

…
Receive Address, pipe 0: b'\xe7\xe7\xe7\xe7\xe7'
Transmit Address: b'\xe7\xe7\xe7\xe7\xe7'
Receive Address, pipe 1: b'\xc2\xc2\xc2\xc2\xc2'
Receive Address, pipe 2: b'\xc3'
Receive Address, pipe 3: b'\xc4'
Receive Address, pipe 4: b'\xc5'
Receive Address, pipe 5: b'\xc6'

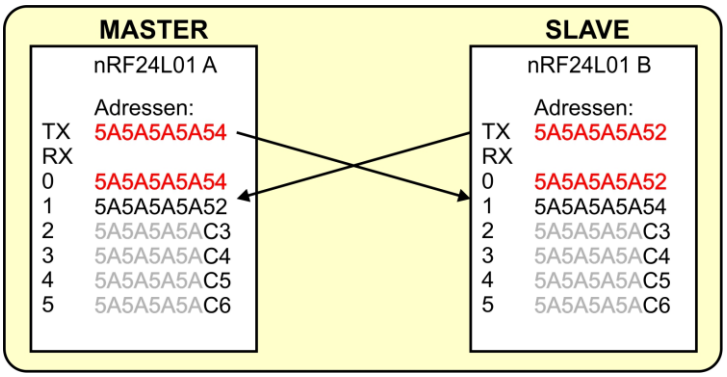

Abbildung 15: Master und Slave - Verbindungszuordnung wie bei der RS232

Der Punkt f) aus der obigen Aufzählung macht deutlich, dass ein Halbduplexbetrieb bei mehreren Slaves mit einem erhöhten Aufwand verbunden ist. Um sechs Slaves anfunken zu können, müsste bei jedem Verbindungswechsel die TX-Adresse des Masters ausgetauscht werden.

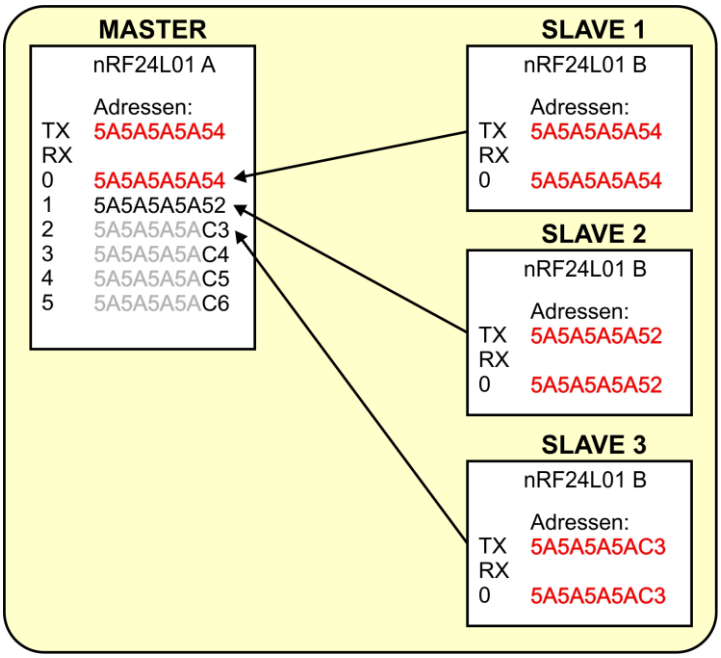

Abbildung 16: Multi-Transmitter-Modus

```
    while durchgang < versucheMax:
        nrf.stopListening()
        text="send:{}".format(durchgang)
        print("\nsending:", text)
        try:
            nrf.sendData(text.encode())
        except OSError:
            pass
```

Die while-Schleife zählt die Durchgänge und kann im Produktionssystem einfach durch while 1: ersetzt werden. Wir stoppen den Lauschmodus durch **stopListening**(). Dadurch wird das Funkmodul des nRF24L01 ausgeschaltet. Der zu sendende Text wird zusammengestellt und, als Bytesobjekt codiert, gesendet. Aufgetretene Fehler werden durch **try – except** abgefangen. Die Methode **sendData**() schaltet das Sendemodul selbständig zu gegebener Zeit ein und aus.

```
        nrf.startListening()
        timedOut=nrf.TimeOut(50)
        timeState=timedOut()
        while not nrf.bytesAvailable()and not timeState:
            timeState=timedOut()
```

Wir warten jetzt auf die Antwort vom Slave und schalten dafür das Radio wieder an und auf Lauschstation. Wir machen den Timer **timedOut**() auf 50ms scharf und lauern auf eingegangene Bytes. **timeState** wird mit **True** vorbelegt und mit jedem Schleifendurchlauf auf den Zustand des Timers gebracht.

```
        if timeState:
            print("Timeout!", durchgang)
            fehler += 1
            blink(led,50,200,inverted=True)
        else:
            antwort=nrf.getData()
            print("Durchgang:",durchgang)
            response=antwort.decode()
            print("Antwort:",response.strip("\x00\n\r"))
            erfolgreich += 1
```

Sind innerhalb von 50ms keine Zeichen eingetroffen, wird **timeState True**. Wir erhöhen den Fehlerzähler und geben eine Timeout -Meldung aus.

Andernfalls holen wir die Daten ab, decodieren zum String, von dem wir die nichtdruckenden Zeichen (\x00,Linefeed und Carriage return) entfernen und geben den Text aus. Zum Schluss gibt's noch eins drauf für den Erfolgszähler.

```
        blink(led,50,950,inverted=True)
        durchgang +=1
    print("Von {} Durchgaengen waren {} erfolgreich.".\
          format(versucheMax,erfolgreich))
```

Ein "Blink" nach jedem Durchgang macht zusammen eine Sekunde Pause bis zur nächsten Abfrage. Dann Schlusszusammenfassung, Masterteil fertig.

Der Slave

Der Slave startet fast genauso wie der Master, aber wir müssen die umgekehrte Zuordnung der Adressen beachten. Die TX-Pipe bekommt Adresse 1 und die RX-Pipe 1 kriegt die Adresse 0.

```
if slave:
    nrf.setChannel(kanal)
    nrf.openTXPipe(pipeAdr[1])
    nrf.openRXPipe(1, pipeAdr[0])
    nrf.info()
    nrf.startListening()
    print("SLAVE-Modus: Listening on channel ",kanal)
```

Dann geht es auch schon in die Jobschleife.

```
    while True:
        buffer = b''
        if nrf.bytesAvailable():
            #print(".",end='')
            while nrf.bytesAvailable():
                recv=nrf.getData()
                print(recv)
                buffer = buffer+recv
                sleep_ms(15)
            msg=buffer.decode()
            print("got:",msg)
            pos=msg.find(":")
            wert=str(adc.read())
            antwort=wert.encode()
```

Wir löschen das Bytesobjekt **buffer** in welchem die empfangenen Zeichen gesammelt werden, solange welche eintrudeln. Wir decodieren das Bytesobjekt zum String und suchen nach einem ":". Wir könnten jetzt prüfen ob die Zeichen bis zum Doppelpunkt einem Befehl entsprechen und welche Zahl danach folgt. Statt dessen hole ich gleich den LDR-Wert vom ADC, wandle zum String um und codiere diesen als Bytes-Objekt.

```
            nrf.stopListening()
            try:
                nrf.sendData(antwort)
                blink(led,50,1950,inverted=True)
            except OSError:
                pass
            print("gesendet: {}".format(antwort))
            nrf.startListening()
```

```
        if taste.value()==0:
            sys.exit()
```

Mit **stopListening**() schalten wir zum Sendemodus um. Wir senden unsere Antwort und starten danach wieder den Lauschangriff.

Mit der Flashtaste am ESP8266-Board können wir an dieser Stelle das Programm abbrechen, um wieder in der Editiermodus zu gelangen.

Mit dem Download der Datei **master+slave.py** erhalten Sie den gesamten Programmtext.

Der Test

Für den Test brauchen wir eine Station nach dem Muster der Abbildung 4 als Master und eine Schaltung nach Abbildung 5 als Slave. Sie können als Slave auch einen ESP8266 verwenden, müssen dann allerdings für den Slave den Masterteil des Programms entfernen, da Sie sonst Speicherprobleme bekommen.

Um beide Einheiten steuern beziehungsweise überwachen zu können, brauchen wir ein zweites Terminal. Hierzu benutzen wir Putty. Laden sie am besten eine ausführbare Version für Ihr System herunter und speichern Sie diese in einem Verzeichnis Ihrer Wahl.

Wir steck nun die Mastereinheit am PC-an und werden sie für einen autonomen Start einrichten.

```
#master=False; slave=True
master=True; slave=False
```

Das Programm wird als Master deklariert. Auf den ESP8266 laden wir das Modul nrf24simple.py hoch und rufen dann die Konfiguration auf, rechts unten im Thonny-Fenster.

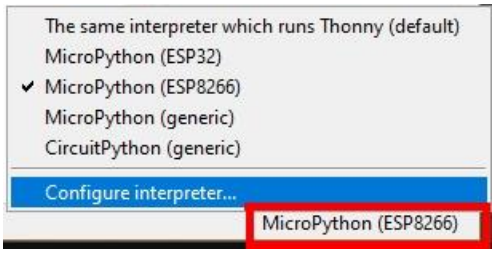

Abbildung 17: Konfiguration aufrufen

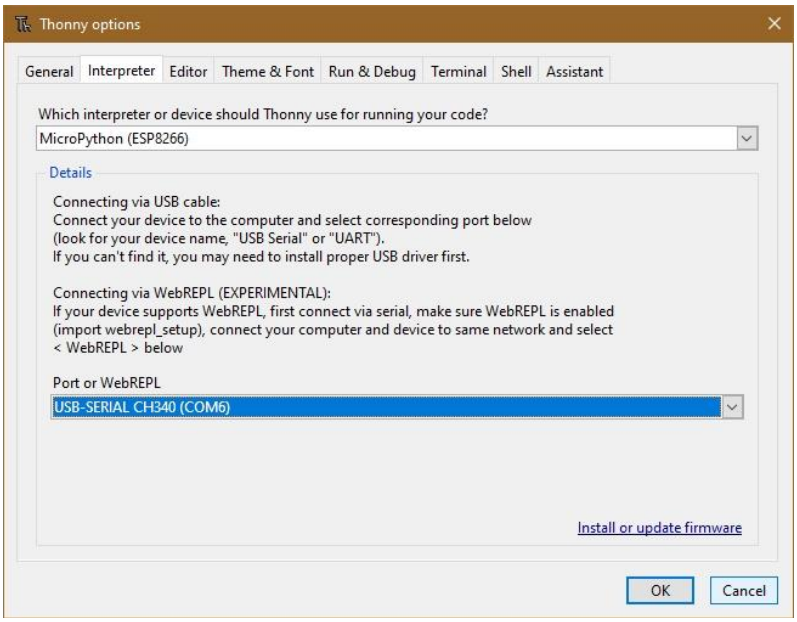

Abbildung 18: Options für die Mastereinheit

Die Nummer der COM-Schnittstelle, hier **COM6**, merken wir uns für später, OK. Wir rufen über die Taste **F5** das Programm **master+slave.**py im Editorfenster auf. Wenn es fehlerfrei läuft, die Timeout-Meldungen können wir erst einmal ignorieren, markieren wir den gesamten Text mit **Strg+A** und kopieren ihn in die Zwischenablage. Jetzt öffnen wir die Datei **boot.py** vom ESP8266 durch Doppelklick in einem Editorfenster.

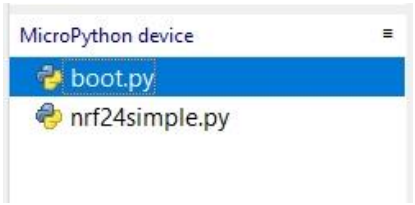

Abbildung 19: Bootdatei vom ESP8266 öffnen

Den gesamten Text dort markieren wir erneut durch Strg+A und fügen unseren Programmtext aus der Zwischenablage ein. Unser Text ersetzt den markierten. Mit Strg+S speichern wir die **boot.py** zurück auf das Board.

Als Nächstes stöpseln wir den Slaveaufbau am PC an und stellen die Konfiguration von Thonny um.

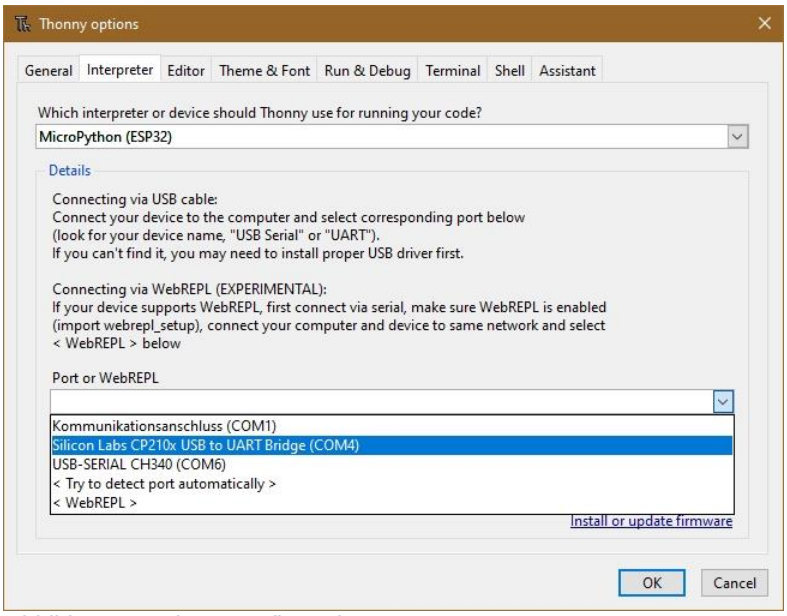

Abbildung 20: Slave-Konfiguration

Damit haben wir den Anschluss COM6 unter Thonny freigegeben und können diesen jetzt in Putty verwenden. Nachdem Putt gestartet ist, stellen wir die Parameter analog nach Abbildung 21 ein und speichern das Profil ab – **Save**. Mit **Open** wird ein Terminal geöffnet.

Abbildung 21: Putty ausführen

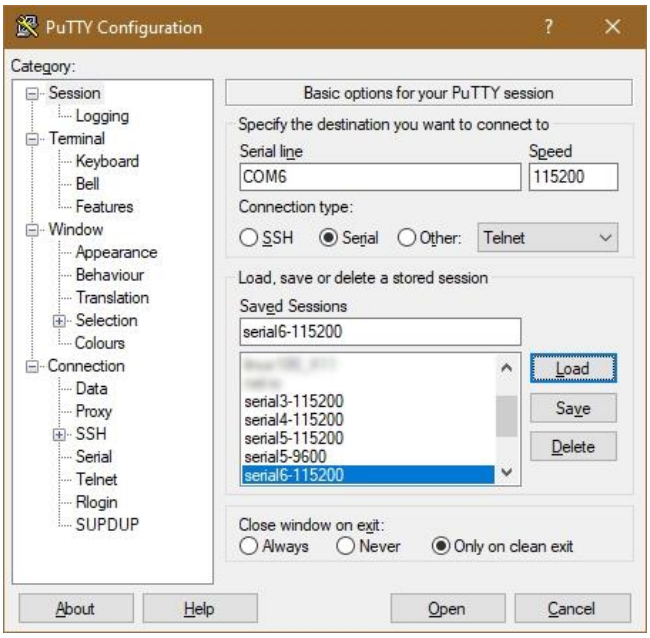

Abbildung 22: Putty Beispiel-Konfiguration

Alle Meldungen, die zuvor in Thonny im Terminalbereich angezeigt wurden, erscheinen jetzt in Putty, nachdem der Master durch drücken der RST-Taste neu gestartet wurde. Danach wird der Kommandozeilenprompt von MicroPython angezeigt.

Wir wechseln zurück in die Thonny-Umgebung und setzen den ESP32 als Slave ein.

```
master=False; slave=True
#master=True; slave=False
```

Nach dem Speichern starten wir mit F5 das Programm **master+slave.py** im Editorfenster und resetten das Masterboard. Wenn die Ausgaben in den Terminalfenstern jetzt so ähnlich wie in den Abbildungen 23 und 24 aussehen, dann haben Sie es geschafft, den beiden ESPs einen neuen Kommunikationsweg zu erschließen.

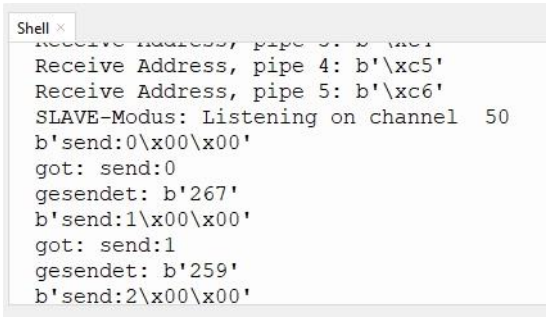

Abbildung 23: Output im Slave-Terminal

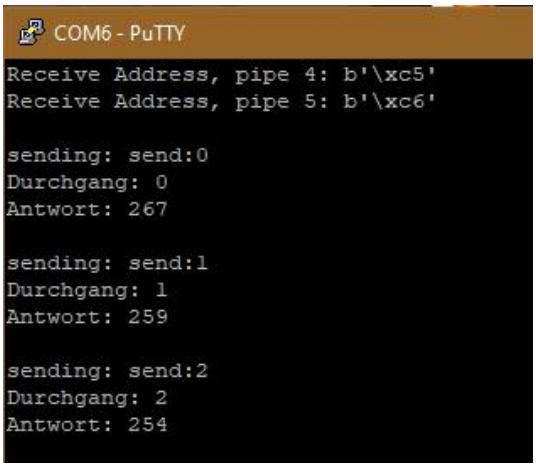

Abbildung 24: Ausgabe am Master-Terminal

Natürlich ist es sicher interessant, dass sich ESPs mitunter unentdeckt vom WLAN-Verkehr unterhalten können. Viel entscheidender ist jedoch der Fall, dass Arduinos und deren Clones über Funk an eine Relaisstation andocken können, welche deren Meldungen dann zum Beispiel ins WLAN weitergeben kann. Genau das schauen wir uns in der nächsten Folge an.