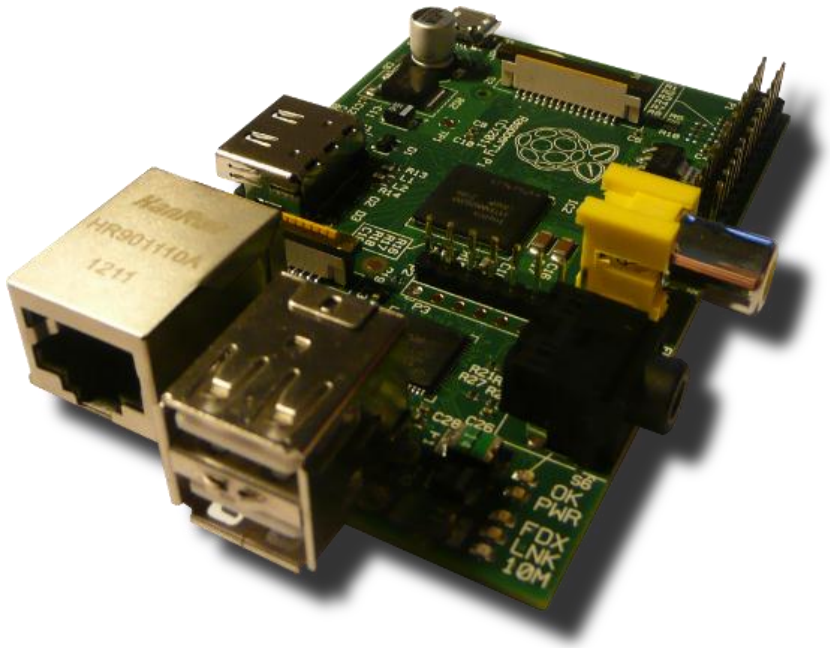
PiMeUp

Raspberry Pi



AddOn

Der A/D-Wandler MCP3008

Version 1.2,
01.11.2012

www.erik-bartmann.de

Raspberry Pi

Worum geht's?

Hallo zusammen,

in diesem *RasPi-AddOn* möchte ich ein paar Worte über den Baustein *MCP3008* erzählen. Warum? Nun, das hat ganz einfache und praktische Gründe. Einige von euch werden sicherlich die Stirn runzeln, wenn sie die Bezeichnung *MCP3008* lesen. Was könnte das wohl sein. Es handelt sich um einen integrierten Schaltkreis, der wie folgt aussieht:

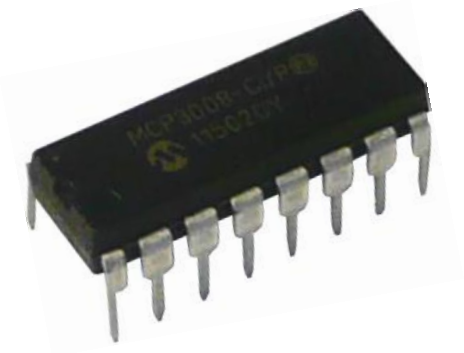
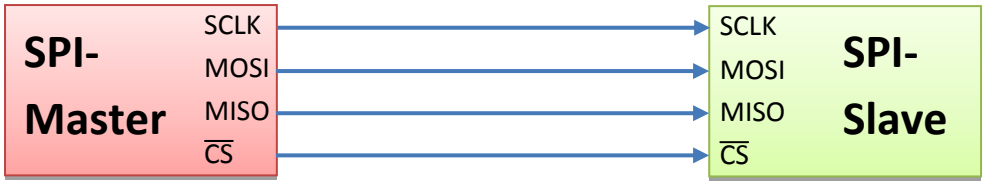


Abbildung 1 Der MCP3008

Was ist aber seine Funktion? Da werde ich ein wenig ausholen müssen. Der *Raspberry Pi* kann über seine *GPIO-Schnittstelle* Kontakt zu seiner Außenwelt aufnehmen. Die vorhandenen Pins können entweder als digitale Ein- oder Ausgänge programmiert werden. Es fehlt die Möglichkeit, *analoge Signale* auszuwerten, so wie es z.B. der *Arduino* mit seinen *6* analogen Eingängen in der Lage ist. Zum Glück weisen jedoch einige der GPIO-Pins eine alternative Funktionalität auf, die es uns ermöglicht z.B. über einen *I^2C-Bus* oder über *SPI* mit externen Komponenten zu kommunizieren. Bleiben wir doch bei *SPI*.

Was bedeutet SPI?

Diese Abkürzung steht für *Serial Parallel Interface* und ist ein von *Motorola* entwickeltes Bus-System, um zwischen integrierten Schaltkreisen mit möglichst wenigen Leitungen Daten auszutauschen. Diese Bausteine werden nach dem *Master-Slave-Prinzip* miteinander verbunden.



Es existieren *4* Leitungen, die vom *Master* zum *Slave* oder auch mehreren *Slaves* führen. In diesem *AddOn* wollen wir den Baustein *MCP3008* über die *SPI-Schnittstelle* mit dem *Raspberry Pi* verbinden. Dieser integrierte Baustein ist ein *Analog/Digital-Wandler* mit *8* unabhängig voneinander arbeitenden Eingängen, die jeweils eine Auflösung von *10-Bits* besitzen. Das ist schon eine Menge an Funktionalität, die so ein Baustein bereitstellt und das alles wird über *4* Leitungen gehandhabt. Wie aber funktioniert *SPI*? Damit eine Kommunikation zustande kommt, müssen Daten in beide Richtungen fließen. Also vom *Master* zum *Slave* und umgekehrt. Das alles erfolgt über zwei getrennte Leitungen.

- MOSI (*Master-Out-Slave-In*)
- MISO (*Master-In-Slave-Out*)

Für jede Richtung wird eine einzige Leitung benötigt. Die Datenübertragung erfolgt zwischen den beiden Busteilnehmern also *seriell* und *synchron*. Da der *Master* der Hauptverantwortliche bei dieser Kommunikation ist, wird die *MOSI*-Signalleitung auch als *Serial-Data-Out*, kurz *DO* bezeichnet, wobei die *MISO*-Signalleitung im Gegensatz dazu als *Serial-Data-In*, kurz *DI* arbeitet. Nun können diese Signale nicht einfach so auf den Bus gelegt werden. Es fehlt eine Synchronisationsinstanz, damit alle wissen, wann welche Signale kommen bzw. wann sie zeitlich beginnen bzw. enden. Aus diesem Grund gibt es noch die *SCLK*-Leitung (*Serial-Clock*), die quasi den *Schiebetakt* vorgibt, vergleichbar mit dem Paukenschlag auf einer Galeere. Zu guter Letzt müssen noch die am Bus angeschlossenen Teilnehmer (*Slave*) ausgewählt werden, damit klar ist, zu wem eine Kommunikation aufgebaut werden soll. Wir nutzen in unserem Beispiel nur einen einzigen *Slave*. Dafür ist die *CS*-Leitung (*Chip-Select*) verantwortlich. Wie diese einzelnen Signale anzusteuern sind, dass wirst Du im Kapitel über die Programmierung erfahren. Sehen wir uns doch zunächst einmal den integrierten Baustein *MCP3008* aus der Nähe an.

Der MCP3008

Mittlerweile weißt du sicherlich Bescheid, worum es in diesem *AddOn* geht. Wir wollen mit Hilfe des *MCP3008* das Manko eines fehlenden *Analog/Digital-Wandlers* am *Raspberry Pi* beheben. Der *MCP3008* hat sein Zuhause in einem *16*-poligen *DIL*-Gehäuse gefunden. Es handelt sich um den besagten *Analog/Digital-Wandler* mit einer *10-Bit* Auflösung und *8* unabhängigen Kanälen. Schauen wir doch einfach einmal in das Gehäuse hinein und erkunden die Bedeutungen der einzelnen Pins.

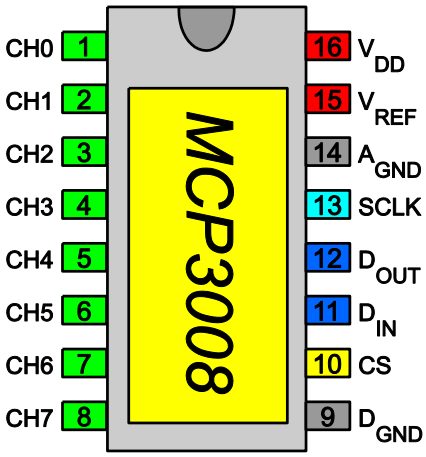


Abbildung 2 Die Pinbelegung des MCP 3008

Der integrierte Schaltkreis ist schön symmetrisch aufgebaut, so dass sich die analogen Eingänge aus dieser Sicht allesamt auf der linken Seite befinden. Auf der rechten Seite müssen wir die Spannungsversorgung und die Steuerleitungen anschließen. Das ist aber absolut kein Hexenwerk. Schauen wir uns zunächst die Leitungen auf der rechten Seite des Bausteins an:

- V_{DD} (Spannungsversorgung: 3,3V)
- V_{REF} (Referenzspannung: 3,3V)
- A_{GND} (Analoge Masse)
- $SCLK$ (Clock)
- D_{OUT} (Data-Out vom *MCP3008*)
- D_{IN} (Data-In vom *Raspberry Pi*)
- $\overline{CS}$ (Chip-Select, *LOW-Aktiv*)
- D_{GND} (Digitale Masse)

Die eigentliche Kommunikation findet über die beiden Leitungen D_{OUT} und D_{IN} statt.

Die analogen Eingänge befinden sich auf der linken Seite des Bausteins, wobei die einzelnen Pins die Bezeichnung *CH0* bis *CH7* besitzen. Es handelt sich um die 8 Kanäle des *AD-Wandlers*. Wie du die Kanäle ansteuern kannst, das wirst Du gleich im Schaltplan sehen. Der Pin A_{REF} wurde bei uns mit *3,3V* versehen, so dass die Eingangsspannung von *0V* bis *3,3V* schwanken darf. Die Frage, die sich uns sicherlich an dieser Stelle aufdrängt ist die Folgende: „Wenn wir eine *10-Bit Auflösung* haben, wie groß bzw. klein ist die Spannung pro Bit?“ Schauen wir zuerst einmal, wie viele unterschiedliche Bitkombinationen wir mit *10-Bits* erreichen können. Dies wird über die folgende Formel berechnet:

$$Anzahl\ der\ Bitkombinationen = 2^{Anzahl\ der\ Bits}$$

$$Anzahl\ der\ Bitkombinationen = 2^{10} = 1.024$$

Wenn wir jetzt die Referenzspannung von *3,3V* durch diesen Wert dividieren, dann erhalten wir den Spannungswert pro Bit-Sprung.

$$U = \frac{U_{REF}}{1024} = \frac{3{,}3V}{1024} = 0{,}003222V = 3{,}2mV$$

Wenn Du nun die anliegende Spannung berechnen möchtest, dann musst du lediglich den ermittelten Wert, der sich zwischen *0* und *1.023* bewegen kann, mit *3,2mV* multiplizieren. Hier ein kleines Beispiel dazu. Das Programm, das wir uns gleich anschauen werden, liefert z.B. einen Wert von *512* zurück, was bedeutet, dass du folgenden Spannungswert am analogen Eingang anliegen hast:

$$Berechnete\ Spannung = 512 \cdot 3{,}22mV = 1{,}65V$$

Und *hey*... das ist genau die Hälfte von U_{REF}, denn *2*-mal *1,65V* entsprechen *3,3V*. Warum? Ganz einfach: *512* ist auch genau die Hälfte von *1.024*. Doch nun haben wir erst einmal genug gerechnet. Bevor wir uns der Programmierung widmen, werfen wir einen Blick auf den Schaltplan.

Der Schaltungsaufbau mit Fritzing

Mit der Software Fritzing lässt der Schaltungsaufbau sehr gut darstellen.

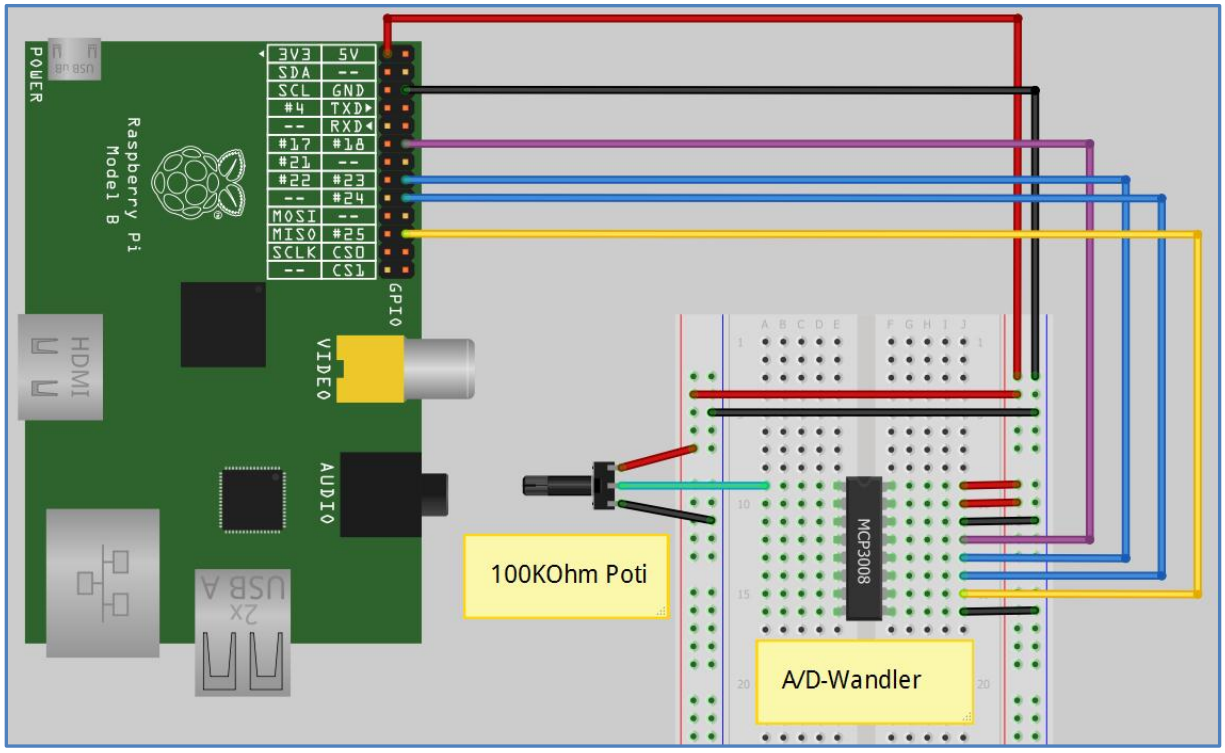


Abbildung 3 Schaltungsaufbau mit Fritzing

Auf der linken Seite siehst Du unseren *Raspberry Pi*, der über die *GPIO-Schnittstelle* mit dem auf einem Breadboard aufgesteckten *A/D-Wandler MCP3008* verbunden ist. Des Weiteren habe ich ein *100 KΩ Potentiometer* mit dem *Kanal 0* des Bausteins verbunden. Das Potentiometer arbeitet wie ein variabler Spannungsteiler, der in Abhängigkeit von der Schleiferposition zwischen den beiden Potentialen *Masse* bzw. V_{DD} vermittelt und das Signal an den analogen Eingang legt.

Ein einfacher Spannungsteiler wird wie folgt mit *2* Widerständen aufgebaut.

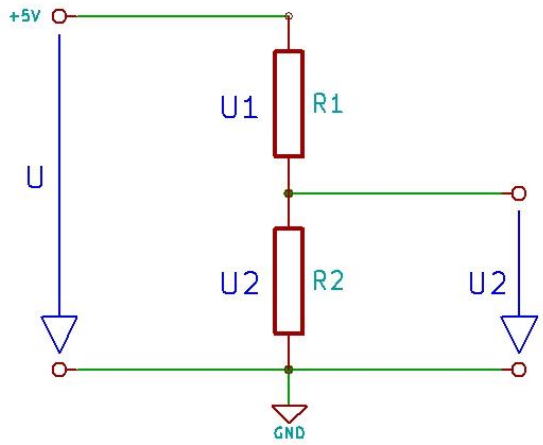


Abbildung 4 Der Spannungsteiler

In Abhängigkeit des Widerstandsverhältnisses wird die Eingangsspannung *U* an den beiden Widerständen *R1* und *R2* aufgeteilt und liegt als Ausgangsspannung *U2* am Widerstand *R2* an. Die Ausgangsspannung berechnet sich wie folgt:

$$U_2 = \frac{R_2}{R_1 + R_2} \cdot U$$

Ein *Potentiometer* kann als variabler Spannungsteiler angesehen werden, der die Widerstände *R1* und *R2* in Abhängigkeit der Schleiferposition verändert.

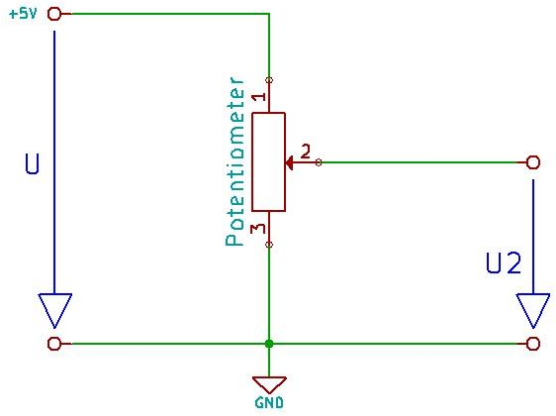


Abbildung 5 Das Potentiometer als variabler Spannungsteiler

Übertragen auf die festen Widerstände *R1* bzw. *R2* verhält sich das Potentiometer wie folgt.

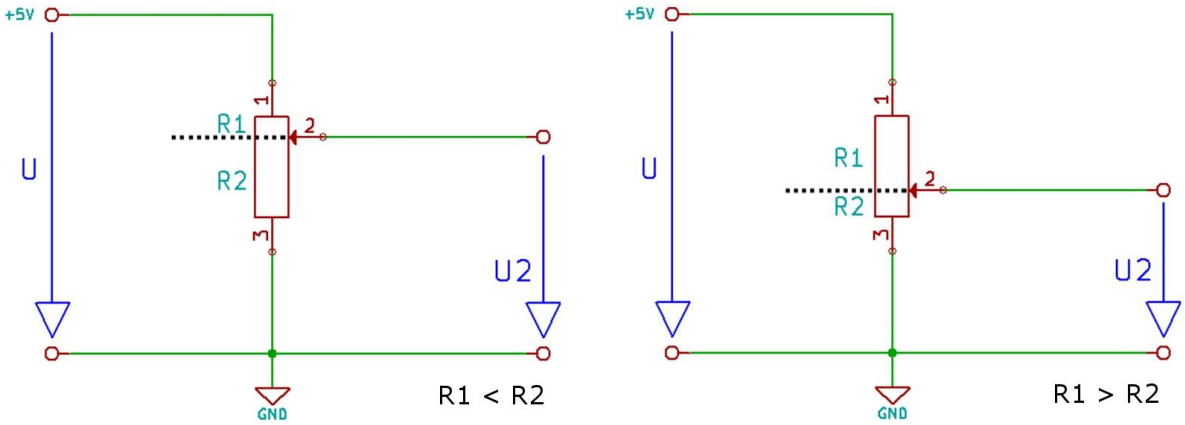


Abbildung 6 Die Widerstandsverhältnisse bei 2 Potentiometerstellungen

Du siehst, dass dir dieses Bauteil eine wunderbare Möglichkeit bietet, eine vorhandene Spannung in den gegebenen Grenzen zwischen *Masse* und V_{DD} zu regeln.

Die Programmierung

Wenn es um die Programmierung geht, dann müssen wir uns auch ein wenig mit dem Timing bzw. der Ansteuerung des *MCP3008* befassen. Das mache ich aber immer zu gegebener Zeit. Ok, dann fangen wir an. Da der *MCP3008* ja über *8* unabhängige analoge Eingänge verfügt, müssen wir dem Chip mitteilen, welchen Eingang wird denn abfragen möchten. Erst dann können wir die angeforderten Daten abrufen. Ich hatte bei der Erklärung der einzelnen Pins des *MCP3008* u.a. erwähnt, dass der *CS*-Pin *LOW-Aktiv* ist. Das bedeutet, dass wir den Baustein mit einem *Masse-Signal* auswählen. Detaillierte Informationen finden sich natürlich im Datenblatt des *MCP3008* (*Siehe Figure 6-1 SPI-Communication with the MCP3004/3008*). Doch bevor wir auf die Details des *MCP3008* eingehen, sollten wir die notwendigen GPIO-Pins des *Raspberry Pi* unter die Lupe nehmen. Du hast den Schaltungsaufbau zwar schon mit *Fritzing* gesehen, doch ich werde die einzelnen Leitungen etwas übersichtlicher darstellen.

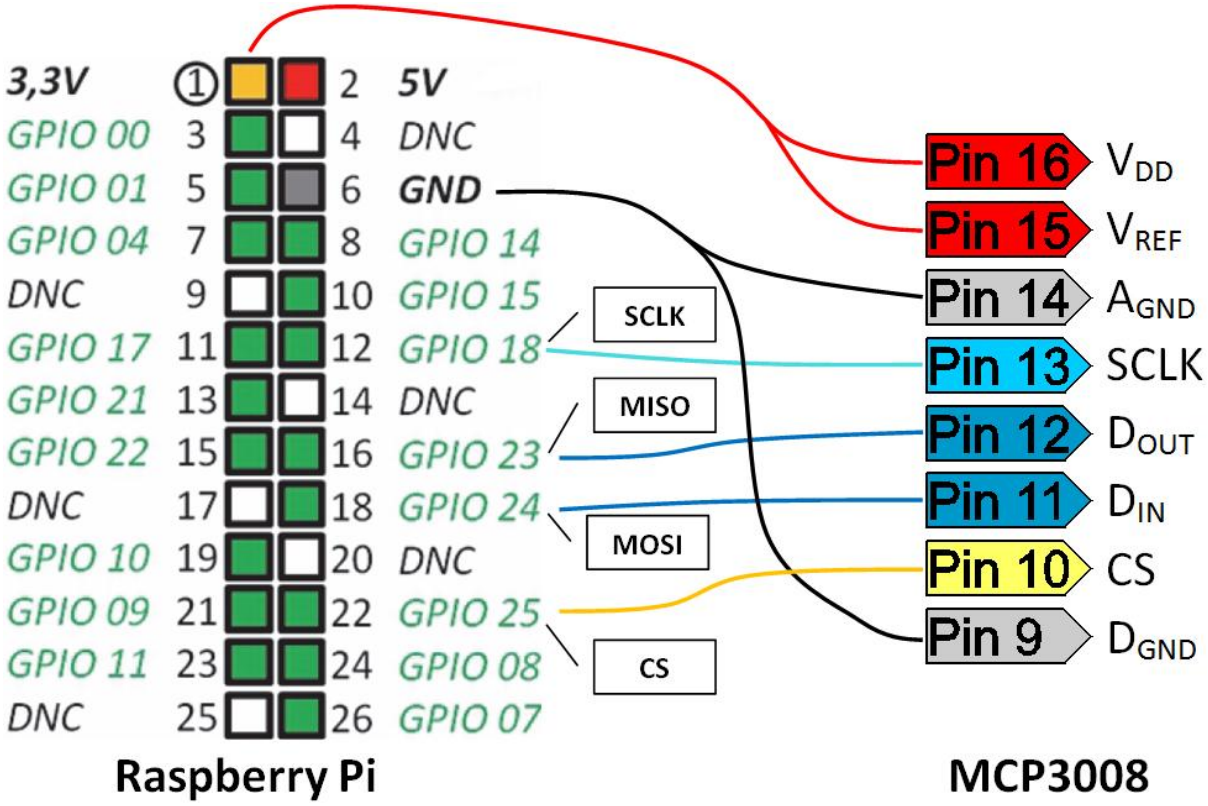


Abbildung 7 Die Verdrahtung zwischen Raspberry Pi und dem MCP3008

Die *4 SPI*-Leitungen

- SCLK
- MISO
- MOSI
- CS

werden über das Programm, was wir gleich schreiben werden, beeinflusst. Es werden sowohl Signale an den *MCP3008* versendet, als auch empfangen.

Hier zeige ich Dir noch kurz den Aufbau, den ich auf meinem Breadboard durchgeführt habe.

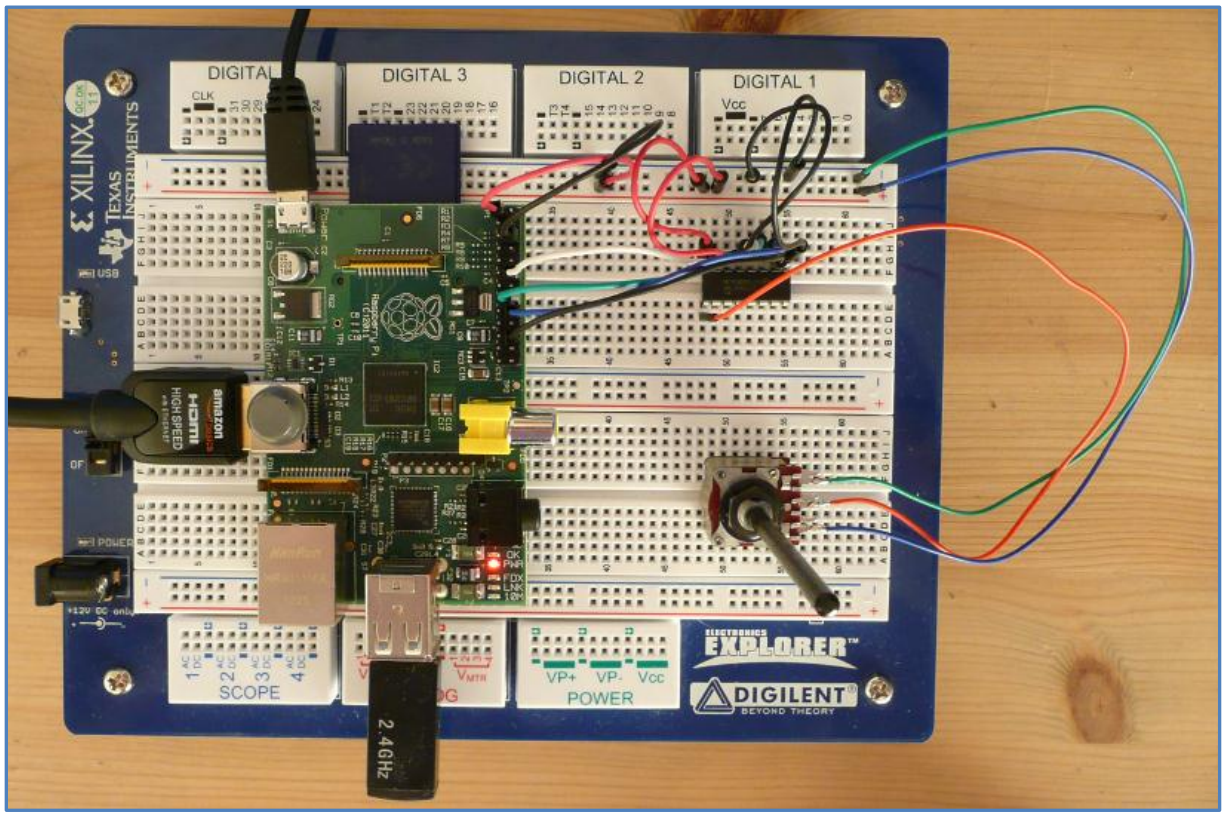


Abbildung 8 Der Schaltungsaufbau auf einem Breadboard

Welche Programmiersprache verwenden wir?

Es ist wohl an der Zeit mit der Information rauszurücken, welche Programmiersprache verwendet wird. Es ist *Python*! Diese Programmiersprache ist sowohl im Skriptingumfeld, als auch bei der Entwicklung von komplexen Stand-Alone-Programmen sehr verbreitet und beliebt. Für den *Raspberry Pi* wurde eigens eine Library entwickelt, die das Programmieren der GPIO-Schnittstelle sehr vereinfacht.

Vorbereitungen

Folgende Schritte sind notwendig, damit Du auf Deinem *Raspberry Pi* in der Programmiersprache *Python* deine Programme entwickeln kannst.

Python-Dev installieren

Falls es nicht schon geschehen ist, musst du *Python-Dev* über die folgende Kommandozeile installieren:

```
# sudo apt-get install python-dev
```

Die Raspberry Pi GPIO-Library installieren

Diese Library findest Du auf der folgenden Seite im Internet.

http://code.google.com/p/raspberry-gpio-python/downloads/list

Du wirst sehen, dass sich dort eine ganze Anzahl von unterschiedlichen Versionen befindet. Für meine Versuche habe ich die Version *0.3.1a* verwendet. Halte Ausschau nach der neuesten Version, denn sie ist sicherlich Fehler bereinigt und bietet von Zeit zu Zeit ein paar neue Features. Hast Du die gewünschte Version in Deinem *Home-Verzeichnis* heruntergeladen, dann liegt sie in der folgenden Form vor:

RPi.GPIO-0.3.1a.tar.gz

Du kannst an der Endung *gz* erkennen, dass die Datei noch *komprimiert* ist. Mit dem folgenden Kommando *dekomprimierst* du die Datei:

```
# gunzip RPi.GPIO-0.3.1a.tar.gz
```

Die einzelnen Dateien bzw. Verzeichnisse sind jedoch dann noch in einer einzigen Datei mit der Endung *tar* zusammengefasst bzw. archiviert. Du musst über das folgende Kommando das Archiv entpacken.

```
# tar –xvf RPi.GPIO-0-3-1a.tar
```

Jetzt kannst Du über das *cd*-Kommando in das neu entstandene Verzeichnis wechseln. Dort befindet sich u.a. eine Datei mit dem Namen *setup.py*. Es handelt sich um eine Installationsdatei von Python, über die Du das Python-GPIO-Paket installieren kannst. Starte über die folgende Befehlszeile die Installation:

```
# sudo python setup.py install
```

Nach der erfolgreichen Installation können wir unmittelbar mit der Programmierung beginnen.

Das Python-Programm

Hinsichtlich der Programmierung in *Python* gibt es die unterschiedlichsten Entwicklungsumgebungen, die du nutzen kannst. In meinem Buch über den *Raspberry Pi* habe ich u.a. auch *Stani's Python Editor* – kurz *SPE* genannt – verwendet. Du kannst ihn mit der folgenden Befehlszeile installieren:

```
# sudo apt-get install spe
```

Das ist schon ein interessantes und mächtiges Werkzeug, doch auch der unter *Raspian* (*Debian Wheezy*) vorinstallierten Texteditor *Nano* ist sicherlich einen Blick wert. Es ist aber in jedem Fall zu bedenken, dass das Python-Skript mit *Root-Rechten* gestartet werden muss. Also angenommen, du hast das Skript *read_mcp3008.py* genannt, dann musst Du es wie folgt starten:

```
# sudo python read_mcp3008.py
```

Ich werde in kleinen Schritten vorgehen und jeden zusammenhängenden Skriptblock detailliert erläutern. Um ein lauffähiges Skript zu erhalten, positioniere einfach alle einzelnen Abschnitte hintereinander.

```
1  import time
2  import RPi.GPIO as GPIO
3
4  GPIO.setmode(GPIO.BCM)
5
6  HIGH = True  # HIGH-Pegel
7  LOW  = False # LOW-Pegel
```

Zu Beginn müssen wir *2* Libraries einbinden, damit unser Skript auch ohne Probleme läuft. Die *time*-Library wird später genutzt, um über den *sleep*-Befehl eine Pause im Ablauf einzulegen. Die *RPi.GPIO*-Library kapselt die komplette Funktionalität hinsichtlich der Ansteuerung der *GPIO-Schnittstelle*. Mit Hilfe der *setmode*-Methode teilen wir der Library mit, dass wir anstelle der *Pin-Nummern*, die *GPIO-Nummern* in der Programmierung verwenden möchten. In der *Abbildung 7* ist auf der linken Seite die Pinbelegung der *GPIO-Schnittstelle* abgebildet. Es gibt aber Pins, die eine *GPIO-Funktionalität* besitzen, die ebenfalls eine Nummerierung haben. Nehmen wir doch einfach *Pin 12*. Er ist mit der Bezeichnung *GPIO18* versehen und dient in unserem Fall als *Clock-Pin*. Ich möchte jedoch in meinem Skript die *GPIO-Bezeichnungen* verwenden. Deshalb füge ich die folgende Zeile zu Beginn ein:

```
GPIO.setmode(GPIO.BCM)
```

Statt mit *True* bzw. *False* hinsichtlich der logischen Pegel zu arbeiten, habe ich diese Werte in den Zeilen *6* und *7* den sprechenderen Variablen *HIGH* und *LOW* zugewiesen. Das bedeutet entweder *3,3V* oder *0V*. Um den anliegenden analogen Wert des *MCP3008* abzurufen, habe ich alle notwendigen Befehle in eine Funktion gepackt, die ich im Anschluss kontinuierlich aufrufen kann. Sie lautet

```
10  def readAnalogData(adcChannel, SCLKPin, MOSIPin, MISOPin, CSPin):
```

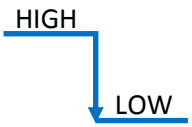
und ihr müssen zur Abarbeitung gewisse Werte (*Argumente*) übergeben werden.

- *adcChannel*
- *SCLK*
- *MOSIPin*
- *MISOPin*
- *CSPin*

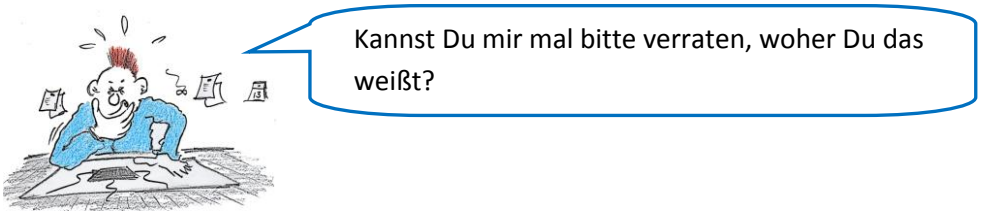
Auf diese Weise können wir die Kodierung recht flexibel gestalten und die Werte ggf. ohne große Probleme anpassen. Zu aller erst müssen wir bestimmte Pegel vorbereiten.

```
11      # Pegel vorbereiten
12      GPIO.output(CSPin,   HIGH)
13      GPIO.output(CSPin,   LOW)
14      GPIO.output(SCLKPin, LOW)
```

Der *Chip-Select-Pin* ist *LOW-Aktiv*, was bedeutet, dass er auf einen Pegelwechsel von *HIGH* auf *LOW* reagiert.



Genau das erreichen wir über die Zeilen *12* und *13*.



Nun, erraten kann man so etwas sicherlich nicht. Da hilft nur ein Blick in das Datenblatt des betreffenden Bausteins.

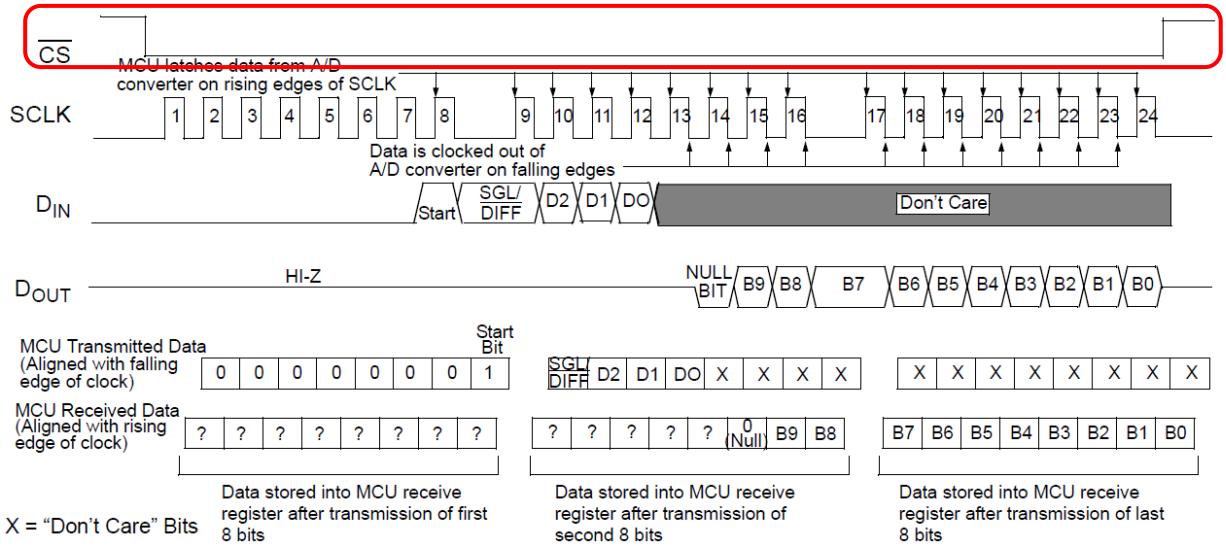
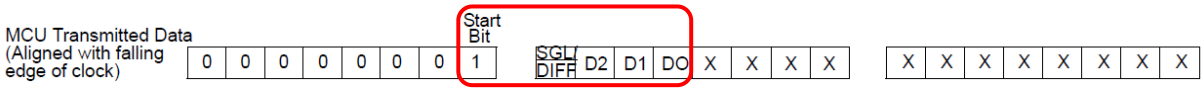


Abbildung 9 SPI-Kommunikation mit dem MCP3008 (Quelle: Datenblatt des Herstellers)

In der obersten Zeile findest du das *Chip-Select-Signal*, was bei der startenden Messung von *HIGH* auf *LOW* geht. Den *Clock-Pin* (*SCLK*) wird in Vorbereitung auf *LOW*-Pegel gesetzt. Jetzt geht es darum, einen Befehl an den Baustein zu verschicken, damit dieser bereit dazu ist, einen bestimmten *A/D-Kanal* auszulesen und den Wert zurückzuliefern. Ein Blick in das Datenblatt gibt Aufschluss über die Bedeutung der einzelnen Bits.



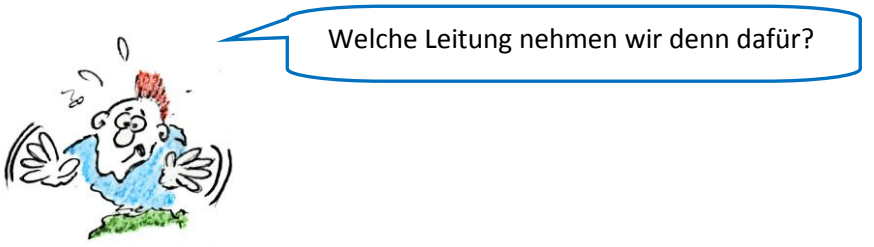
Der Code dazu lautet:

```
sendcmd = adcChannel
sendcmd |= 0b00011000
```

ADC-Kanal

Start-Bit SGL

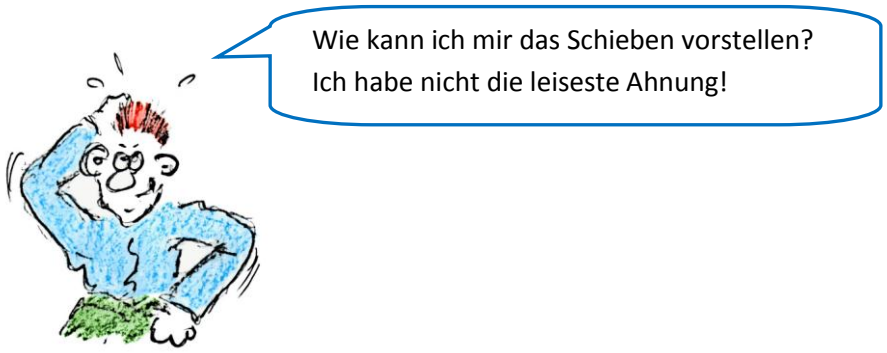
Jetzt geht es darum, die betroffenen Bits auszuwerten. Es handelt sich um *5 Bits* (von rechts gesehen), die die Informationen beinhalten. Diese Informationen müssen an den *MCP3008* übermittelt werden.



Es handelt sich um die *MOSI*-Leitung (*Master-Out-Slave-In*). Dazu schauen wir uns den folgenden Code an.

```
19      # Senden der Bitkombination (Es finden nur 5 Bits Beruecksichtigung)
20      for i in range(5):
21          if (sendcmd & 0x10): # (Bit an Position 4 pruefen. Zaehlung beginnt bei 0)
22              GPIO.output(MOSIPin, HIGH)
23          else:
24              GPIO.output(MOSIPin, LOW)
25          # Negative Flanke des Clocksignals generieren
26          GPIO.output(SCLKPin, HIGH)
27          GPIO.output(SCLKPin, LOW)
28          sendcmd <<= 1 # Bitfolge eine Position nach links schieben
```

In der *for*-Schleife wird der Inhalt der Variablen um *eine Position* nach links verschoben, um dann das Bit an Position *4* (Zählung beginnt bei *0*) zu überprüfen.



Das ist recht simpel, wenn du dir die Werte auf Bitebene anschaust.

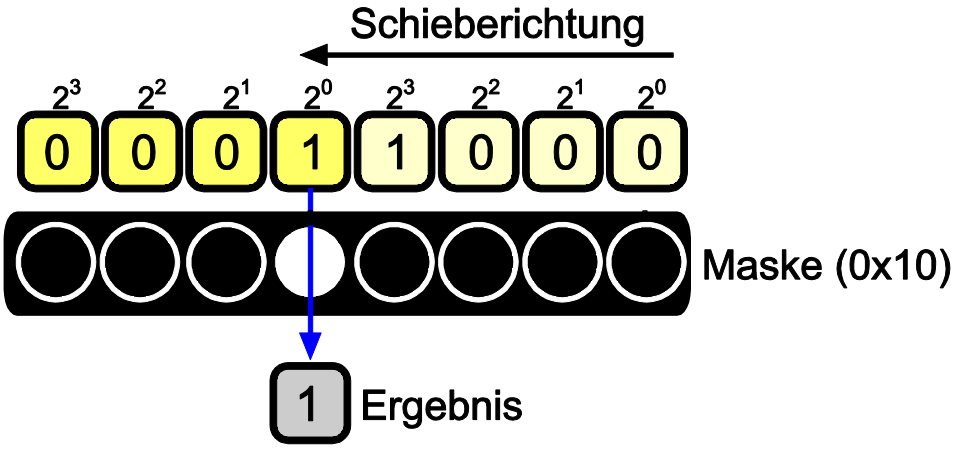


Abbildung 10 Der Schiebevorgang

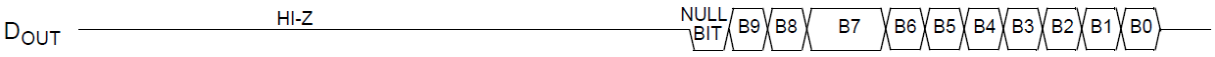
Die Variable *sendcmd* wird mit dem Wert *0x10 UND*-Verknüpft und jedes Ergebnis, was von *0* verschieden ist, wird als *wahr* angesehen. In Abhängigkeit von dieser Bewertung wird entweder ein *HIGH*- oder ein *LOW*-Pegel auf die *MOSI*-Leitung geschickt.

Jede dieser Aktionen muss mit einem *HIGH-LOW* Pegelwechsel, also einer abfallenden Flanke, auf der *Clock*-Leitung quittiert werden.

Schau her:

```
25          # Negative Flanke des Clocksignals generieren
26          GPIO.output(SCLKPin, HIGH)
27          GPIO.output(SCLKPin, LOW)
28          sendcmd <<= 1 # Bitfolge eine Position nach links schieben
```

Am Schluss erfolgt die schon beschriebene Schiebeaktion nach links. Kommen wir jetzt zum Lesen der Daten, die der *MCP3008* als Antwort zurücksendet. Werfen wir dazu einen erneuten Blick auf das Impulsdiagramm des Herstellers.



Jeder Lesezyklus wird wieder mit einem *negativen* Flankenwechsel auf der *Clock*-Leitung quittiert, was du in den Zeilen *33* und *34* sehen kannst. Das soweit zur Einleitung.

```
30      # Empfangen der Daten des ADC
31      adcvalue = 0 # Ruecksetzen des gelesenen Wertes
32      for i in range(11):
33          GPIO.output(SCLKPin, HIGH)
34          GPIO.output(SCLKPin, LOW)
35          adcvalue <<= 1 # 1 Postition nach links schieben
36          if(GPIO.input(MISOPin)):
37              adcvalue |= 0x01
```

Jetzt müssen die Ergebnisdaten wieder Bit für Bit nach links geschoben werden (*Zeile 35*), um der Variablen *adcvalue* zugewiesen zu werden. Das erfolgt über eine bitweise *ODER*-Verknüpfung an der niederwertigsten Stelle, wo quasi immer ein Bit eingeschoben wird. Ob das eine *0* oder eine *1* ist, das hängt von dem gelesenen Wert auf der *MISO*-Leitung ab. Zum Lesen wird hier die *input*-Methode der *GPIO-Library* verwendet. Die ganze Aktion wird *11-mal* durchgeführt, um alle Bits zu erreichen. Am Ende wird eine kurze Pause von ½ Sekunde eingelegt und der berechnete Wert an den Aufrufer der Funktion zurückgeliefert.

```
38      time.sleep(0.5)
39      return adcvalue
```

Kommen wir abschließend noch zum Hauptprogramm, wo einige Variable definiert werden und die Funktion aufgerufen wird.

```
# Variablendefinition
ADC_Channel = 0  # Analog/Digital-Channel
SCLK        = 18 # Serial-Clock
MOSI        = 24 # Master-Out-Slave-In
MISO        = 23 # Master-In-Slave-Out
CS          = 25 # Chip-Select

# Pin-Programmierung
GPIO.setup(SCLK, GPIO.OUT)
GPIO.setup(MOSI, GPIO.OUT)
GPIO.setup(MISO, GPIO.IN)
GPIO.setup(CS,   GPIO.OUT)

while True:
    print readAnalogData(ADC_Channel, SCLK, MOSI, MISO, CS)
```

Das Python Programm wird über die folgende Kommandozeile gestartet.

 # sudo python read_mcp3008.py

Jedenfalls war das mein Name. Falls du einen anderen verwendest, passe ihn entsprechend an. Nachfolgend siehst du das komplette Python-Programm.

```
import time
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)

HIGH = True  # HIGH-Pegel
LOW  = False # LOW-Pegel

# Funktionsdefinition
def readAnalogData(adcChannel, SCLKPin, MOSIPin, MISOPin, CSPin):
    # Pegel vorbereiten
    GPIO.output(CSPin,   HIGH)
    GPIO.output(CSPin,   LOW)
    GPIO.output(SCLKPin, LOW)

    sendcmd = adcChannel
    sendcmd |= 0b00011000 # Entspricht 0x18 (1:Startbit, 1:Single/ended)

    # Senden der Bitkombination (Es finden nur 5 Bits Beruecksichtigung)
    for i in range(5):
        if (sendcmd & 0x10): # (Bit an Position 4 pruefen. Zaehlung beginnt bei 0)
            GPIO.output(MOSIPin, HIGH)
        else:
            GPIO.output(MOSIPin, LOW)
        # Negative Flanke des Clocksignals generieren
        GPIO.output(SCLKPin, HIGH)
        GPIO.output(SCLKPin, LOW)
        sendcmd <<= 1 # Bitfolge eine Position nach links schieben

    # Empfangen der Daten des ADC
    adcvalue = 0 # Ruecksetzen des gelesenen Wertes
    for i in range(11):
        GPIO.output(SCLKPin, HIGH)
        GPIO.output(SCLKPin, LOW)
        adcvalue <<= 1 # 1 Postition nach links schieben
        if(GPIO.input(MISOPin)):
            adcvalue |= 0x01
    time.sleep(0.5)
    return adcvalue

# Variablendefinition
ADC_Channel = 0  # Analog/Digital-Channel
SCLK        = 18 # Serial-Clock
MOSI        = 24 # Master-Out-Slave-In
MISO        = 23 # Master-In-Slave-Out
CS          = 25 # Chip-Select

# Pin-Programmierung
GPIO.setup(SCLK, GPIO.OUT)
GPIO.setup(MOSI, GPIO.OUT)
GPIO.setup(MISO, GPIO.IN)
GPIO.setup(CS,   GPIO.OUT)

while True:
    print readAnalogData(ADC_Channel, SCLK, MOSI, MISO, CS)
```

Eine kleine Verbesserung des Python-Programms

Es kann vorkommen, dass das Potentiometer an einer bestimmten Position steht, wo kein eindeutiger Wert geliefert wird. Das hat die Auswirkung, dass der angezeigte analoge Wert ständig um einen Kernwert herum schwankt. Dieses Verhalten wird *Jittering* genannt. Du kannst dieses Phänomen kompensieren bzw. eliminieren, indem du einen Toleranzwert festlegst. Erst, wenn sich die gemessenen Werte um diesen Toleranzwert voneinander unterscheiden, werden sie zur Anzeige gebracht. Wir brauchen dazu zwei neue Variable. Die erste definiert den Toleranzwert und die zweite den zuletzt gemessenen Wert. Also z.B.

- threshold
- prev_value

Ich habe, da es wohl in vereinzelten Fällen Probleme mit der Ausführung des Python-Programms gegeben hat, eine *Shebang*-Zeile (auch *Magic-Line* genannt) an den Anfang des Programms bzw. Skriptes hinzugefügt. Sie gibt Aufschluss über den Pfad des verwendeten Interpreters und wird durch die Zeichenkombination *#!* Eingeleitet. Das ist bei *Unix/Linux* ggf. notwendig, findet jedoch bei *Windows* keine Verwendung. Das komplette Programm habe ich auf meiner Download-Seite dem Paket *MCP3008Python* hinzugefügt und *read_mcp3008_onlychangedvalues.py* genannt. Hier kommen die Anpassungen im Detail. Zuerst die beiden neuen Variablen:

```
13  threshold = 3  # Toleranz-Wert. Erst, wenn der gemesssene analoge Wert sich um
14                 # den angegebenen Toleranzwert vom vorherigen Wert unterscheidet,
15                 # wird er in der Anzeige beruecksichtigt
16  prev_value = 0 # Speichert den vorherigen analogen Wert
```

Dann der Code, der die Auswertung vornimmt, ob ein Wert angezeigt werden soll:

```
64  while True:
65      actual_value = readAnalogData(ADC_Channel, SCLK, MOSI, MISO, CS)
66      # Erst, wenn sich der neue Wert vom alten um den Toleranzwert unterscheidet,
67      # wird er zur Anzeige gebracht. Das verhindert einen sogenannten Jitter-Effekt.
68      # Das bedeutet die Verhinderung eines Zitters in der Anzeige.
69      if(abs(actual_value - prev_value) > threshold):
70          print actual_value
71      prev_value = actual_value # Vorherigen Wert sichern
```

In Zeile *69* bilde ich die *Differenz* vom gerade gemessenen Wert *actual_value* und dem zuvor gemessenen Wert *prev_value*, der in Zeile *71* gebildet wird. Diese *Differenz* muss größer der festgelegten Toleranz sein, die in der Variablen *threshold* aus Zeile *13* definiert wurde.

Ok, das habe ich verstanden! Doch in *Zeile 69* steht noch ein Wörtchen, dass mir nicht geläufig ist. Was bedeutet *abs*? Ein *Anti-Blockier-System* wird es ja wohl nicht sein.

Opps! Stimmt, das hatte ich vergessen. Es handelt sich dabei um die *abs*-Funktion, die den *Absolutwert* ermittelt. Diese Funktion, die auch *Betragsfunktion* genannt wird, liefert immer den positiven Wert unabhängig des Vorzeichens zurück. Eine positive Zahl bleibt positiv, eine negative Zahl wird positiv. Hier der Kurvenverlauf, der das Verhalten vielleicht ein wenig deutlicher macht.

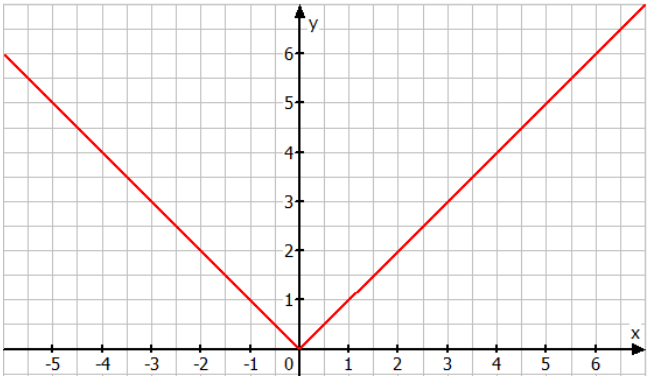


Abbildung 11 Der Verlauf der Betragsfunktion

Wir benötigen diese Funktion in unserem Programm, da die ermittelte Differenz in Abhängigkeit des aktuellen bzw. vorher gemessenen Wertes einmal *positiv*, einmal *negativ* sein kann. Wir möchten jedoch immer den absoluten Wert mit dem positiven Wert der *Toleranz* vergleichen, denn andernfalls würde das nicht immer funktionieren.

Unser abgeändertes Programm liefert jetzt nur noch Werte an das *Terminal-Fenster*, wenn sich die Werte entsprechend den eben genannten Voraussetzungen geändert haben.

Bezugsquellen

Den *MCP3008* kannst du z.B. unter den folgenden Adressen beziehen:

http://www.kessler-electronic.de/Halbleiter/integrierte_Schaltkreise/analog/M__R/MCP/MCP3008-I/P_i606_11106_0.htm

https://hbe-shop.de/10BIT-ADC27V8CHSPI16DIP-Typ-MCP3008-I-P

Schlusswort

Jetzt wünsche ich dir viel Spaß beim Experimentieren und ich würde mich freuen, wenn du von Zeit zu Zeit einen Blick auf meine Internetseite werfen würdest. Dort findest du sicherlich ein paar interessante *AddOns* zu meinen verschiedenen Themen bzw. Büchern.

Erik Bartmann

www.erik-bartmann.de

http://www.oreilly.de/catalog/raspberrypiger/

http://www.oreilly.de/catalog/elekarduinobasger/

http://www.oreilly.de/catalog/processingger/