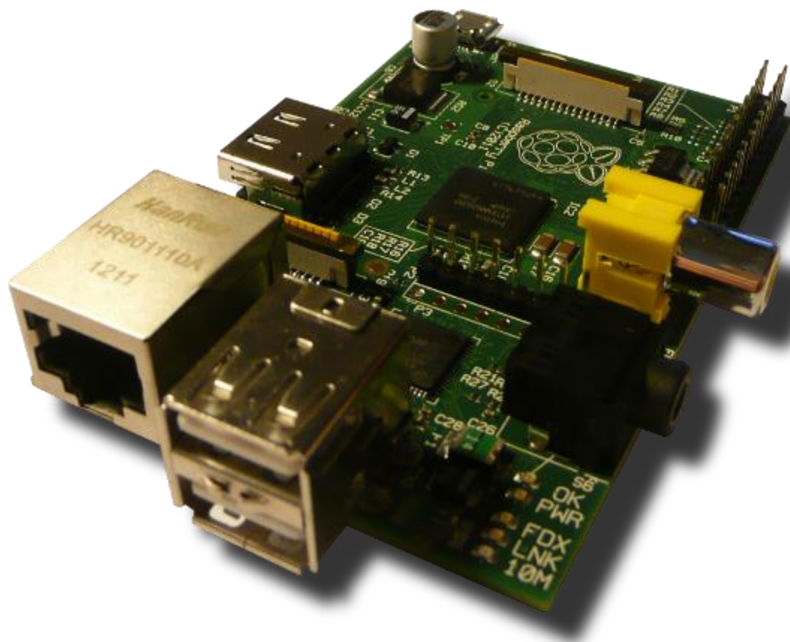
PiMeUp

Raspberry Pi



Raspberry Pi

AddOn

Der Port-Expander MCP23S17 (Teil 2)

Version 1.0,
19.10.2012

© by Erik Bartmann

www.erik-bartmann.de

Worum geht's?

Hallo zusammen,

in diesem *RasPi-AddOn - Teil 2* - möchte ich noch tiefer in die Thematik zur Ansteuerung des Baustein *MCP23S17* eingehen. Es handelt sich hierbei – wie schon in *Teil 1* ausführlich erwähnt - um einen sogenannten *Port-Expander*, den ihr hier auf dem folgenden Bild seht.

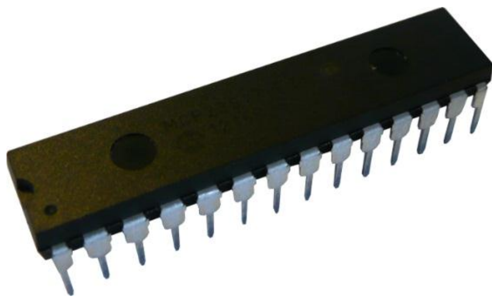


Abbildung 1 Der MCP23S17

In *Teil 1* hast du gesehen, wie die Ansteuerung der angeschlossenen Leuchtdioden an *Port-B* funktioniert. Dabei hast du die einzelnen Pins des Ports lediglich als *Ausgänge* betrieben. Sicherlich möchtest du auch einmal sehen, wie z.B. der Status eines oder mehrerer Taster abgefragt werden kann. Das ist ebenfalls möglich. Jeder einzelne Pin des *Ports A* bzw. *B* kann einzeln angesprochen und individuell als *Ein-* oder *Ausgang* programmiert werden. Damit die ganze Angelegenheit etwas interessanter wird, habe ich eine flexible Schaltung nicht auf einem *Breadboard*, sondern auf einer *Lochrasterplatine* untergebracht. So habe ich die Möglichkeit, über flexible Steckbrücken die unterschiedlichsten Verbindungen zu den Ports und den Leuchtdioden bzw. Tastern herzustellen. Die Verbindung zum *Raspberry Pi* erfolgt über ein geeignetes Flachbahnkabel, was die ganze Sache schon etwas professioneller gestaltet.

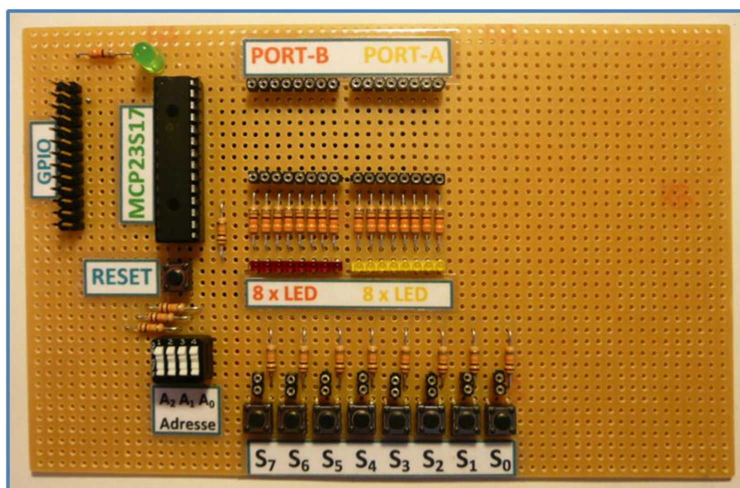


Abbildung 2 Das MCP23S17 Prototyping-Board

Über die aufgelöteten Buchsenleisten kannst du prima die unterschiedlichsten Verbindungen herstellen, um so deine Schaltung bzw. deine Programmierung zu testen.

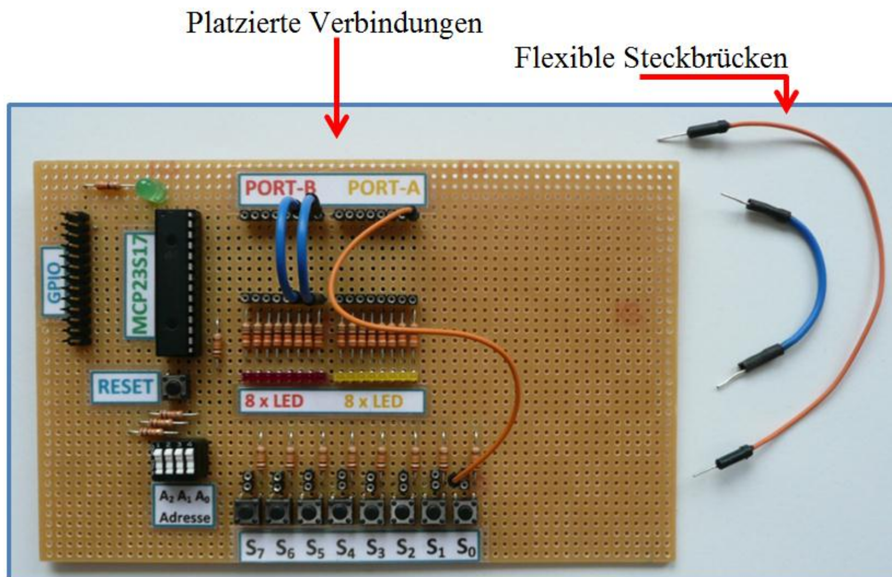


Abbildung 3 Das MCP23S17-Prototyping-Board mit flexiblen Steckbrücken

Auf diese Weise habe ich zwei LEDs mit *Port-B* und einen Taster S_0 mit *Port-A* verbunden. Damit du nicht immer bezüglich der Pinbelegung des *MCP23S17* in das *AddOn Teil 1* schauen musst, zeige ich es dir an dieser Stelle noch einmal.

Der MCP23S17

Die Pinbelegung des Port-Expanders schaut wie folgt aus:

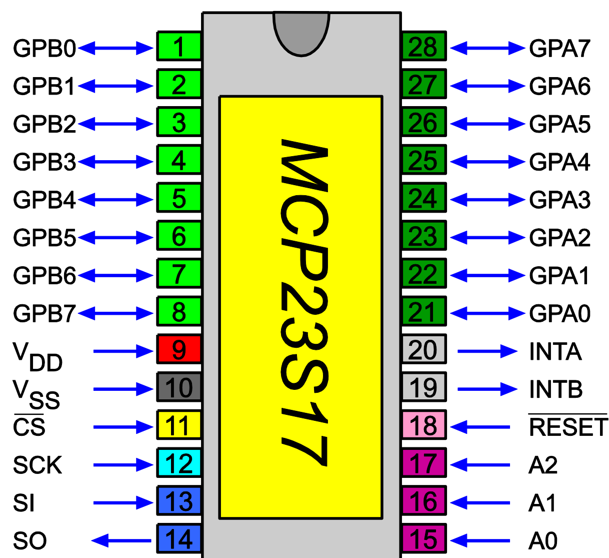
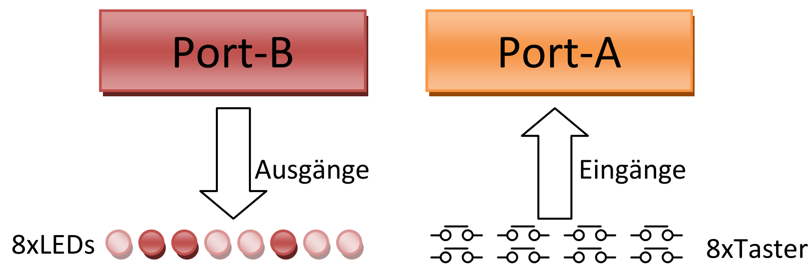


Abbildung 4Die Pinbelegung des MCP 23S17

Ich möchte in diesem *AddOn* das *Prototyping-Board* verwenden und alle *16 Pins* der beiden *Ports A* und *B* verwenden.

Was hast du denn im Einzelnen vor? Kannst du mir das bitte einmal erklären!

Habe ich dich schon jemals im Unklaren darüber gelassen, was ich als nächstes vorhabe? Na also! Also schau her. Die *8* einzelnen *LEDs* möchte ich an *Port-B* belassen und nun aber zusätzlich *8 Taster* an den *Port-A* anschließen, um deren Status dort abzufragen. Das schaut dann wie folgt aus:



Die Anzeige der LED-Patterns aus dem *Teil 1* soll erst einmal beibehalten werden, so dass du siehst, dass die Abfrage der *Taster* unabhängig davon ist. Gleichzeitig werden die *8 Taster* abgefragt und deren Status in einem *Terminal-Fenster* angezeigt. Schauen wir uns dazu erst einmal den Schaltplan genauer an.

Der Schaltplan

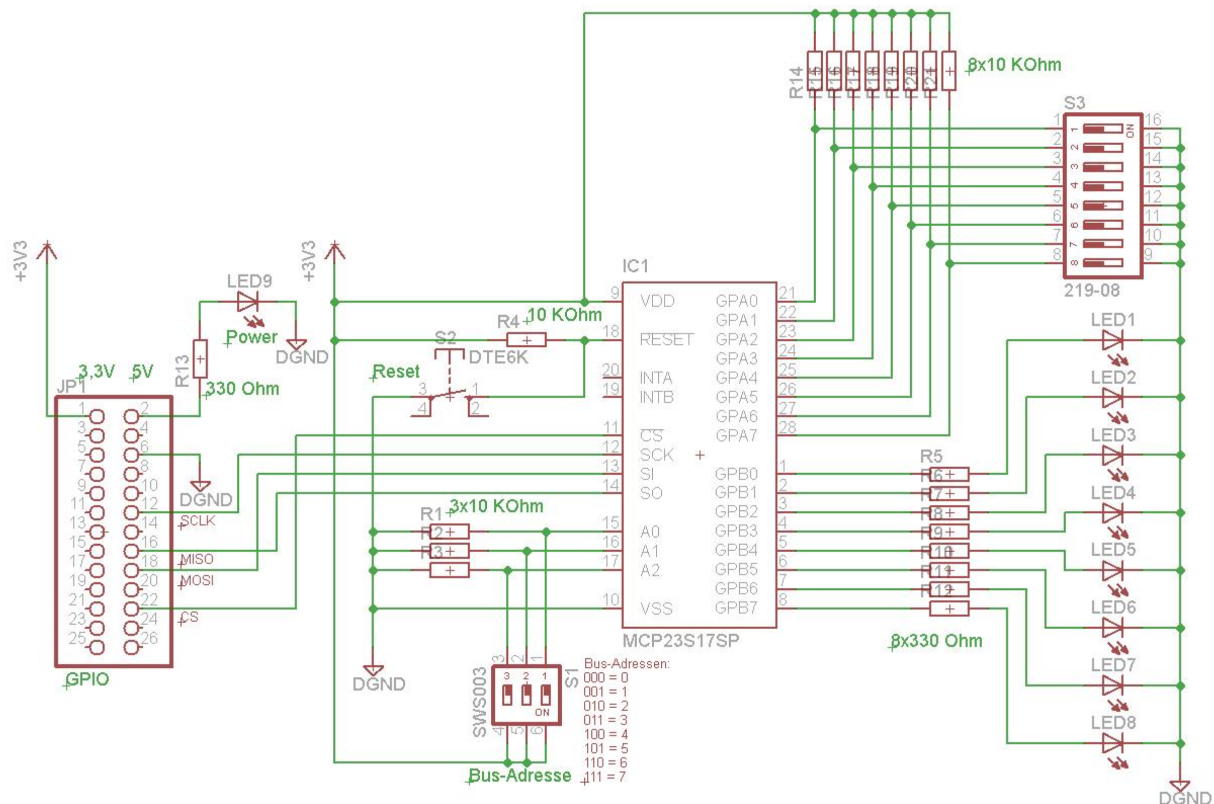


Abbildung 5 Der Schaltplan

Aus Platzgründen habe ich die Eingabe, die normalerweise über *Taster* stattfindet, hier im Schaltplan mit *DIP-Switches* realisiert. Passe das einfach nach deinen Wünschen an.

Da ist eine Sache, die ich im Moment noch nicht so richtig schnalle! Da befindet sich sowohl an den *Bus-Eingängen*, als auch bei den *Port-A* Eingängen an jedem Pin ein Widerstand.

Ok, das hätte ich fast vergessen. Dafür muss ich ein wenig ausholen. In der Digitaltechnik gibt es in Abhängigkeit vom verwendeten Spannungspegel, also z.B. *5V* oder wie hier in den Beispielen *3,3V*, fest definierte *Logik-Pegel*.

Es wird in diesem Fall auch zwischen *Eingangs*- bzw. *Ausgangssignalen* unterschieden. *5V* entspricht in der Regel *TTL*, wobei *3,3V CMOS* ist. Schau her:

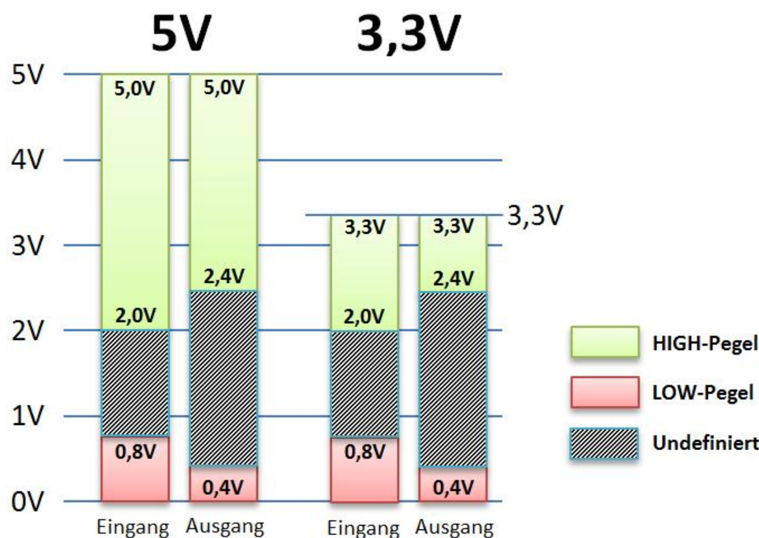


Abbildung 6 Die unterschiedlichen Logikpegel bei TTL und 3,3V - CMOS

Wenn nun ein Eingang eines Logikbausteins z.B. mit V_{SS} *(Masse)* verbunden ist, dann ist sein Logikpegel sicherlich *LOW*, ist er mit V_{DD} verbunden, dann befindet er sich auf *HIGH*-Pegel. Aus Sicherheitsgründen und hinsichtlich bestimmter Toleranzen ist den Logikpegeln nicht ein fester Wert, sondern einen *Bereich* zugeordnet, wie du dass auch im Diagramm erkennen kannst. Damit es aber an der Grenzschicht zwischen *LOW* und *HIGH* nicht zu Sprüngen kommt, existiert eine Grauzone, die keinen definierten Pegel besitzt. Sie sollte auf keinen Fall durchschritten werden, denn dann ist der Pegel *undefiniert* und das ist unter allen Umständen zu vermeiden. Kommen wir nun zu einem Problem, wenn du z.B. einen Taster oder Schalter mit einem Eingang verbindest, um dort ein entsprechendes Signal oder besser ausgedrückt einen definierten Logikpegel anliegen zu lassen. Ist der Taster auf der einen Seite mit dem Eingang des Bausteins verbunden und auf der anderen Seite mit der Spannungsquelle und ist er dann auch noch geschlossen, dann liegt ein definierter *HIGH*-Pegel vor. Was passiert aber, wenn der Taster offen ist? Der Eingang ist dann nicht mit der Spannungsquelle verbunden. Wir haben das Problem, dass der Eingang quasi in der Luft hängt. Er ist mit keinem definierten Potential verbunden. Weder mit *Masse* noch mit dem *positiven Pol* der Spannungsquelle. Er ist offen für jegliche Art von Einstrahlungen von außen. Das kann z.B. eine statische Aufladung einer Person sein, die sich in der unmittelbaren Nähe der Schaltung aufhält oder der Funksender eines Mobiltelefons, der sein Signal in alle Richtungen aussendet. All diese Störeinflüsse können dazu führen, dass sich die Schaltung nicht so verhält, wie es eigentlich beabsichtigt war. Aus diesem Grund hat man sich schaltungstechnisch etwas einfallen lassen.

Sehen wir uns dazu die folgende Schaltung an.

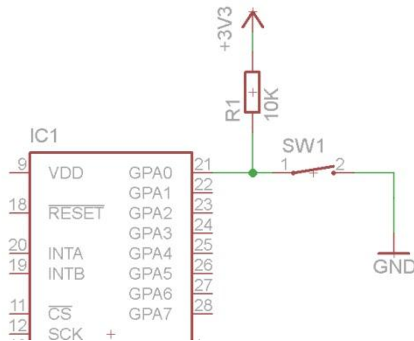


Abbildung 7 Der Pullup-Widerstand am Eingang des Port-Expanders

Es gibt dazu zwei unterschiedliche Ansätze, wovon ich hier erst einmal einen zeige. Der *Eingang* des *Port-Expanders* ist über einen sogenannten *Pullup*-Widerstand mit der Spannungsquelle verbunden. Bei offenem Schalter bekommt dieser Eingang einen definierten Spannungswert von *3,3V* geliefert und es liegt demnach ein *HIGH*-Pegel vor. Schließen wir nun den Schalter, wird das *Masse*-Potential an den Eingang geführt und dieser registriert einen *LOW*-Pegel. Wir haben also in beiden Fällen einen definierten Pegel am Eingang anliegen. Auf diese Weise schaffen wir klare Pegel-Verhältnisse, die die Schaltung kontrolliert arbeiten lassen. Zahlreiche *Mikrocontroller* und auch dieser *Port-Expander* besitzen die Möglichkeit, interne *Pullup-Widerstände* zu aktivieren, die in gleicher Weise arbeiten. Das soll jedoch im Moment nicht unser Thema sein. Bedenke beim Einsatz eines *Pullup*-Widerstandes jedoch die Logik-Umkehrung. Ist der Schalter offen, liegt ein *HIGH*-Pegel vor, ist er geschlossen, ein *LOW*-Pegel. Du erreichst eine Umkehrung der Verhältnisse, verwendest du einen *Pulldown*-Widerstand, der mit *Masse* verbunden ist. Der Schalter liefert dann die Spannungsversorgung an den Eingang. Teste diese beiden Ansätze in unterschiedlichen Experimenten. Das Prinzip der *Pulldown*-Widerstände habe ich übrigens bei den *Busadressen* zur Anwendung gebracht. Schaue die die Verdrahtung dort einmal an. Dann verstehst du auch die beiden unterschiedlichen Ansätze. Nun ist es soweit, dass wir uns der Programmierung zuwenden können.

Die Programmierung

Welche Programmiersprache verwenden wir?

Natürlich werden wir wieder in der Programmiersprache *Python 2.7* mit der *GPIO*-Library-Version *RPi.GPIO 0.4.1a* programmieren. Ich habe es zwar in *Teil 1* schon beschrieben, doch es kann nicht schaden, wenn ich hier nochmals die Installationsschritte zeige.

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 – wie schon erwähnt – diesmal die Version *0.4.1a* verwendet. Hast Du die gewünschte Version in Deinem *Home-Verzeichnis* heruntergeladen, dann liegt sie in der folgenden Form vor:

RPi.GPIO-0.4.1a.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-0.4.1a.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-0-4-1a.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 *control_mcp23s17_read_write.py* genannt, dann musst Du es wie folgt starten:

```
# sudo python control_mcp23s17_read_write.py
```

Ich werde in kleinen Schritten vorgehen und jeden zusammenhängenden Skriptblock detailliert erläutern. Das komplette Skript findest du im Downloadbereich auf meiner Internetseite.

Vorbetrachtung

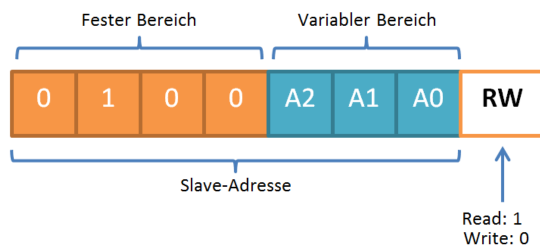
Da wir jetzt nicht nur Werte an den *Port-Expander* versenden, sondern auch etwas von ihm empfangen möchten, noch ein paar einleitende Worte. Ich hatte schon in *Teil 1* angekündigt, dass die Programmierung dort recht statisch ist und das ist für den Teil auch vollkommen ok so. Doch nun passen wir den Code entsprechend an. Rückblickend noch mal ein paar Grundlagen. Bisher hast du lediglich den Informationsfluss in einer einzigen Richtung betrieben. Vom *Raspberry Pi* zum Port-Expander *MCP23S17*, um dort die angeschlossenen LEDs zum Blinken zu bringen.

Wir haben lediglich die *MOSI*-Leitung zum *Port-Expander* genutzt, die dort in den *SI*-Pin (*Serial-Data-In*) geleitet wurde. Zum Ansteuern der LEDs war das vollkommen ausreichend.

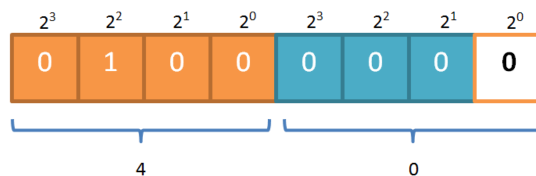
Das reicht uns jetzt aber nicht mehr, denn es sollen die angeschlossenen *Taster* abgefragt werden. Dazu müssen Informationen vom *Port-Expander* zum *Raspberry Pi*, also in die entgegengesetzte Richtung, übermittelt werden.

Die bisher ungenutzte *MISO*-Leitung des *Raspberry Pi* empfängt Daten vom *DO*-Pin (*Serial-Data-Out*) des *Port-Expanders*. Ich denke, dass das als kurze Einleitung genügen mag.

Sehen wir uns an dieser Stelle das *Control-Byte* noch einmal genauer an.



Von links gesehen, haben wir dort den *festen* und den *variablen* Bereich, der auf der rechten Seite mit dem *R/W-Bit* abschließt. Dieses Bit hatten wir in *Teil 1* fest mit dem Wert *0* versehen, da wir nur schreiben wollten. Daraus ergab sich ein konstanter Wert von *0x40*.



Um das für unser kommendes Experiment flexibler zu gestalten, lassen wir zwar den Wert so bestehen, passen ihn aber in Hinblick auf die anstehenden Operationen (*Schreiben + Lesen*) später an.

Wie wollen wir denn einen bestehenden Wert anpassen, wenn du ihn aber eigentlich nicht anpassen willst? Das ist doch widersprüchlich.

Du hast ja vollkommen Recht! Der ursprüngliche Wert wird so belassen, doch wir verknüpfen ihn über eine *bitweisen Operation* mit einem anderen Wert. Hier meine Deklarationszeilen, die ich um zwei weitere Zeilen (*25* und *26*)ergänz habe:

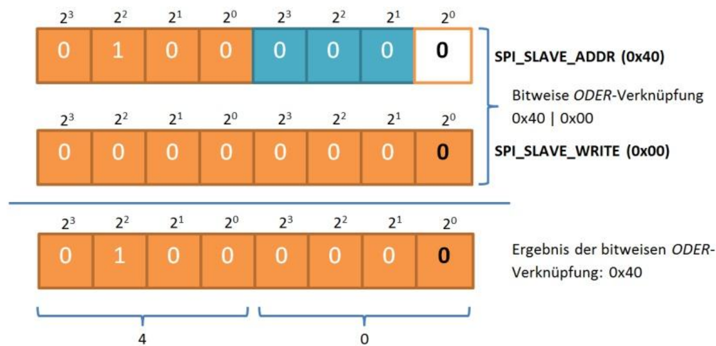
```
17  # MCP23S17 Werte
18  SPI_SLAVE_ADDR  = 0x40
19  SPI_IOCTRL      = 0x0A
20  SPI_IODIRA      = 0x00
21  SPI_IODIRB      = 0x01
22  SPI_GPIOA       = 0x12
23  SPI_GPIOB       = 0x13
24
25  SPI_SLAVE_WRITE = 0x00
26  SPI_SLAVE_READ  = 0x01
```

Die Variablen *SPI_SLAVE_WRITE* und *SPI_SLAVE_READ* werden dazu genutzt, den Datenfluss entsprechend zu steuern. Wir müssen also irgendwie den Wert der *SPI_SLAVE_ADDR* anpassen, dass es nach unseren Wünschen funktioniert. Das geht über eine bitweise *ODER*-Verknüpfung. Die Logiktabelle schaut wie folgt aus:

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	1

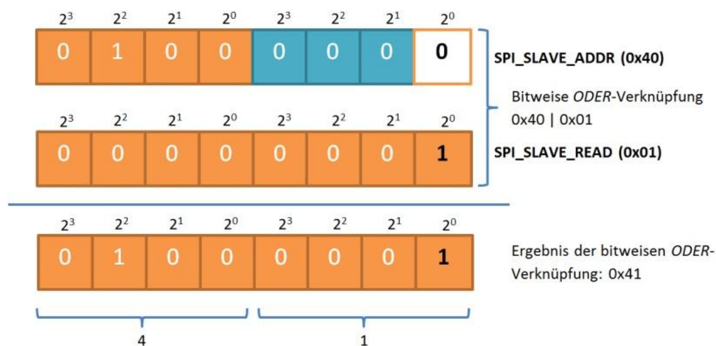
Tabelle 1 Bitweise ODER-Verknüpfung

Sehen wir uns dazu die Verknüpfung der entscheidenden Werte an. Beginnen wir mit der Funktion des *Schreibens*. Das *R/W-Bit* muss den Wert *0* bekommen.



Bei einer *ODER*-Verknüpfung mit dem Wert *0x00* ändert sich nichts am ursprünglichen Wert, was ja auch in unserem Sinne ist.

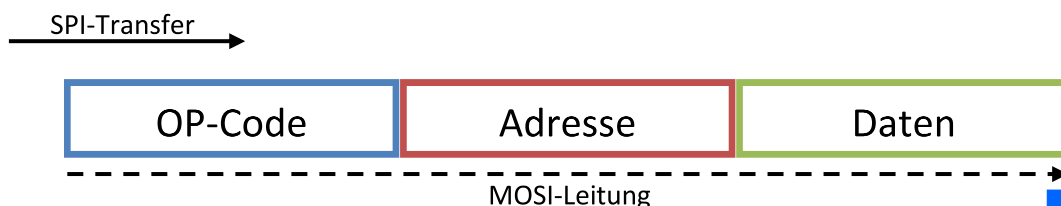
Möchten wir jetzt Werte vom *Port-Expander* lesen, muss das *R/W-Bit* angepasst werden und den Wert *1* erhalten.



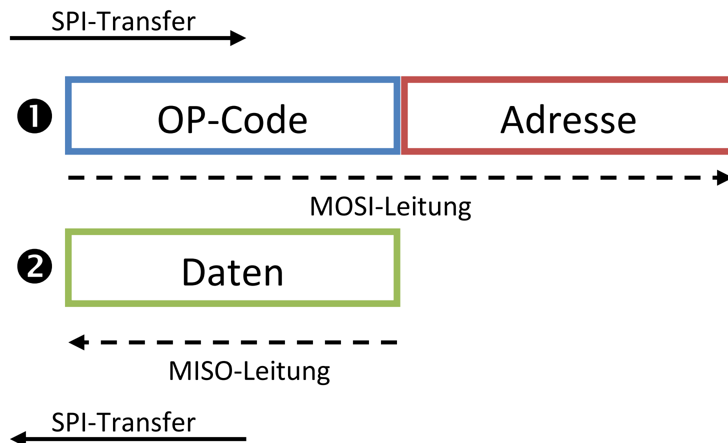
Durch das niederwertigste Bit auf der rechten Seite, was auch *LSB* (*Least-Significant-Bit*) genannt wird, das jetzt auf *1* erhält, erreichen wir, dass die Funktion des *Lesens* ermöglicht wird.

Der Datenempfang

Soweit so gut! Kommen wir zum eigentlichen *Datenempfang*. Doch zuerst möchte ich dir noch einmal das *Versenden* mit den einzelnen Schritten zeigen, was ich in *Teil 1* ausführlich beschrieben habe. Die Übertragung erfolgte in *3* Blöcken mit den Elementen *OP-Code*, *Adresse* und *Daten*.



Da wir nun aber auch Daten empfangen möchten, muss ein Rücktransport erfolgen. Schau her:



❶: *Versenden* der Anforderung zum späteren Empfang der Daten
❷: *Empfangen* der Daten

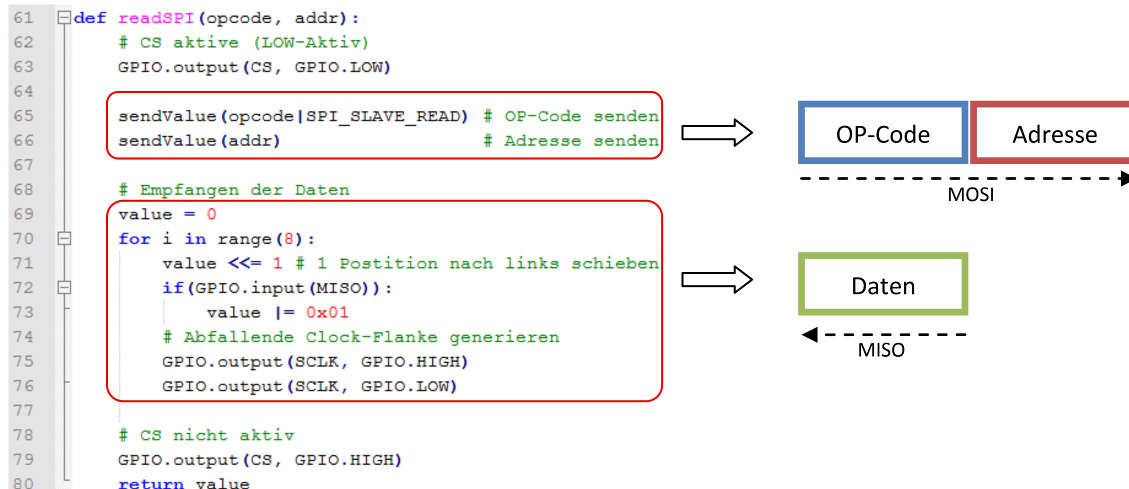
Port-B wird weiterhin als *Ausgang* genutzt. Abweichend davon wird der komplette *Port-A* nun als *Eingang* programmiert,

```
sendSPI(SPI_SLAVE_ADDR, SPI_IODIRB, 0x00) # GPPIOB als Ausgaenge programmieren
sendSPI(SPI_SLAVE_ADDR, SPI_IODIRA, 0xFF) # GPPIOA als Eingaenge programmieren
sendSPI(SPI_SLAVE_ADDR, SPI_GPIOB, 0x00)  # Reset des GPIOB
```

was über die Zuweisung des Wertes *0xFF* erfolgt. Erinnere dich an die Tatsache, dass je Pin folgende Festlegung gilt:

- *0*: Ausgang
- *1*: Eingang

Kommen wir zur Python-Funktion, die das *Empfangen* der Daten erledigt. Ich habe sie entsprechend *readSPI* genannt.



```
61  def readSPI(opcode, addr):
62      # CS aktive (LOW-Aktiv)
63      GPIO.output(CS, GPIO.LOW)
64
65      sendValue(opcode|SPI_SLAVE_READ) # OP-Code senden
66      sendValue(addr)                  # Adresse senden
67
68      # Empfangen der Daten
69      value = 0
70      for i in range(8):
71          value <<= 1 # 1 Postition nach links schieben
72          if(GPIO.input(MISO)):
73              value |= 0x01
74          # Abfallende Clock-Flanke generieren
75          GPIO.output(SCLK, GPIO.HIGH)
76          GPIO.output(SCLK, GPIO.LOW)
77
78      # CS nicht aktiv
79      GPIO.output(CS, GPIO.HIGH)
80      return value
```


Du erkennst im oberen Bereich das *Versenden* von *OP-Code* und *Adresse*, im unteren das *Empfangen* der *Daten*. Die Funktion liefert über eine *Return*-Anweisung den empfangenen Wert an den Aufrufer zurück. Jetzt bist du wo weit, dass ich dich mit der eigentlichen *main*-Funktion konfrontieren kann. Natürlich findest du den kompletten Quellcode wieder auf einer Internetseite im Downloadbereich.

```
81  def main():
82      # Pin-Programmierung
83      GPIO.setup(SCLK, GPIO.OUT)
84      GPIO.setup(MOSI, GPIO.OUT)
85      GPIO.setup(MISO, GPIO.IN)
86      GPIO.setup(CS,   GPIO.OUT)
87
88      # Pegel vorbereiten
89      GPIO.output(CS,   GPIO.HIGH)
90      GPIO.output(SCLK, GPIO.LOW)
91
92      # Initialisierung de MCP23S17
93      sendSPI(SPI_SLAVE_ADDR, SPI_IODIRB, 0x00) # GPPIOB als Ausgaenge programmieren
94      sendSPI(SPI_SLAVE_ADDR, SPI_IODIRA, 0xFF) # GPPIOA als Eingaenge programmieren
95      sendSPI(SPI_SLAVE_ADDR, SPI_GPIOB, 0x00)  # Reset des GPIOB
96
97      while True:
98          for i in range(len(ledPattern)):
99              sendSPI(SPI_SLAVE_ADDR, SPI_GPIOB, ledPattern[i]) # PORT-B ansteuern
100             print bin(readSPI(SPI_SLAVE_ADDR, SPI_GPIOA))      # PORT-A abfragen
101             time.sleep(0.5)
```

Innerhalb der *while*-Schleife, die als eine Endlosschleife arbeitet, befinden sich die beiden Aufrufe zum Ansteuern der LEDs über *PORT-B* (*sendSPI*) und das abfragen der Taster über *PORT-A* (*readSPI*). Am Ende wird bei jedem Schleifendurchlauf über den Aufruf der *sleep*-Funktion ½-Sekunde Pause eingelegt.

Etwas hast du aber noch vergessen! Wie sieht denn die Ausgabe der Statuswerte auf der Konsole aus? Kannst du mir das bitte einmal zeigen.

Du bist ja wieder einmal schnell! Das wollte ich doch gerade machen. Du musst mir schon ein wenig Zeit geben, die Sache so vorzubereiten, dass du es auch sicherlich verstehst.

```
pi@raspberrypi ~ $ sudo python control_mcp23s17_read_write001.py
0b11111111
0b11111110
0b11111101
0b11111100
0b11110100
0b10110100
```

Beachte, dass wir es hier mit einer umgekehrten Logik zu tun haben!

Das kommt dadurch, dass wir mit *Pullup*-Widerständen arbeiten und bei einem Tastendruck der Pegel nach Masse gezogen wird.

Hey cool, du hast es voll im Griff! Eigentlich brauche ich das hier ja nicht mehr zu erklären, doch wir müssen auch an die anderen denken, die vielleicht nicht so schnell wie du sind.

Schauen wir uns die Ausgabe im übertragenen Sinne in einer Wertetabelle an, die dann wie folgt aussieht:

S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_0	Was würde gedrückt?
1	1	1	1	1	1	1	1	Kein Taster
1	1	1	1	1	1	1	0	Taster S_0
1	1	1	1	1	1	0	1	Taster S_1
1	1	1	1	1	1	0	0	Taster $S_0 + S_1$
1	1	1	1	0	1	0	0	Taster $S_0 + S_1 + S_3$
1	0	1	1	0	1	0	0	Taster $S_0 + S_1 + S_3 + S_6$

Tabelle 2 Wertetabelle des Tasterstatus

Die Verkabelung auf meinem *Prototyping-Board* schaut wie folgt aus:

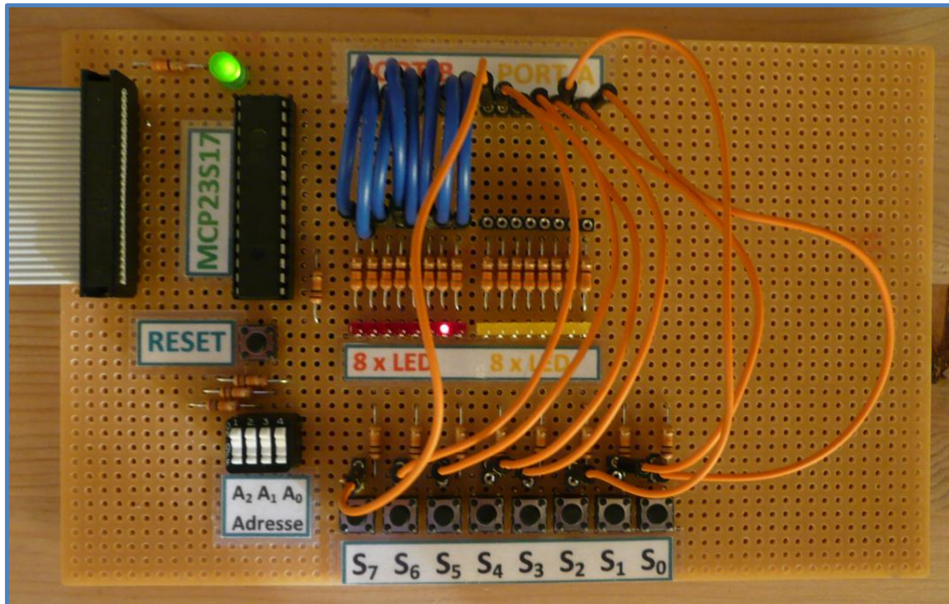


Tabelle 3 Verkabelung des Prototyping-Boards

Was hast du also jetzt gelernt? Wie du mit ganz einfachen Mitteln den *Port-Expander* derart ansteuern kannst, dass du z.B. angeschlossene LEDs blinken lässt oder Taster abfragst. Die Möglichkeiten, die sich daraus ergeben, sind nahezu unbegrenzt.

Auf *YouTube* kannst du dir ein kleines Video anschauen, das dir die Funktionsweise zeigt:

http://www.youtube.com/watch?v=05FvxaswZMc

Bezugsquellen

Den *MCP23S17* kannst du z.B. unter der folgenden Adresse beziehen:
http://www.reichelt.de/ICs-M-MN-/MCP-23S17-E-SP/3//index.html?ACTION=3&GROUPID=2914&ARTICLE=90047&SHOW=1&START=0&OFFSET=16&

Hier noch weitere Komponenten, die dich sicherlich interessieren werden:

Lochraster/Punktraster-Platine:
http://www.pollin.de/shop/dt/OTM3OTU1OTk-/Bauelemente_Bauteile/Mechanische_Bauelemente/Montagematerial/Punktrasterplatine.html

Buchsenleisten:
http://www.pollin.de/shop/dt/NDE5OTQ1OTk-/Bauelemente_Bauteile/Mechanische_Bauelemente/Steckverbinder_Klemmen/Buchsenleiste.html

Vorschau

Ich plane weitere *AddOns* zur Ansteuerung des *Port-Expanders*. Ich hatte schon im ersten Teil kurz erwähnt, dass es eine interessante Möglichkeit gibt, über ein grafisches Frontend den Port-Expander zu steuern.

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/