Python GUI mit TKinter

Kurze Einführung

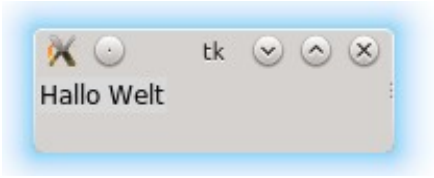
Fast alle moderne Programme besitzen eine graphische Benutzeroberfläche (***Graphical User Interface (GUI)***). Die Bedienung des Programms erfolgt also nicht mehr über Texteingaben im Terminal-Emulator (Konsole) sondern über interaktive Elemente wie Knöpfe (*buttons*) oder Kontrollkästchen (*checkbox*) mit der Maus.

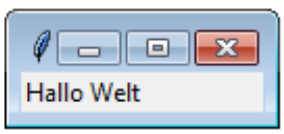
Das Python-Modul **Tkinter** dient der Gestaltung einer solchen graphischen Benutzeroberfläche. Es existieren noch andere solcher GUIs für Python (GTK, Qt), allerdings ist Tkinter schon in Python enthalten und soll hier genutzt werden. Um die modernere Version von TKinter (mit thematischen Ttk-Widgets) nutzen zu können, wird eine neuere Python Version >= 3.1 (>= 2.7) benötigt. Wir verwenden hier eine dreier Version.

Hallo Welt

```
#!/usr/bin/env python3
# tk_hallo.py

from tkinter import *    # Python 2.7 "from Tkinter import *"
from tkinter import ttk  # Python 2.7 "import ttk"

mainWin = Tk()

label_1 = ttk.Label(mainWin, text='Hallo Welt')
label_1.grid()

mainWin.mainloop()
```

Nachdem das erste Programm als **`tk_hallo.py`** (Windows `tk_hallo.pyw` damit die Konsole nicht aufgerufen wird) abgespeichert wurde und mit **`python3 tk_hallo.py`** (bzw.: `C:\Python34\pythonw tk_hallo.pyw`) aufgerufen wurde erscheint ein Fenster mit dem entsprechenden beschrifteten Button. (Linux bzw. Windows).






```
#!/usr/bin/env python3
```

```
# tk_hallo.py
```

Die erste Zeile ist an sich ein Kommentar, der aber unter Linux aufzeigt mit welchem Programm das Python-Script ausgeführt werden soll. Dazu muss die Datei allerdings die richtigen Rechte besitzen, d.h. sie für den Anwender ausführbar sein (Befehl: **chmod**). Die zweite Zeile enthält den Dateinamen als Kommentar.

```
from tkinter import *    # Python 2.7 "from Tkinter import *"
from tkinter import ttk  # Python 2.7 "import ttk"
```

In den nächsten beiden Zeilen werden zwei Module (Klassenbibliotheken) geladen. Beim ersten Modul (Standard Tkinter Klassen) wird alles importiert, so dass auf diese Klassen ohne den Vorsatz "`tkinter.`" zugegriffen werden kann. Beim zweiten Import wird nur Ttk geladen, um die neueren Ttk-Widgets nutzen zu können. Um sie von den Standard-Widgets zu unterscheiden muss hier dann der Vorsatz "`ttk.`" verwendet werden.

```
mainWin = Tk()
...
mainWin.mainloop()
```

Grafische Elemente (Hauptfenster, Knöpfe, Text, Eingabefelder ...) einer GUI werden **Widgets** genannt (*window gadgets*). Ein Widget kann aus mehreren Widgets zusammengesetzt sein.

Jedes Widget entspricht einer Klasse (Bauplan für Objekte), aus der dann die Objekte gebildet werden. Jedes Objekt besitzt Attribute (Daten des Objekts) und Methoden (Funktionen) des Objekts. Eine GUI kann man sich als Baumstruktur vorstellen. Die Wurzel ist das Hauptfenster, ein Objekt der Klasse Tk. In der dritten Zeile wird diese Objekt mit dem (beliebigen) Namen "`mainWin`" erstellt. In der letzten Zeile wird dann die Methode (Funktion) `mainloop()` des Objekts `mainWin` aufgerufen. Dabei werden Objekt und Methode mit einem Punkt verbunden.

```
label_1 = ttk.Label(mainWin, text='Hallo Welt')
label_1.grid()
```

In der vierten Zeile wird ein Ttk-Objekt mit dem Namen `label_1` erzeugt. Dieses Objekt ist ein "Kind" des Hauptfensters. Dies wird durch den ersten Eintrag (Parameter) in der Klammer festgelegt. Die Beschriftung des Labels wird beim zweiten Parameter in der Variablen "`text`" hinterlegt. Dem Widget **`ttk.Label`** können noch andere Parameter übergeben werden (siehe später). Das Objekt wurde zwar erzeugt, ist aber noch nicht sichtbar, da Tk nicht klar ist wie das Widget in Bezug zum Hauptfenster platziert werden soll. Dies übernimmt die Methode `grid()` in der fünften Zeile.

Aufgabe Tk1:

Teste das "Hallo Welt" Programm auf dem Raspberry Pi.

Einige Basis Widgets

Neben dem Hauptfenster und dem Label existieren noch viele andere Widgets wie zum Beispiel Knöpfe (*button, checkbutton, radiobutton*), Auswahlfelder (*combobox, listbox, spinbox*), Eingabefelder (*Entry, Text*) usw.

Um ein erstes sinnvolles Programm zu schreiben wollen wir uns ***Label***, ***Entry*** und ***Button*** etwas näher ansehen:

Label

Labels sollen dem Betrachter Informationen oder Resultate liefern. Hierfür wird das Widget **`ttk.Label`** genutzt. Der darzustellende Text wird meist mit dem "`text=`" Parameter übergeben und ist konstant (siehe Kapitel "Hallo Welt"). Man kann aber stattdessen auch den Parameter "`textvariable=`" benutzen um veränderbaren Text, also eine Variable, zu übergeben:

```
result = StringVar()
result.set('neuer_Wert')
label_1 = ttk.Label(mainWin, textvariable=result)
label_1.grid()
```

Tkinter ermöglicht uns die Klasse `StringVar()` für die Textvariable zu nutzen. Diese kümmert sich um die ganze Logistik zur Überwachung von Änderungen der Variablen und zur Kommunikation zwischen der Variable und dem Label.

Auch ist es möglich Bilder im Label anzuzeigen (Parameter: "`image=`"). Mit dem Parameter "`compound=`" gibt man an, ob nur der Text (`text`), nur das Bild (`image`), Text im Bild (`center`) oder das Bild über, unter, rechts oder links vom Text (`top`,`bottom`,`right`,`left`) angezeigt wird. Hier ein kleines Beispiel:

```
#!/usr/bin/env python3
# tk_label.py

from tkinter import *    # Python 2.7 "from Tkinter import *"
from tkinter import ttk  # Python 2.7 "import ttk"

mainWin = Tk()

result = StringVar()
result.set('weigu.lu')
imageLabel_1 = PhotoImage(file='myimage.png')

label_1 = ttk.Label(mainWin, textvariable=result, image=imageLabel_1, compound='top')
label_1.grid()

mainWin.mainloop()
```

Referenz zum `ttk.Label` Widget: https://www.tcl.tk/man/tcl/TkCmd/ttk_label.htm

Aufgabe Tk2:

Teste das Programm mit eigenem Text und Bild.

Entry

Zur Eingabe von Daten kann man das Widget **`ttk.Entry`** nutzen. Es handelt sich hierbei um eine Textzeile mit der ein String übergeben wird. Das Widget erhält also einen String sobald Tasten im Eingabefeld betätigt werden. Der String wird während des Tippens dauernd aktualisiert. Mit dem Parameter "`textvariable=`" gibt man ab welchen Variablen (von der Klasse `StrinVar()`) die Information weitergereicht werden soll. Sobald Änderungen im `Entry` Textfeld auftreten werden diese an die Variable weitergereicht.

```
password = StringVar()
entry_1 = ttk.Entry(mainWin, textvariable=password, width=6, show='*')
entry_1.grid()
entry_1.focus()
```

Mit dem Parameter "`width=`" kann man die Standardlänge des Textfeldes verändern. Mit dem Parameter "`show=`" kann man verhindern, dass der momentane Text angezeigt wird, zum Beispiel bei einer Passwortabfrage. Mit der Methode "`focus()`" wird der Cursor gleich nach dem Starten des Programms ins Eingabefeld gesetzt, ohne dass dieses zuerst angeklickt werden muss.

Um Text oder Bilder neben dem Textfeld zu nutzen muss ein zusätzliches Label Widget genutzt werden.

Referenz zum `ttk.Entry` Widget: https://www.tcl.tk/man/tcl/TkCmd/ttk_entry.htm

Aufgabe Tk3:

Schreibe ein kleines Programm, das mit Hilfe eines Textfensters und dreier Labels das folgende Fenster erzeugt. Während des Tippens soll die letzte Zeile den Inhalt des Textfensters anzeigen.

Button

Im obigen Programm wäre es eventuell wünschenswert das Passwort erst nach dem Tippen anzuzeigen. Eine Methode um dies zu erreichen ist das Betätigen eines Button. Das Widget "`ttk.Button`" dient mehr als die beiden oberen Widgets der Interaktion mit dem Benutzer. Hauptsächlich soll er eine Aktion auslösen d.h. ein Kommando ausführen.

```
button_1 = ttk.Button(mainWin, text='show', command=showPass, width=6)
button_1.grid()
```

Neben vielen neuen Eigenschaften kann der Button aber auch Text und Bilder enthalten. Dieselben Parameter wie beim Label können genutzt werden ("`text=`", "`textvariable=`", "`compound=`", "`image=`", "`width=`").

Im folgenden wird unsere Aufgabe Tk3 um einen Button erweitert, der das Kommando "`showPass`" ausführt. Die Funktion `showPass()` muss sich vor dem Aufruf des Kommando befinden, damit das Programm das Kommando kennt.

In der Funktion wird das Passwort der Variablen des dritten Labels übergeben. Wir benötigen also jetzt zwei StringVar()-Objekte, eine für das Entry-Widget und eine für das Label-Widget. Das es sich bei einem StringVar()-Objekt nicht um eine normale Variable handelt kann für die Zuweisung nicht einfach das Gleichheits-Zeichen benutzt werden, sondern es müssen die Methoden `.get()` und `.set()` verwendet werden, um die Daten der `StringVar()`-Objekte auszutauschen.

```
def showPass():
    passwordClear.set(password.get())
```

Wem diese Zeile nicht geheuer ist, kann natürlich die Zuweisung mit einer lokalen StringVar()-Variablen auch schrittweise durchführen:

```
def showPass():
    temp = StringVar()
    temp = password.get()
    passwordClear.set(temp)
```

Hier das vollständige Programm:

```
#!/usr/bin/env python3
# tk_button.py

from tkinter import *    # Python 2.7 "from Tkinter import *"
from tkinter import ttk  # Python 2.7 "import ttk"

def showPass():
    passwordClear.set(password.get())

mainWin = Tk()

password = StringVar()
passwordClear = StringVar()

label_1 = ttk.Label(mainWin, text='Type your password: ')
label_1.grid()
entry_1 = ttk.Entry(mainWin, textvariable=password, width=6, show='*')
entry_1.grid()
entry_1.focus()
label_2 = ttk.Label(mainWin, text='Your password is: ')
label_2.grid()
label_3 = ttk.Label(mainWin, textvariable=passwordClear)
label_3.grid()
button_1 = ttk.Button(mainWin, text='show', command=showPass, width=6)
button_1.grid()

mainWin.mainloop()
```

Referenz zum `ttk.Button` Widget: https://www.tcl.tk/man/tcl/TkCmd/ttk_button.htm

Aufgabe Tk4:

a) Erweitere das obige Programm um die folgende Zeile:

```
mainWin.bind('<Return>',showPass)
```

Mit dieser Zeile wird die Return- bzw. Enter-Taste mit unserer Funktion `showPass` verbunden, so dass das Passwort auch beim Drücken der Enter-Taste erscheint. Allerdings erhalten wir nun die Fehlermeldung:

```
TypeError: showPass() takes 0 positional arguments but 1 was given
```

Die Zeile der Funktionsdeklaration muss um "`*args`" erweitert werden, damit das Argument angenommen werden kann:

```
def showPass(*args):
```

b) Teste das vollständige Programm!

Frame

Da das Hauptfenster leider nicht zum Ttk-Widget-Set gehört werden wir einen Ttk- Rahmen (`ttk.Frame`) verwenden, um in diesem die Widgets anzuordnen (Dies ist auch vorteilhaft wenn man den ganzen Bildschirm in eine eigene Klasse einpacken möchte).

Das Widget **`ttk.Frame`** wird meist als Container für andere Widgets genutzt um Ordnung im Layout zu schaffen. Normalerweise erhält der Rahmen automatisch seine Größe, durch die Größe der Widgets die er umfasst. Mit "`width=`" und "`height=`" kann die Größe aber auch statisch festgelegt werden. Wird eine Zahl eingegeben, so handelt es sich um Bildschirmpixel. Mit einem angehängten 'c' (15c) kann Höhe bzw. Breite aber auch in Zentimeter (i für inch, p für printer's point) festgelegt werden. Damit die Höhe und die Breite berücksichtigt werden muss aber mit der Methode `grid_propagate(0)` die automatische Anpassung abgeschaltet werden.

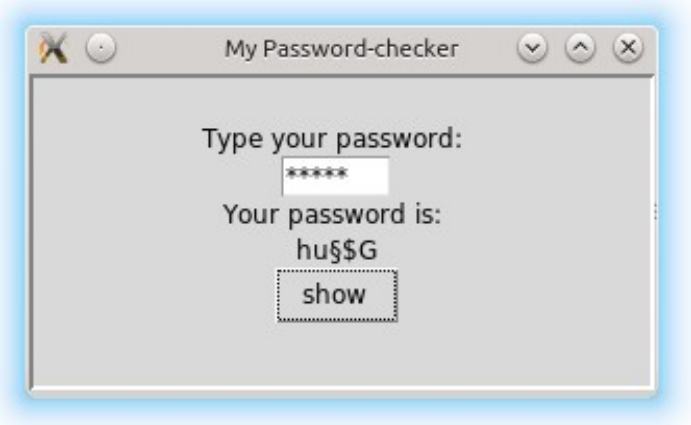
Mit dem Parameter "`padding=`" kann ein Abstand der inneren Widgets zum Rahmen vereinbart werden. Wird eine Zahl eingegeben, so ist der Abstand überall gleich. Mit zwei Zahlen wird der vertikale und der horizontale Abstand definiert. Mit vier Zahlen der Abstand in vier Richtungen (left, top, right, bottom im Uhrzeigersinn).

Natürlich kann der Rahmen auch einen sichtbaren Rand haben. Dies geschieht mit dem Parameter "`borderwidth=`" (default = 0). Mit dem Parameter "`relief=`" kann die Optik des sichtbaren Randes verändert werden. Optionen sind: '`flat`' (default), '`raised`', '`sunken`', '`solid`', '`ridge`' und '`groove`'.

In unser Programm mit Rahmen müssen natürlich die Widgets jetzt "Kinder" des Rahmens sein und nicht mehr des Hauptfensters. Das Programm (Hauptfenster) erhält zusätzlich einen Titel mit der Methode `title()`. Zur Demonstration, und damit der Titel lesbar ist wurde hier mit fester Rahmengröße gearbeitet. Im Normalfall ist es besser die Widgets passen sich der Fenstergröße an, wie wir im nächsten Kapitel sehen werden.

```
#!/usr/bin/env python3
# tk_frame.py

from tkinter import *    # Python 2.7 "from Tkinter import *"
from tkinter import ttk  # Python 2.7 "import ttk"

def showPass(*args):
    passwordClear.set(password.get())

mainWin = Tk()
mainWin.title('My Password-checker')

password = StringVar()
passwordClear = StringVar()

mainFrame = ttk.Frame(mainWin, borderwidth=2, width=300, height=150, relief='groove', padding='80 20
80 20')
mainFrame.grid_propagate(0)
mainFrame.grid()

label_1 = ttk.Label(mainFrame, text='Type your password: ')
label_1.grid()
entry_1 = ttk.Entry(mainFrame, textvariable=password, width=6, show='*')
entry_1.grid()
entry_1.focus()
label_2 = ttk.Label(mainFrame, text='Your password is: ')
label_2.grid()
label_3 = ttk.Label(mainFrame, textvariable=passwordClear)
label_3.grid()
button_1 = ttk.Button(mainFrame, text='show', command=showPass, width=6)
button_1.grid()
```

```
mainWin.bind('<Return>',showPass)

mainWin.mainloop()
```



Referenz zum `ttk.Frame` Widget: https://www.tcl.tk/man/tcl/TkCmd/ttk_frame.htm

Aufgabe Tk5:

a) Entferne die feste Rahmengröße wieder aus dem Programm. Teste das Programm mit unterschiedlichen Reliefs, Randbreiten und Paddings. Wähle eine Kombination die dir gefällt.

b) Ändere dein Programm so um, dass ein Passwort-checker entsteht. Statt der Funktion `showPass()` soll eine neue Funktion `checkPass()` überprüfen wie viele Zeichen das Passwort enthält. Unter 6 Zeichen meldet das Programm dann 'weak', zwischen 6 und 8 Zeichen 'OK' und bei mehr als 8 Zeichen 'strong'. Dazu wird das `StringVar()`-Objekt in einen String geladen (`check = password.get()`), der dann mit der Funktion `len()` überprüft wird. Passe die Namen der Variablen, der Funktion und den Text des Button an.

Der Grid-Layout-Manager

Die Aufgaben des Layout-Manager (Geometrie-Manager) sind recht komplex, da Widgets unterschiedliche Größen haben und Fenster skaliert werden können. In Tk gibt es unterschiedliche Layout Manager. Am intuitivsten einsetzbar ist der Grid-Layout Manager. Mit ihm kann ein Tabellen-Raster (Zeilen und Spalten) erzeugt werden an dem die Widgets ausgerichtet werden. Zeilen und Spalten sind von oben nach unten und rechts nach links durchnummeriert beginnend bei Null.

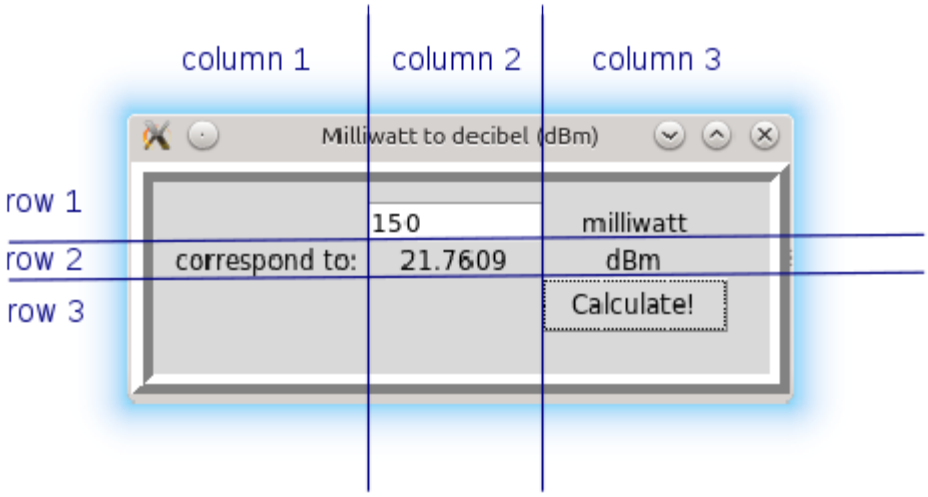
Reihen und Kolonnen

Im folgenden wollen wir ein neues Programm entwickeln. Bevor wir das tun ist es sinnvoll sich zu überlegen, wie das Programm aussehen soll, und die einzelnen Komponenten in

einer Tabellenzelle zuzuordnen. Hier ein Beispiel für unser Programm, das eine Leistung in Milliwatt in Dezibel (bezogen auf ein Milliwatt) umrechnen soll.

$$L_p = 10 \cdot \lg \frac{P}{1\,mW} \rightarrow L_p \cdot \text{in} \cdot \text{dBm}$$

In der ersten Zeile befindet sich ein Entry-Widget und ein Label-Widget. In der zweiten Zeile drei Label-Widgets und in der vierten Zeile das Button-Widget.



Mit dem Erlernten aus dem vorigen Kapitel ist es nicht schwierig das Programm zu erstellen. Neu ist, dass wir der `grid()`-Methode die Parameter "`column=`" und "`row=`" übergeben, und damit die Position der Widgets im Raster festlegen.

Zur Berechnung wird der Inhalt des `StringVar()`-Objekts mit der Methode `get()` abgeholt. Der Inhalt entspricht einer Zeichenkette (string) und muss mit der Funktion `int()` nach Integer konvertiert werden. Wenn Kommastellen erwünscht sind ist die Funktion `float()` zu verwenden. (Das Komma entspricht dabei einem Punkt!) Damit die Logarithmus-Funktion nutzbar ist muss sie aus dem math-Modul importiert werden. Der zweite Parameter ist die Basis des Logarithmus (hier 10). Mit der Funktion round() wird die Stellenzahl hinter dem Komma auf 4 begrenzt. Mit der Funktion `str()` wird das Integer-Resultat schlussendlich wieder in einen String überführt und mit `.set()` dem zweiten `StringVar()`-Objekt übergeben.

Falsche Eingaben werden mit der "`try...except`"-Anweisung abgefangen. Der Programmteil, wo eine Ausnahme zu erwarten ist befindet sich hinter "`try:`". Hinter "`except:`" befindet sich dann der Code der ausgeführt werden soll, falls eine solche Ausnahme auftritt. Da wir in unserem Fall wissen, dass es sich um einen Eingabefehler handeln muss, können wir das mit der "`ValueError`"-Ausnahme (exception) und einer eindeutigen Fehleraussage auch sauber dokumentieren.

Hier das vollständige Programm:

```
#!/usr/bin/env python3
# tk_grid1.py

from tkinter import *    # Python 2.7 "from Tkinter import *"
from tkinter import ttk  # Python 2.7 "import ttk"
from math import log

def calculate(*args):
    try:
        power=int(power_mW.get())
        dBm=round(10*log(power,10),4)
        result_dBm.set(str(dBm))
    except ValueError:
        result_dBm.set('error: entry not valid!')

mainWin = Tk()
mainWin.title('Milliwatt to decibel (dBm)')

power_mW = StringVar()
result_dBm = StringVar()

mainFrame = ttk.Frame(mainWin, borderwidth=10, relief='ridge', padding="20")
mainFrame.grid()

entry_1 = ttk.Entry(mainFrame, textvariable=power_mW, width=10)
entry_1.grid(column=2, row=1)
entry_1.focus()
label_1 = ttk.Label(mainFrame, text='milliwatt')
label_1.grid(column=3, row=1)
label_2 = ttk.Label(mainFrame, text='correspond to: ')
label_2.grid(column=1, row=2)
label_3 = ttk.Label(mainFrame, textvariable=result_dBm)
label_3.grid(column=2, row=2)
label_4 = ttk.Label(mainFrame, text='dBm')
label_4.grid(column=3, row=2)
butt_1 = ttk.Button(mainFrame, text='Calculate!', command=calculate, width=10)
butt_1.grid(column=3, row=3)

mainWin.bind('<Return>',calculate)

mainWin.mainloop()
```

Aufgabe Tk6:

Teste das Programm auf dem Raspberry Pi. Teste auch was bei falscher Eingabe passiert!

Skalierbare Fenster

Die Widgets wurden zwar schön angeordnet, allerdings passen sie sich nicht der Fenstergröße an, falls diese verändert wird. Dazu werden die beiden Methoden `columnconfigure=()` und `rowconfigure=()` benötigt. Sie müssen auf jede Kolonne und jede Reihe des Rahmens angewendet werden. Zusätzlich müssen sie auf das Hauptfenster angewendet werden, da dieses ja verändert wird und der Rahmen sich in diesem befindet.

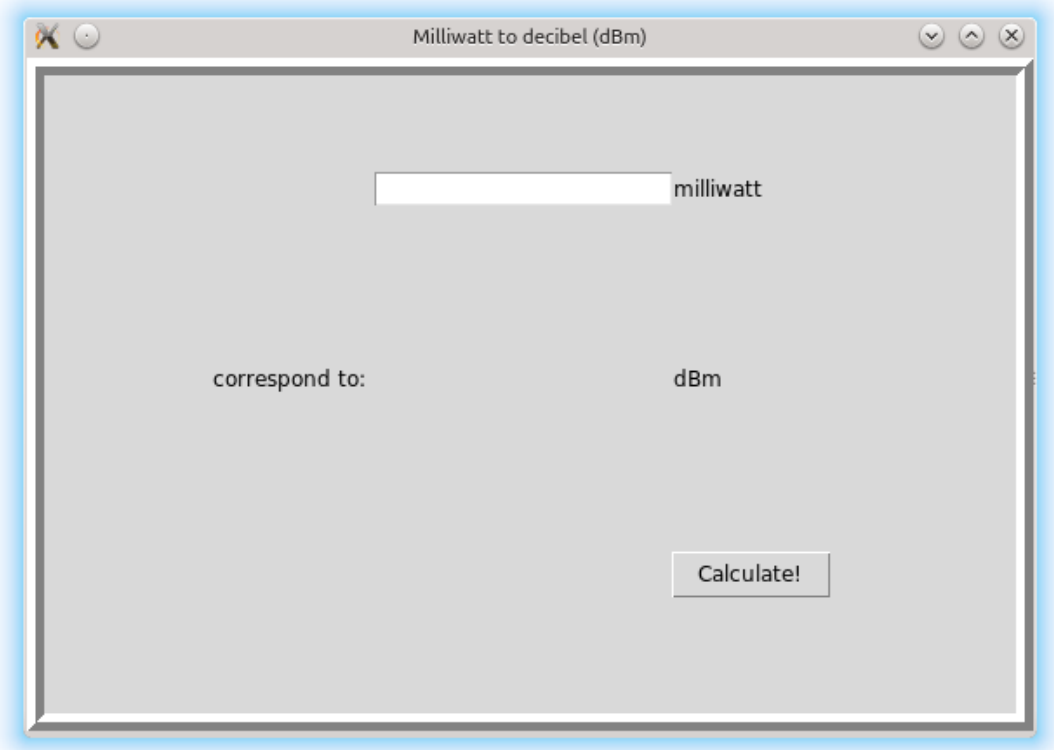
```
mainWin.columnconfigure(0, weight=1)          # numbering beginns with 0 for old Tk widgets
mainWin.rowconfigure(0, weight=1)
...
mainFrame.columnconfigure(1, weight=5)        # numbering beginns with 1 for newer Ttk widgets
mainFrame.columnconfigure(2, weight=5)
mainFrame.columnconfigure(3, weight=5)
mainFrame.rowconfigure(1, weight=1)
mainFrame.rowconfigure(2, weight=1)
mainFrame.rowconfigure(3, weight=1)
```

Der erste Parameter bezeichnet die Reihe bzw. die Kolonne, der zweite Parameter "`weight=`" gibt Gewichtung, wie schnell sich die Widgets bei der Veränderung des Fensters bewegen (Mit `weight=0` bewegen sie sich nicht). Bei den alten Tk-Widgets begann die Nummerierung der Reihen und Kolonnen mit 0 statt 1. Dies gilt hier für unser Hauptfenster!

Zusätzlich muss jedem Widget mitgeteilt werden, zu welchen der vier Kanten es sich hin bewegen soll. Dies wird mit dem Parameter "`sticky=`" der `grid()`-Methode erreicht. Die vier Himmelsrichtungen dienen hier der Ausrichtung (N: Norden, oben, S: Süden, unten, W: Westen, links, E: Osten, rechts). Wird mehr als eine Richtung angegeben müssen Klammern gesetzt werden.

Das veränderte Programm sieht jetzt folgendermaßen aus:

```
#!/usr/bin/env python3
#tk_grid2.py

from tkinter import *    # Python 2.7 "from Tkinter import *"
from tkinter import ttk  # Python 2.7 "import ttk"
from math import log

def calculate(*args):
    try:
        power=int(power_mW.get())
        dBm=round(10*log(power,10),4)
        result_dBm.set(str(dBm))
    except ValueError:
        result_dBm.set('error: entry not valid!')

mainWin = Tk()
mainWin.title('Milliwatt to decibel (dBm)')
mainWin.columnconfigure(0, weight=1)         # numbering beginns with 0 for old Tk widgets
mainWin.rowconfigure(0, weight=1)

power_mW = StringVar()
result_dBm = StringVar()

mainFrame = ttk.Frame(mainWin, borderwidth=10, relief='ridge', padding="20")
mainFrame.grid(column=0, row=0, sticky=(W, N, E, S))
mainFrame.columnconfigure(1, weight=5)       # numbering beginns with 1 for newer Ttk widgets
mainFrame.columnconfigure(2, weight=5)
mainFrame.columnconfigure(3, weight=5)
mainFrame.rowconfigure(1, weight=1)
mainFrame.rowconfigure(2, weight=1)
mainFrame.rowconfigure(3, weight=1)

entry_1 = ttk.Entry(mainFrame, textvariable=power_mW, width=10)
entry_1.grid(column=2, row=1, sticky=(W, E))
entry_1.focus()
label_1 = ttk.Label(mainFrame, text='milliwatt')
label_1.grid(column=3, row=1, sticky=W)
label_2 = ttk.Label(mainFrame, text='correspond to: ')
label_2.grid(column=1, row=2, sticky=E)
label_3 = ttk.Label(mainFrame, textvariable=result_dBm)
label_3.grid(column=2, row=2, sticky=(W, N, E, S))
label_4 = ttk.Label(mainFrame, text='dBm')
label_4.grid(column=3, row=2, sticky=W)
butt_1 = ttk.Button(mainFrame, text='Calculate!', command=calculate, width=10)
butt_1.grid(column=3, row=3, sticky=W)

mainWin.bind('<Return>',calculate)

mainWin.mainloop()
```



Padding und Columnspan

Der Abstand zwischen dem Eingabefeld und der Einheit ist zu klein. Mit Hilfe der Parameter "padx=" und "pady=" kann bei der grid()-Methode ein innerer Abstand zur Tabellenzelle eingehalten werden. Zum Beispiel für den oben erwähnten Abstand:

```
label_1.grid(column=3, row=1, sticky=W, padx=30)
```

In unserem Fall wäre ein Abstand zwischen allen Feldern wünschenswert. Dies lässt sich um Tipparbeit zu sparen auch leicht in einer Schleife mit der grid_configure() Methode erreichen:

```
# Padding
for child in mainFrame.winfo_children():
    child.grid_configure(padx=10, pady=10)
```

Manchmal ist es auch nötig ein Widget über mehrere Felder auszudehnen.Dies kann mit den Parametern "columnspan=" und "rowspan=" der grid()-Methode erfolgen. Als Beispiel soll der Button sich über drei Felder erstrecken (width=10 wurde gelöscht):

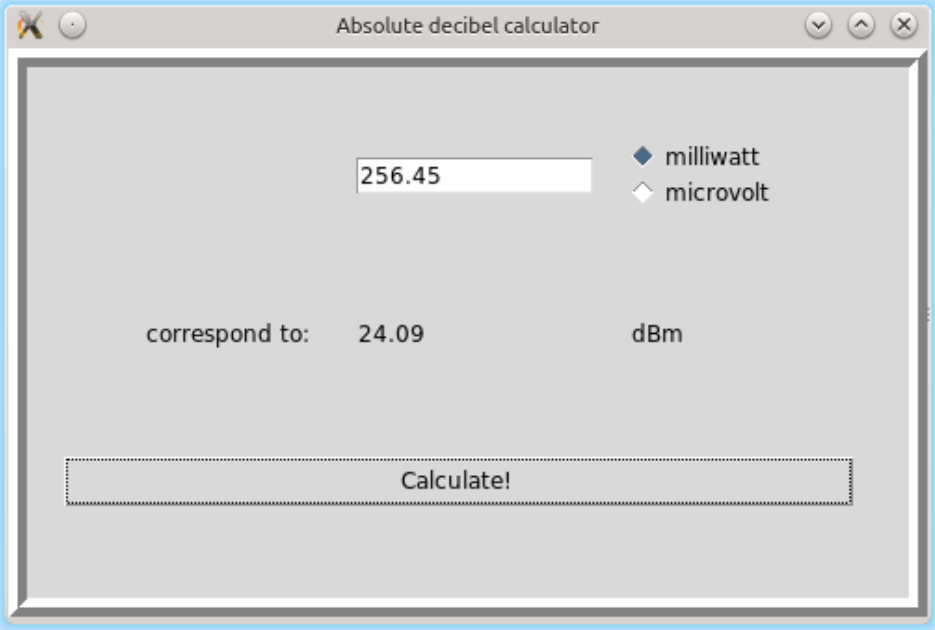
```
butt_1 = ttk.Button(mainFrame, text='Calculate!', command=calculate, width=10)
butt_1.grid(column=1, row=3, sticky=(W, E), columnspan=3)
```

Aufgabe Tk7:

a) Erweitere das Programm indem jede Zelle einen inneren Abstand erhält und der Button sich über 3 Kolonnen erstreckt.

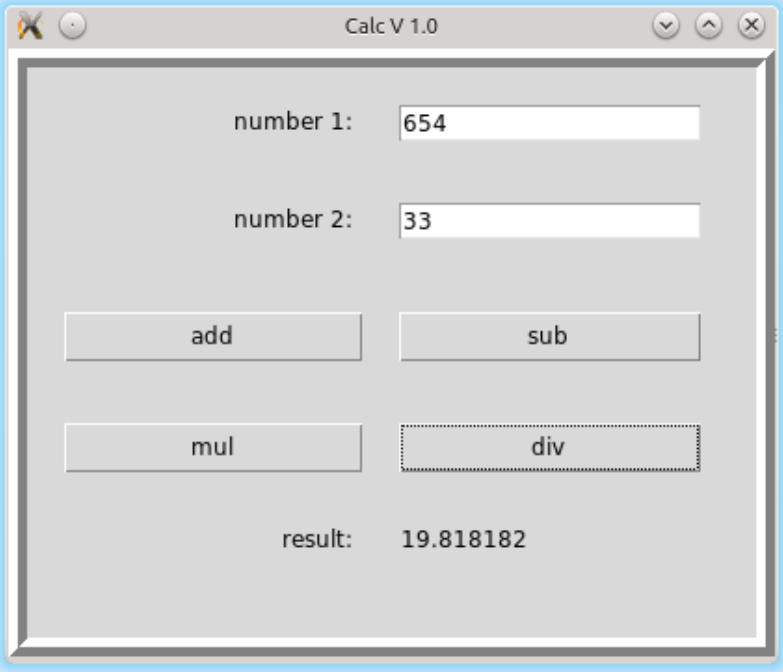
b) Ändere dein Programm so um, dass der erste Label (rechts oben) durch einen Rahmen, der zwei Radiobutton enthält, ersetzt wird. Suche Informationen zum Widget **ttk.Radiobutton** im Netz. Der eine Radiobutton soll eine Berechnung in

dBm (wie bisher, text=’milliwatt’) und der zweite Radiobutton eine Berechnung in dBμV (text='microvolt', in der Formel muss der Multiplikationsfaktor 10 durch 20 ersetzt werden) erlauben. Entsprechend sollen auch der Text des Label mit der Einheit ('dBm' bzw. 'dBμV') geändert werden.



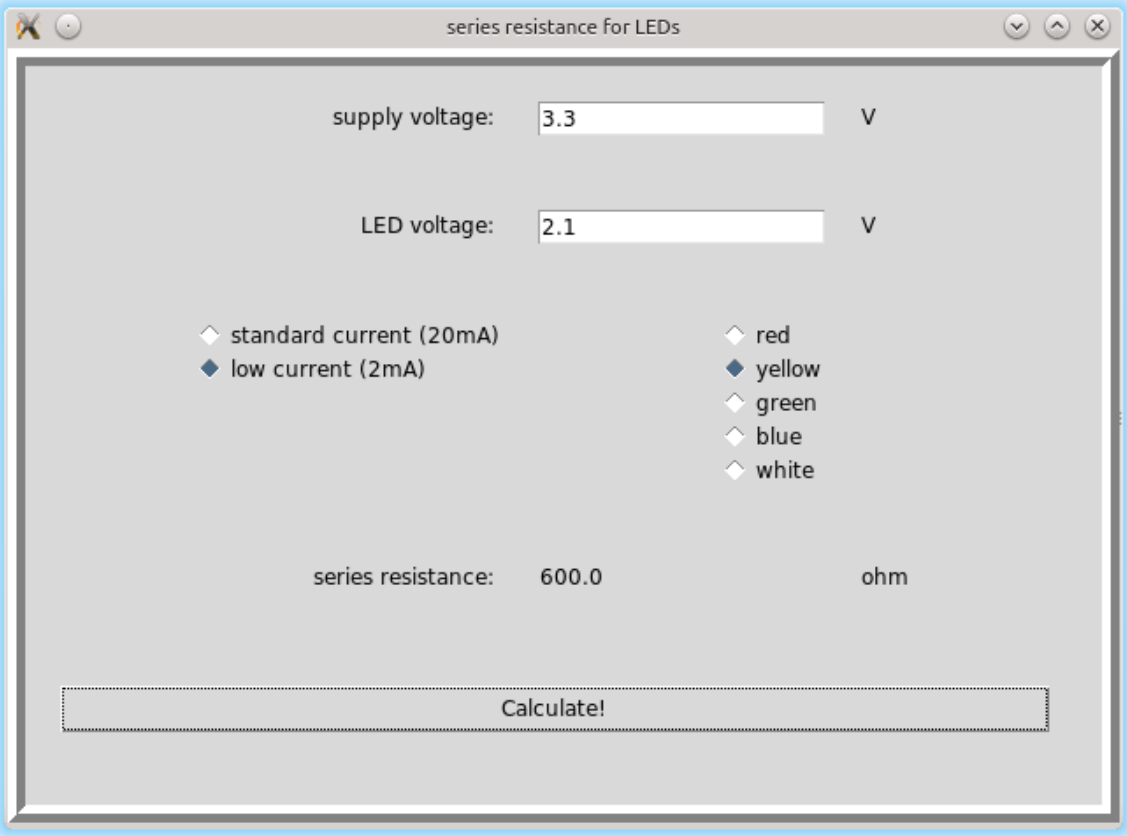
Aufgabe Tk8:

Erstelle ein Programm, das zwei Zahlen mit den Grundrechenarten verknüpfen kann.



Zusatzaufgabe Tk9: (für Fleißige :))

Erstelle ein Programm, das den Vorwiderstand einer LED berechnet. Der Strom wird mit einer Checkbox ausgewählt. Die Spannung an der LED ebenfalls. Sie kann aber wahlweise auch mit einem Entry-Feld manuell eingegeben werden.



Zusatzaufgabe Tk10: (für sehr Fleißige :)))

Programmiere das Spiel TIC TAC TOE. Sobald ein Button gedrückt wird, wird er mit:

```
butt_11.state(["!disabled"])
```

deaktiviert. Eine Zählvariable achtet darauf, dass ein Gleichstand (niemand winnt) erkannt wird. Um das ganze farbiger zu gestalten können "styles" verwendet werden. Eine mögliche Lösung findet man unter: http://www.weigu.lu/c/python/download

- http://www.tutorialspoint.com/python/python_basic_syntax.htm
- http://www.tkdocs.com/tutorial/
- https://docs.python.org/3/library/tkinter.ttk.html
- https://www.tcl.tk/man/tcl/TkCmd/contents.htm
- http://www.python-kurs.eu/python_tkinter.php

Dateioperationen mit Python

Kurze Einführung

Arbeitet man mit Daten, so ist es sinnvoll diese in Dateien abzuspeichern. So bleiben sie auch nach dem Ausschalten des Computers erhalten. Heutige eingebettete Systeme wie der Raspberry Pi besitzen meist eine SD-Karte auf der Dateien abgelegt werden können oder eine USB-Schnittstelle die es ermöglicht Dateien auf einem USB-Stick abzulegen. Mit solchen Systemen lassen sich dann zum Beispiel leicht, mit Hilfe von Python, die Daten eines angeschlossenen Sensors loggen. Sind Daten in Dateien abgespeichert, so müssen sie meist später noch verarbeitet werden. Auch dies kann meist mit einigen Zeilen Python-Code erledigt werden.

In Linux werden auch Geräte wie Dateien angesprochen. Die hier erlernten Kenntnisse werden also auch benötigt um auf Geräte zuzugreifen (zum Beispiel die serielle Schnittstelle).

Dateien öffnen, lesen und schließen

Mit der Methode `open()` wird eine Dateiobjekt erstellt (oft mit dem Namen f für *file*) , die Datei geöffnet, und die Datei dem Dateiobjekt zugewiesen. Die Mehode `read()` des Dateiobjekt lies dann die Datei ein. **Wird eine geöffnete Datei nicht mehr benötigt, sollte sie immer sofort geschlossen werden um den Speicher wieder frei zu geben und anderen Programmen den Zugriff wieder zu erlauben.**

```
#!/usr/bin/env python3
# file_open.py

f = open('akkudata.dat')
data = f.read()
print (data)
f.close()
```

Ausgabe der Datei:

```
29.10.13    09:09:58    1.81    22.87    0.0150833333 41.39      0.34
29.10.13    09:10:28    1.81    22.95    0.0301666667 41.54      0.69
29.10.13    09:10:58    1.8     23.02    0.0451666667 41.44      1.04
29.10.13    09:11:28    1.8     23.1     0.0601666667 41.58      1.38
29.10.13    09:11:58    1.81    23.16    0.07525      41.92      1.73
...
```

Aufgabe F1:

Lade die Datei 'akkudata.dat'
(http://www.weigu.lu/c/python/download/akkudata.dat)

herunter und speichere sie in das Verzeichnis deiner Python Programme. Teste dann das obige Programm in einem Terminalfenster (`python3 file_open.py`).

IO-Fehler abfangen:

Wurde die Datei nicht gefunden, so erhalten wir folgende Fehlermeldung:

```
FileNotFoundError: [Errno 2] No such file or directory: 'akkudata.dat'
```

Um dies zu vermeiden werden wir den Fehler mit der "`try...except`"-Anweisung abfangen. So können wir die Ausnahme (exception) mit einer eindeutigen Fehleraussage sauber dokumentieren. Mit der Methode `exit()` kann man das Programm dann verlassen. Es macht Sinn Dateinamen am Anfang des Programms einer Variablen zuzuweisen. So lassen sie sich später leichter ändern.

```
#!/usr/bin/env python3
# file_open2.py

filename = 'akkudata.dat'

try:
    f = open(filename)
except IOError:
    print('Cannot find file:', filename)
    exit()

data = f.read()
print (data)
f.close()
```

Aufgabe F2:

Teste das Programm der ersten Aufgabe indem du es mit einem falschen Dateinamen aufrufst. Erweitere dann dein Programm um die Fehlerbehandlung. Teste das Programm dann noch einmal mit richtigem und falschem Dateinamen.

Dateien schreiben

Daten werden mit der Methode `write()` geschrieben. Beim Öffnen der Datei muss man zusätzlich angeben, in welchem Modus die Datei geöffnet werden soll. Dabei gibt es folgende Möglichkeiten:

- **`r` (*read*)**
- **`w` (*write*)** Ersetzt den Inhalt einer bestehenden Datei.
- **`a` (*append*)** Hängt den Inhalt ans Ende einer bestehenden Datei an.
- **`r+`** Öffnet Datei zum Lesen und Schreiben (wird nicht oft benutzt).

Gibt man den Parameter nicht an, wird die Datei (wie in unserem Beispiel) zum Lesen geöffnet.

```
#!/usr/bin/env python3
# file_write.py
```

```
filename = 'test.txt'

try:
    f = open(filename, 'w')
except IOError:
    print('Cannot create file:', filename)
    exit()

for i in range(10):
    f.write('Dies ist die ' + str(i) + 'te Zeile\n')
f.close()

# print the file for verification
try:
    f = open(filename)
except IOError:
    print('Cannot find file:', filename)
    exit()

data = f.read()
print (data)
f.close()
```

Aufgabe F3:

Teste das obige Programm. Ändere den Parameter "write" nach "append" und teste das Programm.

Daten in Dateien verändern

Die Daten-Datei im obigen Beispiel ist sehr groß. Dadurch muss das Variablenobjekt "`data`" viel Speicher in Anspruch nehmen. Dies kann bei noch größeren Dateien und eingebetteten Systemen mit wenig RAM zum Problem werden. Außerdem braucht Python viel Zeit um alle Daten einzulesen bevor eine Operation mit den Daten durchgeführt wird. Besser ist es die Daten zum Beispiel zeilenweise zu verarbeiten:

```
#!/usr/bin/env python3
# file_open3.py

filename = 'akkudata.dat'

try:
    f = open(filename)
except IOError:
    print('Cannot find file:', filename)
    exit()

line = f.readline()
while line != '':
    print (line, end='')
    line = f.readline()
f.close()
```

Der zusätzliche Parameter `end=''` in der Print-Anweisung verhindert, dass `print()` eine zusätzliche Leerzeile ausgibt.

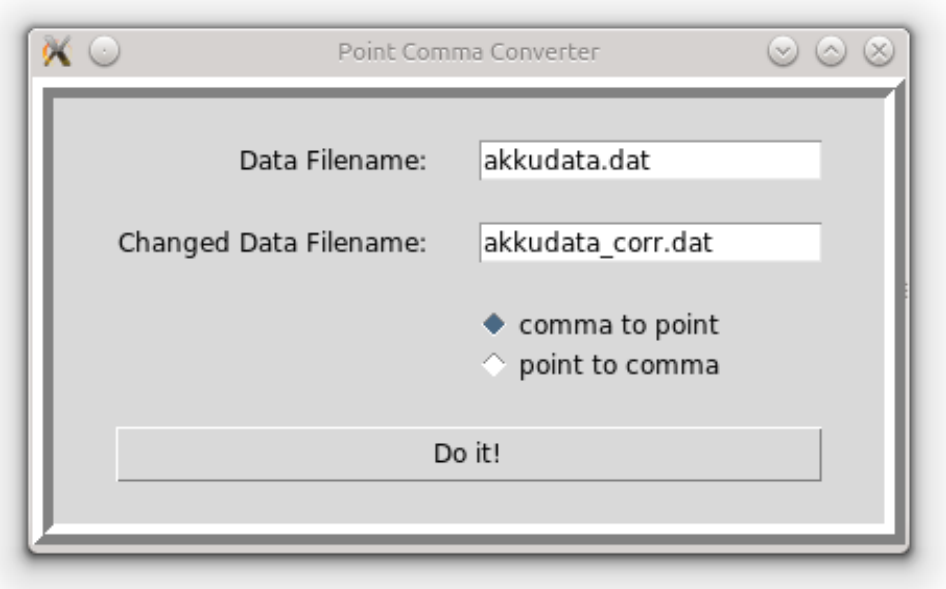
Die verwendete Datei enthält Daten die beim Laden eines Akkus aufgenommen wurden. Die Daten sind durch ein Tabulatorzeichen (ASCII 0x09, `'\t'`) getrennt und jede Zeile endet mit dem Linefeed-Zeichen (LF, ASCII 0x0A, `'\n'`) wie in Linux üblich. Die Datei soll

in einer anderen Software unter Windows weiterverarbeitet werden. Dazu sind folgende Änderungen nötig: Der Punkt soll durch ein Komma-Zeichen ersetzt werden, der Datum-Punkt durch einen Schrägstrich (`'/'`) und das Newline-Zeichen am Ende soll durch einen zusätzlichen Wagenrücklauf (*carriage return*, CR, ASCII 0x0D, `'\r'`) erweitert werden, da Windows beide Zeichen erwartet (siehe http://en.wikipedia.org/wiki/Newline). Dazu wird eine zweite Datei geöffnet, welche dann nach dem Schreiben die veränderte Datei enthält. Das entsprechende Programm kann dann folgendermaßen aussehen:

```
#!/usr/bin/env python3
# file_change.py

filename1 = 'akkudata.dat'
filename2 = 'akkudata_corr.dat'

try:
    f1 = open(filename1)
except IOError:
    print('Cannot find file:', filename)
    exit()
try:
    f2 = open(filename2, 'w')
except IOError:
    print('Cannot create file:', filename)
    exit()

line = f1.readline()
while line != '':
    line = line.replace('.', '/', 2)    # replace two date points with slashes
    line = line.replace('.', ',')
    line = line.replace('\n', '\r\n')  # replace Linefeed with CR+LF
    f2.write(line)
    print (line, end='')
    line = f1.readline()
f1.close()
f2.close()
```

Aufgabe F4:

Schreibe ein Programm das die GUI Tkinter nutzt und es erlaubt die Punkte in einer Datei durch Kommas zu ersetzen oder umgekehrt (**`ttk.Radiobutton`**). Die beiden Dateinamen sind durch das Widget **`ttk.Entry`** veränderbar.



Zusatzaufgabe F5:

Da die originale Daten-Datei (Laden eines Akkus) sehr groß ist, soll über jeweils 10 Werte ein Mittelwert gebildet werden. Auch sollen nur Strom und Spannung (dritte und vierte Kolonne) verwendet werden. Schreibe das entsprechende Programm (ohne GUI). Die neue Datei soll `akkudata_short.dat` heißen.

Tipp: Mit der Methode `split()` kann man die Zeile aufspalten (hier beim Tabulator) und in eine Liste umwandeln.

```
spline=line.split('\t')
```

Auf den Strom kann dann zum Beispiel mit `spline[2]` zugegriffen werden.

Schnittstellen mit dem Raspberry Pi

Kurze Einführung

Mit den einzelnen Pins des Raspberry Pi (Raspi) lassen sich bequem Schalter einlesen und LEDs oder Relais einschalten. Werden die Aufgaben komplexer, so greift man auf spezialisierte Bausteine zurück. A/D- oder D/A-Wandler-Bausteine ermöglichen das Verarbeiten analoger Daten. Temperatur-, Feuchtigkeit-, Luftdruck-Sensoren liefern uns Umweltwerte. Port- Expander erweitern das Angebot an vorhandenen Pins. Farbige graphische Displays erleichtern die Kommunikation mit dem Anwender. Es existieren spezialisierte Bausteine für unterschiedliche serielle Schnittstellen. Meist trifft man hier auf I2C, SPI oder 1-Wire. Linux unterstützt diese Schnittstellen auf dem Raspi, so dass sich diese spezialisierten Bausteine nutzen lassen.

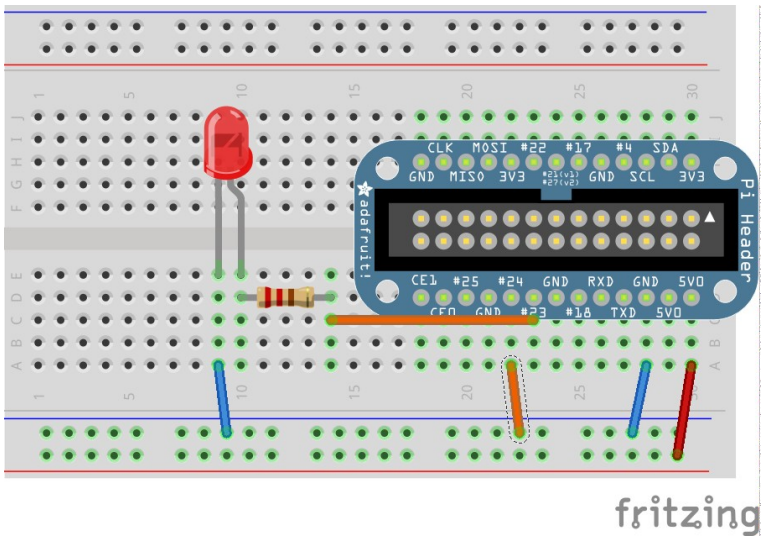
Die serielle Schnittstelle (EIS232) wird heute noch häufig zur Kommunikation (meist ASCII-Daten) verwendet. Auch sie soll mit dem Raspi in Betrieb genommen werden.

Digitale Ein-/Ausgabe (GPIOs)

Die einzelnen Pins des Raspi können unterschiedliche Aufgaben wahrnehmen (*GPIO General Purpose Input Output*). Oft werden sie als einfache digitale Ein- bzw. Ausgänge genutzt um LEDs einzuschalten oder Schalter abzufragen (Wiederholung 12. Klasse!).

Es ist hierbei darauf zu achten, dass die maximale Spannung von 3,3V nicht überschritten wird, und auch kein nennenswerter Strom geliefert werden kann. Sind höhere Ströme oder Spannungen nötig, so ist ein Treiber-IC wie der ULN2008 o.ä. zu nutzen.

!!! Achte beim Verdrahten darauf keine Kurzschlüsse zu verursachen. Die GPIO- Pins des Raspi sind nicht so robust wie die verschiedener Mikrocontroller! Auf www.weigu.lu/c/rpibuffer wird eine Schutzplatine mit kurzschlussfesten Buffer-Bausteinen vorgestellt, die hier genutzt werden soll um die Raspis nicht zu zerstören.



Die Python-Bibliothek **RPi.GPIO** ermöglicht den direkten Zugriff auf die GPIOs.

Für die Beschriftung der Raspi Pins werden die Chip-Nummern des Raspi (*Broadcom SOC channel numbers*) genutzt die oft mit dem Vorsatz GPIO auftauchen, und nicht die Pinnummern des Raspi-Board! Dies wird mit der Methode **setmode** festgelegt.

Die Methode **setup** legt fest ob das jeweilige Pin als Ein- oder Ausgang verwendet wird. Die Methode **output** schaltet das Pin auf **HIGH** bzw. **LOW**. Mit der Methode **cleanup** wird das Pin wieder rückgesetzt (kein Ausgang mehr).

Das folgende Programm lässt eine LED an GPIO23 10 mal blinken:

```
#!/usr/bin/env python3
# gpio_blink.py

import RPi.GPIO as GPIO # sudo apt-get install rpi.gpio
from time import sleep

GPIO.setmode(GPIO.BCM) #Broadcom SOC channel number (numbers after GPIO)

GPIO.setup(23, GPIO.OUT)

for i in range(0,10):
    GPIO.output(23, GPIO.HIGH)
    sleep(1)
    GPIO.output(23, GPIO.LOW)
    sleep(1)

GPIO.cleanup()
```

Das nächste Programm fragt einen Schalter (Methode **input**) ab und schaltet eine LED am GPIO23 ein, wenn der Schalter (hier Verbindungsdraht) mit Masse verbunden wird.

```
#!/usr/bin/env python3
# gpio_read.py

import RPi.GPIO as GPIO # sudo apt-get install rpi.gpio
from time import sleep

GPIO.setmode(GPIO.BCM) #Broadcom SOC channel number (numbers after GPIO)


GPIO.setup(24, GPIO.IN)
GPIO.setup(23, GPIO.OUT)

while (1):
    iv = GPIO.input(24)
    if iv == 0:
        GPIO.output(23, GPIO.HIGH)
    else:
        GPIO.output(23, GPIO.LOW)
    sleep(0.5)
```

Die 1-Wire Schnittstelle

Der 1-Wire-Bus wurde von der Firma Dallas Semiconductor Corp. entwickelt. Es handelt sich um eine serielle asynchrone Schnittstelle, bei der mindestens 2 Leitungen vorhanden

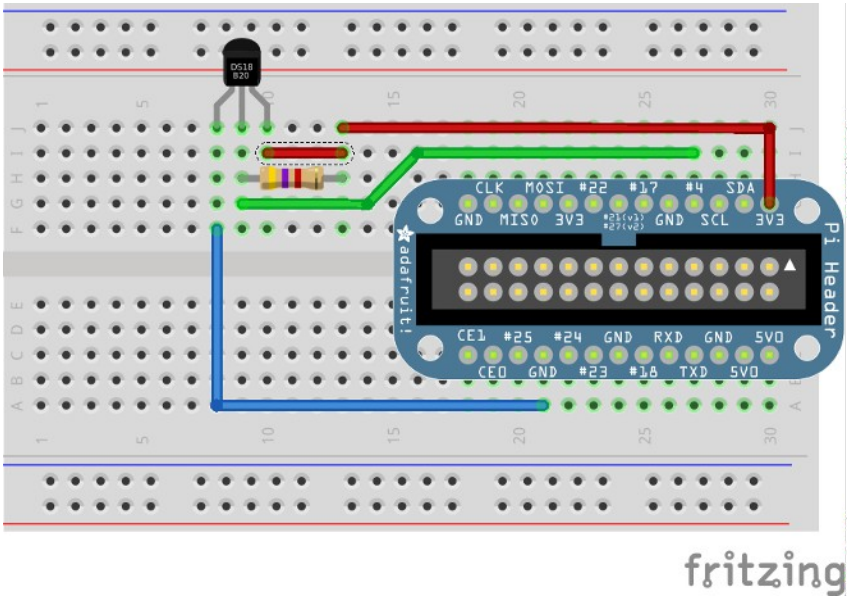
sind (Datenleitung und Masse). Die Spannungsversorgung (2,8-6V) kann über die Datenleitung erfolgen, da jeder Baustein einen internen Kondensator besitzt, der durch die vorhandene Spannung auf der (inaktiven) Datenleitung aufgeladen wird. Es wird über die gleiche Datenleitung im Halbduplex-Verfahren gesendet und empfangen. An der Leitung kann ein Master und bis zu 100 Slaves angeschlossen werden. Jeder Slave besitzt eine eindeutige 64-Bit-Adresse (8-Bit-Family-Code, 48-Bit-Seriennummer, 8 Bit CRC Checksumme), die fest einprogrammiert ist.

Temperaturmessung mit dem DS18B20

Wir verwenden hier den Temperatursensor DS18B20. Dieser Sensor kann für Temperaturen von -55°C bis +125°C verwendet werden. Die Genauigkeit ist hoch (weniger als ±0.5°C Fehler zwischen -10°C to +85°C). Der interne A/D-Wandler nutzt 12 Bit und die Erfassung und Wandlung des Temperaturwertes benötigt rund 750ms.

Die Datenleitung ist im inaktivem Zustand High und wird vom Master (hier unser Raspi) auf Masse gezogen um Daten zu übertragen. Der 1-Wire-Linuxtreiber des Raspi ist so konfiguriert, dass er die Datenleitung auf Pin Nr 4 erwartet. Mit Hilfe eines externen Pull-Up Widerstandes ziehen wir die Datenleitung auf 3,3V. Da der typische Strom des Sensors um 1mA liegt, ist ein Widerstand von 4,7k eine gute Wahl. Der Temperatursensor wird zusätzlich extern mit Spannung versorgt (3 Leitungen), da diese Variante eine robustere Datenübertragung erlaubt.

!!! Achte beim Verdrahten darauf keine Kurzschlüsse zu verursachen. Die GPIO- Pins des Raspi sind nicht so robust wie die verschiedener Mikrocontroller!
Wird der rpibuffer-Adapter (weigu.lu/c/rpibuffer) genutzt, so ist der Jumper auf 3.3V zu setzen.



Die Treiber für den 1-Wire-Bus des Raspi sind nicht im Kernel fest eingebunden, sondern müssen über so genannte Module nachgeladen werden. Dies ist möglich von Hand mit den Befehlen:

```
modprobe w1-gpio
modprobe w1-therm
```

Diese Befehle kann man jeweils vor dem Starten des Programms auf der Kommandozeile ausführen. Man kann sie aber zum Beispiel auch im Python-Programm mit Hilfe der Bibliothek `os` (*operating system*) aufrufen.

```
import os

os.system('modprobe w1-gpio')
os.system('modprobe w1-therm')
```

Mit dem Befehl:

```
lsmod
```

(*list modules*) kann dann kontrolliert werden ob die jeweiligen Module geladen wurden.

Mit Einführung des Kernels 3.18.3 wird das für Linux empfohlene Device-Model (Gerätemodell) für den Raspberry Pi genutzt. Der ***device tree***, abgekürzt auch DT, wird beim Raspberry Pi in der Datei `/boot/config.txt` angesprochen. Damit der 1-Wire Bus funktioniert ist folgende Zeile

```
dtoverlay=w1-gpio,gpiopin=4
```

in der Datei "`/boot/config.txt`" hinzuzufügen.

Dadurch werden die entsprechenden Module beim Starten automatisch geladen.

Aufgabe I1:

Trage die Zeile **`dtoverlay=w1-gpio,gpiopin=4`** in die Datei **`/boot/config.txt`** ein. Nutze dazu einen Editor deiner Wahl mit Root-Rechten (zB: "`sudo leafpad`" oder "`sudo nano`"). Speichere die Datei und boote den Raspi neu mit "`sudo reboot`".

Bemerkung: Auf der Kommandozeile kann man sehr bequem mit dem "midnight commander" (mc) arbeiten. Sollte er nicht installiert sein, so installiere ihn mit "`sudo apt-get install mc`". Mit "`sudo mc`" hat man Root-Rechte und kann mit F4 Dateien editieren. Mit Ctrl+O kann man zwichen mc und Terminal hin- und herschalten.

Ist der Sensor richtig verdrahtet, so kümmert der Treiber sich um das Abfragen der eindeutigen Adresse und legt ein Verzeichnis für den Sensor mit dieser Adresse im Unterverzeichnis

```
/sys/bus/w1/devices
```

an. In diesem Verzeichnis befindet sich dann die Gerätedatei mit dem Namen `w1_slave`. Den Inhalt der Datei kann man sich dann mit dem cat-Befehl anzeigen lassen.

```
pi@T3EC-13 /sys/bus/w1/devices/28-0000068e3d97 $ cd /sys/bus/w1/devices
pi@T3EC-13 /sys/bus/w1/devices $ ls
28-0000068e3d97  w1_bus_master1
pi@T3EC-13 /sys/bus/w1/devices $ cd 28-0000068e3d97
pi@T3EC-13 /sys/bus/w1/devices/28-0000068e3d97 $ ls
driver  id  name  subsystem  uevent  w1_slave
pi@T3EC-13 /sys/bus/w1/devices/28-0000068e3d97 $ cat w1_slave
52 01 4b 46 7f ff 0e 10 ff : crc=ff YES
52 01 4b 46 7f ff 0e 10 ff t=21125
pi@T3EC-13 /sys/bus/w1/devices/28-0000068e3d97 $
```

```
#!/usr/bin/env python3
# interface_w1_ds18b20_1.py

# add dtoverlay=w1-gpio,gpiopin=4 the file /boot/config.txt and reboot

import time

deviceFile = '/sys/bus/w1/devices/28-0000068e3d97/w1_slave'

try:
    f = open(deviceFile, 'r')
except IOError:
    print('IOError')
line1 = f.readline()
print(line1, end='')
line2 = f.readline()
print(line2, end='')
f.close()
time.sleep(1)
```

Das Programm kann mit der Tastenkombination `Ctrl+c` (bzw. `Strg+c`) abgebrochen werden.

Nachdem die Datei geöffnet wurde werden 2 Zeilen eingelesen. Mit Hilfe der `print()`-Funktion sehen wir uns den Inhalt an:

```
pi@raspberrypi ~/INFAP $ python3 interface_w1_ds18b20_1.py
68 01 4b 46 7f ff 08 10 05 : crc=05 YES
68 01 4b 46 7f ff 08 10 05 t=22500
```

Die benötigte Information befindet sich in der 2. Zeile hinter dem String `'t='`. Mit der `find()`-Methode lässt sich der Temperatur-String extrahieren und in eine Zahl umwandeln. Die ganze Temperaturerfassung packen wir in eine Funktion. Eine mögliche Version des Programms könnte dann wie folgt aussehen:

```
#!/usr/bin/env python3
# interface_w1_ds18b20_2.py

# add dtoverlay=w1-gpio,gpiopin=4 the file /boot/config.txt and reboot

import time
```

```
deviceFile = '/sys/bus/w1/devices/28-0000068e3d97/w1_slave'


def readTemp():
    try:
        f = open(deviceFile, 'r')
    except IOError:
        print('IOError')
    line1 = f.readline()
    line2 = f.readline()
    f.close()
    pos = line2.find('t=')
    if pos != -1:
        tempString = line2[pos + 2:]
        temp = round(float(tempString) / 1000.0, 1)
    else:
        print('error')
    return temp
while True:
    print(str(readTemp())+' °C')
    time.sleep(1)
```

Aufgabe I2:

Das manuelle Ausspähen der Gerätedatei ist lästig, wenn unterschiedliche Sensoren getestet werden sollen. Um dies zu automatisieren kann man das glob-Modul von Python verwenden, das uns ermöglicht nach Pfadnamen mit Hilfe von Wildcards zu suchen. Da die Adresse des DS18B20 immer mit 28 beginnt (*Family-Code*), ermitteln wir so das Verzeichnis (die Adressen des DS18S20 beginnen mit 10). Die Methode glob gibt eine Liste zurück. Der String befindet sich im ersten Element der Liste, auf die wir mit [0] zugreifen. Erweitere dein Programm um folgende Zeilen und teste es.

```
import glob

deviceFolder = glob.glob('/sys/bus/w1/devices/28*')
deviceFolderString = deviceFolder[0]
deviceFile = deviceFolderString + '/w1_slave'
```

Aufgabe I3:

Erweitere dein Programm, so dass es die Temperatur von zwei Temperatursensoren erfassen und ausgeben kann. Die Ausgabe soll folgendermaßen aussehen:

```
pi@raspberrypi ~/INFAP $ python3 interface_I2_a_2DS18B20.py
1. Sensor: Adresse = 28-00000360812d Temperatur = 23.4 °C
2. Sensor: Adresse = 28-0000036086d2 Temperatur = 22.7 °C
```

Aufgabe I4: (für Fleißige)

Schreibe ein Programm mit graphischer Oberfläche, das es ermöglicht die Temperatur in Grad Celsius, Kelvin oder Fahrenheit anzeigen zu lassen (Checkboxes).

Aufgabe I5: (für sehr Fleißige)

Erweitere das GUI-Programm um ein Auswahlfeld, das alle vorhandenen Sensoren mit ihrer Adresse anzeigt.

Die I²C Schnittstelle

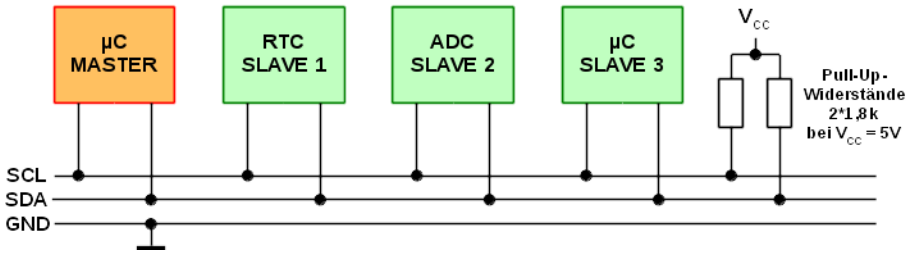
Die von Philipps entwickelte serielle synchrone I²C-Master-Slave-Schnittstelle, (*Inter-Integrated Circuit*, gesprochen **I-Quadrat-C**) dient der Kommunikation zwischen verschiedenen integrierten Schaltkreisen (IC, Integrated Circuit, Chip). Sie wurde für die Unterhaltungselektronik entwickelt (Fernsehgeräte) und ist dort weit verbreitet (viele ansteuerbare Spezial-ICs).

Vorteile des I²C-Busses sind der geringe Verdrahtungsaufwand und die geringen Kosten bei der Entwicklung eines Gerätes. Es werden nur drei Leitungen benötigt. Ein Mikrocontroller kann so ein ganzes Netzwerk an ICs mit nur drei Leitungen und einfacher Software kontrollieren. Dies senkt die Kosten des zu entwickelnden Gerätes. Während des Betriebes können Chips zum Bus hinzugefügt oder entfernt werden (*hot-plugging*).

Nachteile des I²C-Busses sind die geringe Geschwindigkeit und die geringe überbrückbare Distanz. Daten können nur abwechselnd über die Datenleitung gesendet werden (Halbduplex) und zusätzlich zu den Daten müssen die Adressen der Bausteine versendet werden.

Verwendung:
Der I²C-Bus wird meist zur Übertragung von Steuer- und Konfigurationsdaten verwendet, da es dabei meist nicht auf Schnelligkeit ankommt. Er wird zum Beispiel verwendet bei Echtzeituhren, Lautstärkereglern, Sensoren, A/D- und D/A-Wandlern mit niedriger Abtastrate, EEPROM Speicherbausteinen oder bidirektionale Schaltern und Multiplexern. Große Bedeutung hatte das I²C- Protokoll in der Vergangenheit im Chipkartenbereich. Er ist nicht geeignet für größere Entfernungen, da die Störsicherheit gering ist und zu Fehlern in der Übertragung führt.

Der I²C-Bus ist als synchrone Master-Slave-Schnittstelle konzipiert (es ist allerdings auch möglich mehrere Master einzusetzen (multimaster mode)). Die Buszuteilung ist dabei in den Spezifikation geregelt und verhindert Kollisionen. Im Normalfall sendet der Master (bei uns der Raspi) und ein Slave reagiert darauf. Es können je nach verwendeten ICs vier Geschwindigkeiten eingesetzt werden: **100 kHz (*standard mode*) 400 kHz (*fast mode* ; erweiterter Adressraum) 1 MHz (*fast mode plus*) 3,4 MHz (*high-speed mode*).** Der Raspi arbeitet default mit 100 kHz. Es ist aber möglich die Geschwindigkeit zu verändern (google).

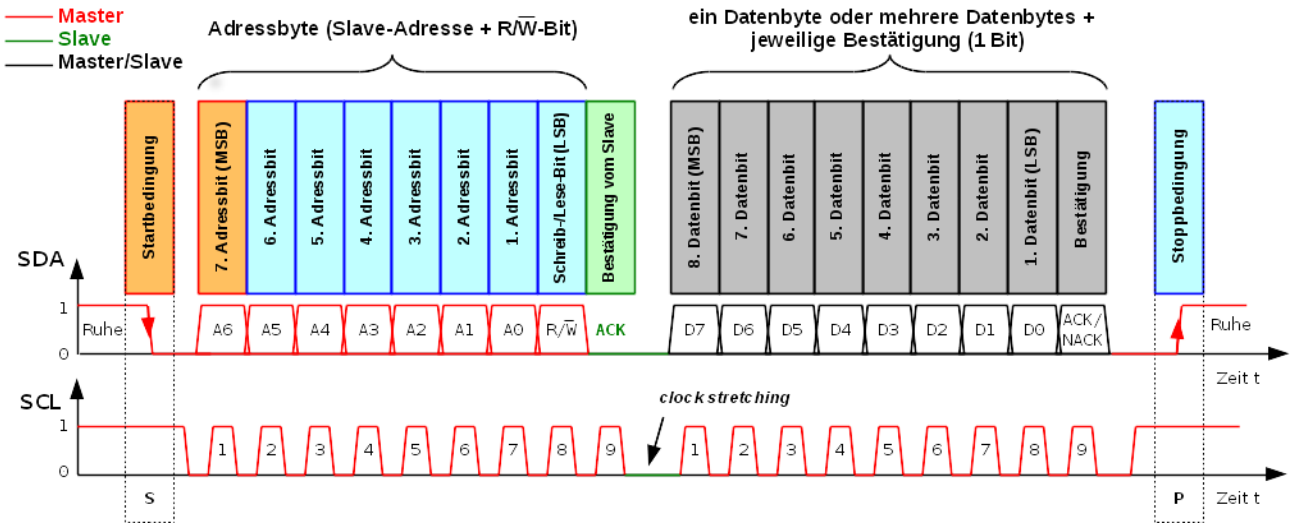


Im obigen Bild sind ein Master und drei Slaves eingezeichnet. Der synchrone I²C- Bus benötigt eine **Taktleitung** (*serial clock line*, **SCL**) und eine **Datenleitung** (*serial data line*, **SDA**). Jedes Datenbit auf der SDA- Leitung wird mit dem Takt der SCL-Leitung

synchronisiert. Die Pull-Up-Widerstände an der Takt- und Datenleitung ziehen beide Leitungen im Ruhezustand auf High-Pegel. Alle am Bus angeschlossene Bausteine besitzen einen Open-Collector- oder Open- Drain-Ausgang (der Kollektor bzw. Drain eines Transistors ist unbeschaltet (offen) und wird durch den gemeinsamen Pull-Up Widerstand des Busses mit VCC verbunden). Ist der bipolare Transistoren bzw. der FET durchgeschaltet, so wird der Bus auf Masse gezogen. Man nennt einen solche Verschaltung auch noch eine Wired-And-Verknüpfung da die Schaltung wie ein Und-Gatter wirkt.

Das I²C-Protokoll und die Adressierung

Mit einer fallende Flanke auf SDA (SCL = High) startet der Master die Kommunikation. Nach dem Startbit sendet der Master als erstes das Adressbyte an den Slave. Das Adressbyte besteht aus einer 7-Bit Slave-Adresse und einem Schreib-Lese-Bit, welches die Richtung der Kommunikation festlegt. Der Slave bestätigt den korrekten Empfang mit einem ACK-Bestätigungsbit (*ACKnowledgement*). Der Master erzeugt die 9 Taktimpulse und liest dann die Taktleitung. Hier kann dann ein langsamer Slave mit einem Low-Pegel eine Wartezeit erzwingen (*clock stretching*).



Je nach Richtung der Kommunikation sendet jetzt der Master oder der Slave beliebig viele Datenbytes (8 Bit, MSB first). Jedes Datenbyte wird vom Gegenüber mit einem ACK-Bit (Low-Pegel) bestätigt. Die Übertragung wird durch das Senden eines NACK-Bits (Not ACKnowledge, High-Pegel) vom Master oder Slave abgebrochen. Mit einer steigenden Flanke auf SDA (SCL = High) gibt der Master den Bus wieder frei (Stoppbit).

Um Zeit zu sparen kann der Master auch den Bus nicht freigeben (kein Stoppbit) und gleich mit einem weiteren Startbit (*Repeated Start*) eine neue Kommunikation starten. Die Kommunikationsrichtung kann hierbei natürlich beliebig geändert werden. Das vom Master gesendete Adressbyte besteht, wie beschrieben, aus sieben Bit die die eigentliche Adresse des Slave darstellen und einem achten Bit das die Lese- oder Schreibrichtung festlegt. Die I²C-Schnittstelle nutzt einen Adressraum von 7 Bit, womit gleichzeitig 112 Bausteine auf einem Bus angesprochen werden können (16 der 128 möglichen Adressen sind für Sonderzwecke reserviert). Jeder I²C- fähige Baustein (IC) hat eine festgelegte Adresse. Bei

manchen ICs kann ein Teil der Adresse hardwaremäßig mittels Steuerpins festgelegt werden. So können z.B. bis zu acht gleichartige ICs an einem I²C-Bus betrieben werden. Immer häufiger kann die Adresse aber auch softwaremäßig umprogrammiert werden (z.B. bei digitalen Sensoren). Es besteht auch noch eine neuere alternative 10 Bit-Adressierung (1136 Bausteine). Sie ist abwärtskompatibel zum 7 Bit-Standard (nutzt zusätzlich 4 der 16 reservierten Adressen).

I²C mit dem Raspberry Pi

Um den I²C Bus nutzen zu können müssen die Kernelmodule **`i2c-dev`** und **`i2c-bcm2708`** geladen werden. Am einfachsten geht das wenn wenn man

```
sudo raspi-config
```

aufruft.

Im achten Menupunkt "*8 Advanced Options*" wählen wir Punkt "*A7 I2C*" und schalten die Option mit *Yes* ein. Dann verlassen wir das Programm mit *Finish*. Nach einem Neustart werden die Module dann automatisch geladen (Kontrolle mit **`lsmod`**).

(Alternativ muss von Hand **`dtparam=i2c_arm=on in`** die Datei **`/boot/config.txt`** eingetragen werden (i2c-bcm2708) und **`i2c-dev`** in die Datei **`/etc/modules.`** Eventuell muss auch noch der Eintrag **`blacklist i2c-bcm2708`** mit einem Hash **`#`** (*number sign*) in der Datei **`/etc/modprobe.d/raspi-blacklist.conf`** auskommentiert werden, so dass er nicht mehr gültig ist.)

Weiter benötigen wir die Pakete **`i2c-tools`** und **`python-smbus`**. Sie werden installiert mit:

```
sudo apt-get install i2c-tools python-smbus python3-smbus
```

Damit die Programme nicht mit Root-Rechten ausgeführt werden müssen, muss der Benutzer pi zur Gruppe `i2c` gehören. Dies erledigen wir mit:

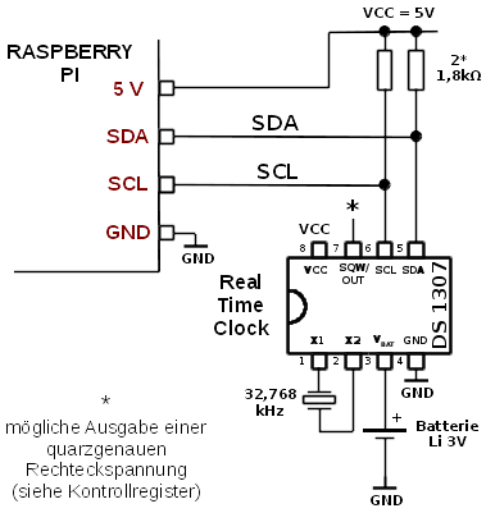
```
sudo adduser pi i2c
```

Aufgabe I6:

Erledige alle Einstellungen damit der I²C-Bus beim Raspi mit Python 3 genutzt werden kann. Achte auf die Kleinschrift. Nutze dazu einen Editor deiner Wahl mit Root-Rechten (zB: "`sudo leafpad`" oder "`sudo nano`").Boote danach den Raspi neu mit `sudo reboot`.

Die Echtzeituhr (RTC) DS1307

Das Senden und Empfangen von Daten über den I²C-Bus soll mit einer Echtzeituhr (*Real Time Clock*) getestet werden. Eine Echtzeituhr läuft auch ohne externe Spannungsversorgung mit einer Batterie (üblicherweise Lithium-Knopfzelle mit 3 V) weiter. Es wird der I²C Baustein DS1307 von Maxim als Echtzeituhr verwendet. Zur äußeren Beschaltung wird nur ein Uhrenquarz (32,768 kHz) und die Batterie benötigt. Der DS1307 besitzt auch einen Ausgang (SQW/OUT) mit dem ein quarzgenauer Takt ausgegeben werden kann. Die I²C-Adresse des Bausteins ist 0x68. Der DS1307 arbeitet nur im Standard Mode (100 kHz).

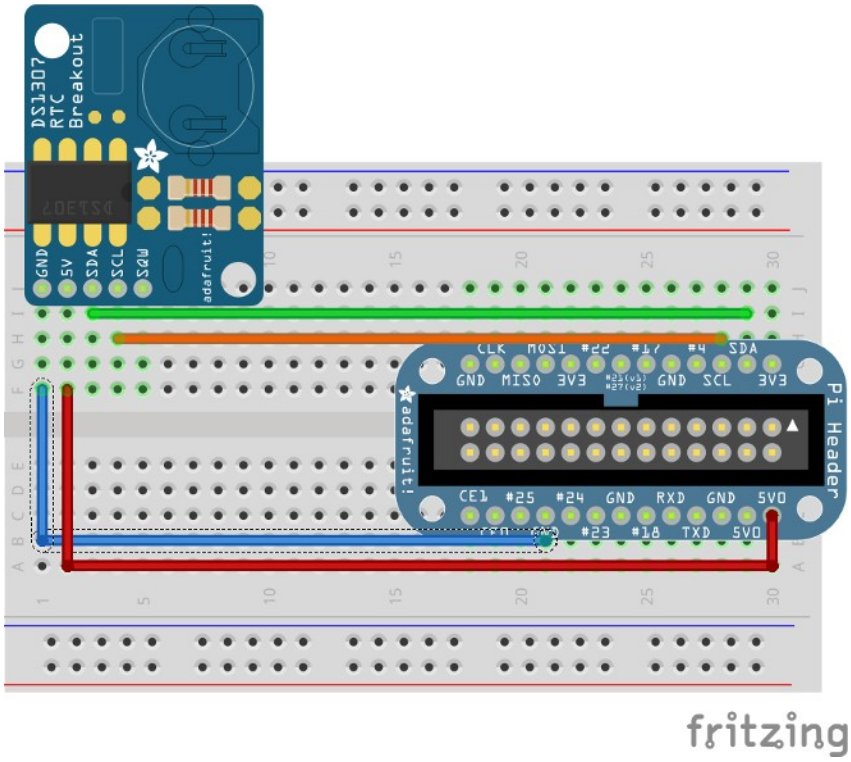


Wichtig: Wird keine Batterie angeschlossen, so muss Pin 3 mit Masse verbunden werden, damit der Baustein angesprochen werden kann.

Der Uhrenbaustein besitzt 64 Register (RAM-Speicherzellen), welche über eine Registeradresse angesprochen werden können. Die ersten sieben Register enthalten die Daten der Uhr (Uhrzeit (3), Wochentag (1) und Datum (3)). Das achte Register dient als Kontrollregister. Die restlichen 56 Speicherzellen können beliebig beschrieben und gelesen werden (gepuffertes RAM). Uns interessieren hier besonders die ersten sieben Register. Sobald das Sekundenregister beschrieben wurde (Bit 7 (CH) = 0) läuft die Uhr. Die Daten werden im **BCD-Code** abgelegt. Der BCD-Code (***Binary Coded Decimal***) ist ein Code mit dual kodierten Dezimalziffern; 4 Bit (Nibble) stellen eine Dezimalziffer (0-9) im Dualcode dar (0b0000-0b1001)).

ADDRESS	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	FUNCTION	RANGE
00H	CH	10 Seconds			Seconds				Seconds	00–59
01H	0	10 Minutes			Minutes				Minutes	00–59
02H	0	12	10 Hour	10 Hour	Hours				Hours	1–12 +AM/PM 00–23
		24	PM/AM							
03H	0	0	0	0	0	DAY			Day	01–07
04H	0	0	10 Date		Date				Date	01–31
05H	0	0	0	10 Month	Month				Month	01–12
06H	10 Year				Year				Year	00–99
07H	OUT	0	0	SQWE	0	0	RS1	RS0	Control	—
08H-3FH									RAM 56 x 8	00H–FFH

!!! Achte beim Verdrahten darauf keine Kurzschlüsse zu verursachen. Die GPIO- Pins des Raspi sind nicht so robust wie die verschiedener Mikrocontroller!
Wird der rpibuffer-Adapter (weigu.lu/c/rpibuffer) genutzt, so ist der Jumper auf 3.3V zu setzen.



Nachdem der Baustein angeschlossen wurde können wir mit Hilfe der **`i2c-tools`** testen ob alles klappt. Dazu geben wir das Kommando:

```
i2cdetect -y 1
```

ein falls wir eine moderne Version des Raspi (Model B rev. 2) besitzen. Die Eins erstzen wir durch eine Null bei einer älteren Version (Model B rev. 1). Die Ausgabe zeigt uns alle Adressen der angeschlossenen I²C-Bausteine. In unserem Fall also die Adresse 0x68 des RTC.

```
pi@raspberrypi ~ $ sudo i2cdetect -y 0
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- 68 -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
pi@raspberrypi ~ $
```

Aufgabe I7:

Teste ob dein Baustein erkannt wird.

Ein Programm um den RTC-Baustein anzusprechen kann dann wie folgt aussehen:

```
#!/usr/bin/env python3
# interface_i2c_ds1307_1.py
#
# install i2c-tools and smbus with:
# sudo apt-get install i2c-tools python-smbus python3-smbus
# run: sudo raspi-config
# enable I2C (8 Advanced Options, A7 I2C) and reboot
# add user pi to group i2c: sudo adduser pi i2c and reboot
# test with:                i2cdetect -y 0 (for Pi Model B rev. 1)
#                           i2cdetect -y 1 (for Pi Model B rev. 2)

import smbus
from time import sleep

port = 1                # (0 for rev.1, 1 for rev 2!)
bus = smbus.SMBus(port)
rtcAddr = 0x68


def bcd2str(d):         # // for integer division; % for modulo
    if (d <= 9):
        return '0' + str(d)
    else:
        return str(d // 16) + str(d % 16)

# set clock (BCD: sec,min,hour,weekday,day,mon,year)
td = [0x00, 0x05, 0x08, 0x01, 0x07, 0x01, 0x15]
bus.write_i2c_block_data(rtcAddr, 0, td)

while True:
    rd = bus.read_i2c_block_data(rtcAddr, 0, 7)
    print (bcd2str(rd[4]) + '/' + bcd2str(rd[5]) + '/' + bcd2str(rd[6]) + \
           '  ' + bcd2str(rd[2]) + ':' + bcd2str(rd[1]) + ':' + bcd2str(rd[0]))
    sleep(1)
```

Nachdem das Modul `smbus` geladen wurde, kann die Uhr gesetzt werden. Dazu wird der smbus-Befehl

```
bus.write_i2c_block_data(rtcAddr, 0, td)
```

verwendet um einen ganzen Block von Daten zu senden. Der erste Parameter ist die I²C-Adresse. Der zweite Parameter übergibt ist ein Kommando vom Master. In unserem Fall kann man hier die Anfangsadresse des RTC-Adresszeigers angeben (er wird bei der

Übergabe der Daten automatisch inkrementiert). Wir übergeben als Anfangsadresse Null, damit der Speicher ab der Sekundenadresse geschrieben wird (mit einer Eins würden wir bei den Minuten beginnen.) Der dritte Parameter ist dann eine Sequenz mit den zu schreibenden Daten. ! Damit die Uhr anläuft muss der Sekundenwert geschrieben werden (Bit 7 = 0). Die Uhr wird natürlich nur beim ersten Aufruf des Programms gesetzt. Danach soll die Zeile zum Schreiben mit einem Kommentarzeichen versehen werden. In der folgenden Endlosschleife wird die Uhr im Sekundentakt gelesen und Datum sowie Uhrzeit ausgegeben. Dies passiert mit dem Kommando

```
rd = bus.read_i2c_block_data(rtcAddr, 0, 7)
```

Die beiden ersten Parameter sind die gleichen wie oben. Der dritte Parameter gibt die Anzahl der zu lesenden Bytes an. Da die Daten in BCD vorliegen müssen sie noch in einen String umgewandelt werden. Dies übernimmt die Funktion einfache `bcd2str()`, mit Hilfe der Integer-Division und der Modulo-Operation.

Aufgabe I8:

Erweitere das Programm, so dass auch noch der Wochentag ausgegeben wird.

Aufgabe I9: (für Fleißige)

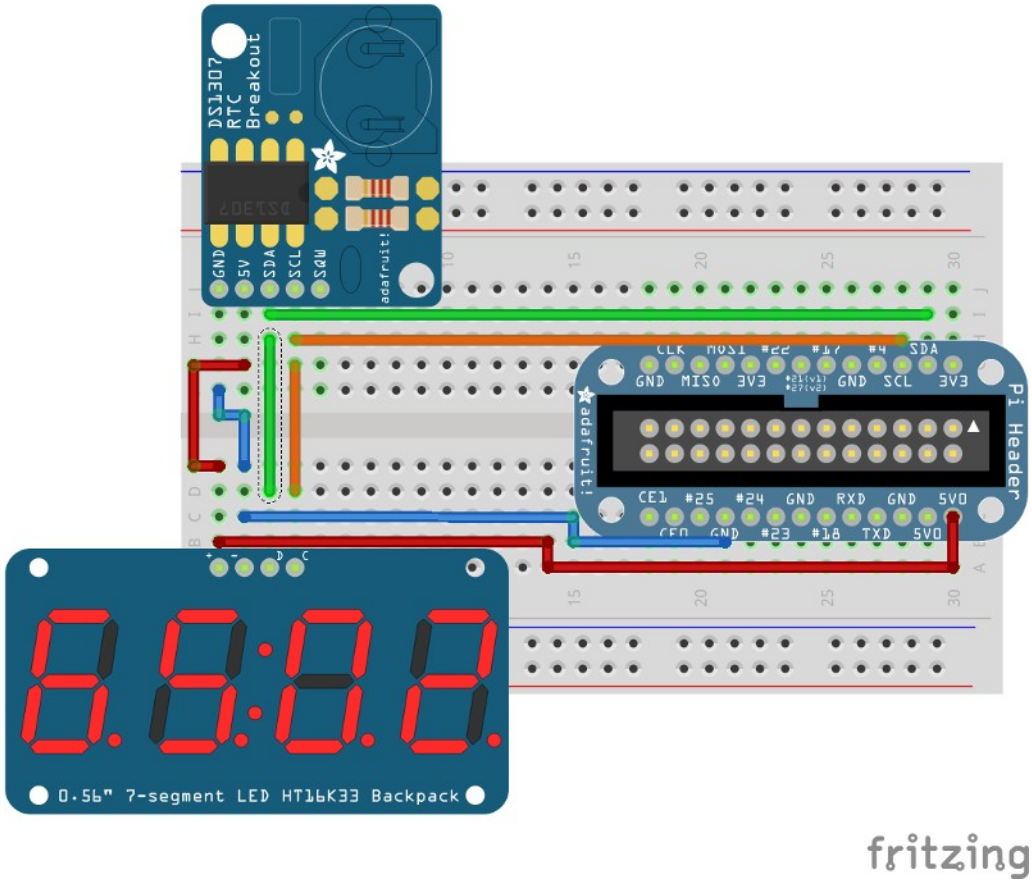
a) Um die Uhrzeit zu setzen soll die aktuelle Uhrzeit des Raspi ermittelt werden. Dies kann mit dem Modul `datetime` erfolgen. Zur Umwandlung der Integer- Werte nach BCD kann folgende Funktion genutzt werden:

```
def int2bcd(d):          # // for integer division; % for modulo
    return (((d // 10) << 4) + (d % 10))
```

b) Die Uhr soll ja nicht dauernd neu gesetzt werden. Erweitere dein Programm, so dass es feststellen kann ob die Uhr läuft oder nicht, und die Uhr nur setzt, wenn sie nicht schon läuft.

Der LED Treiber-Chip HT16K33

Damit unsere Uhr auch ohne PC sichtbar wird, soll jetzt ein 4-stelliges 14 -Segment-Display in Betrieb genommen werden. Die Ansteuerung erfolgt mit dem LED-Treiber ***HT13K33*** von Holtek. Er besitzt einen RAM Speicher mit 8 Byte, in dem die Information, welche LEDs leuchten sollen abgelegt wird. Die Verdrahtung bleibt einfach. C steht für CLK und D für SDA, - für Masse. Das 14-Segment Display hat anders als das 7-Segment-Display im Bild einen zweiten Plus-Pol. Auch dieser ist mit 5V zu verbinden.



Das folgende Programm gibt einen festen Text aus, sowie eine Laufschrift:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# interface_i2c_ht16k33_1.py

# install i2c-tools and smbus with:
# sudo apt-get install i2c-tools python-smbus python3-smbus
# run: sudo raspi-config
# enable I2C (8 Advanced Options, A7 I2C) and reboot
# add user pi to group i2c: sudo adduser pi i2c and reboot
# test with:                i2cdetect -y 0 (for Pi Model B rev. 1)
#                           i2cdetect -y 1 (for Pi Model B rev. 2)
#
# Bit numbers for the Display: 2 Bytes per digit
# HByte: FEDCBA98 LBYTE: 76543210
# A = Bit 10, B = Bit 11, C = Bit 12, D = Bit 13, E = Bit 14 = decimal point
#
#   000000000
#  58   9   A1
#  5 8  9  A 1
#  5  8 9 A  1
#  5   89A   1
#   6666 7777
#  4   BCD   2
#  4  B C D  2
#  4 B  C  D 2
#  4B   C   D2
#   333333333   E
```

```
import smbus
from time import sleep

port = 1  # (0 for rev.1, 1 for rev 2!)
bus = smbus.SMBus(port)
dispAddr = 0x70

d = {'0': 0x0C3F, '1': 0x0006, '2': 0x00DB, '3': 0x008F, '4': 0x00E6, \
     '5': 0x2069, '6': 0x00FD, '7': 0x0007, '8': 0x00FF, '9': 0x00EF, \
     '*': 0x3FC0, '+': 0x12C0, '-': 0x00C0, '_': 0x0008, '.': 0x4000, \
     '/': 0x0C00, "'": 0x0002, '°': 0x00E3, 'A': 0x00F7, 'B': 0x128F, \
     'C': 0x0039, 'D': 0x120F, 'E': 0x00F9, 'F': 0x0071, 'G': 0x00BD, \
     'H': 0x00F6, 'I': 0x1200, 'J': 0x001E, 'K': 0x2470, 'L': 0x0038, \
     'M': 0x0536, 'N': 0x2136, 'O': 0x003F, 'P': 0x00F3, 'Q': 0x203F, \
     'R': 0x20F3, 'S': 0x00ED, 'T': 0x1201, 'U': 0x003E, 'V': 0x0C30, \
     'W': 0x2836, 'X': 0x2D00, 'Y': 0x1500, 'Z': 0x0C09, '[': 0x00FF, \
     ']': 0x00EF, '|': 0x2D00, '=': 0x12C0, '@': 0x00FD, ' ': 0x0000}


def dispInit():
    bus.write_byte(dispAddr, 0xE7)  # dimming 0-F (LowNibble)
    bus.write_byte(dispAddr, 0x21)  # turn system osc. on/off (Bit 0)
    bus.write_byte(dispAddr, 0x81)  # diplay on/off (Bit 0) blink (Bit2 + Bit1)


def dispWrite(dispString):
    bus.write_word_data(dispAddr, 0, d[dispString[0]])
    bus.write_word_data(dispAddr, 2, d[dispString[1]])
    bus.write_word_data(dispAddr, 4, d[dispString[2]])
    bus.write_word_data(dispAddr, 6, d[dispString[3]])


def dispTicker(tString, ttime):
    for i in range(len(tString) - 4 + 1):
        newString = tString[i] + tString[i + 1] + tString[i + 2] + \
            tString[i + 3]
        dispWrite(newString)
        sleep(ttime)


dispInit()
dispWrite('*GO*')
sleep(2)

while True:
    dispTicker('T3EC LOVES PYTHON    ', 0.5)
```

Das Programm arbeitet mit einem Python-Wörterbuch (***dictionary***). Das Dictionary ist eine Folge von Wertpaaren in geschweiften Klammern. Die Wertepaare sind durch Kommata getrennt; zwischen den Werten befindet sich ein Doppelpunkt. Der erste Wert ist der Schlüssel (***key***) mit dem man Zugriff auf den zweiten Wert, der auch mit Wert bezeichnet wird (***value***) hat. Ein typisches Anwendungsbeispiel ist ein reales Wörterbuch, z.B. Deutsch-Französisch:

```
wb = {'angewandt':'appliqué','Informatik':'informatique','Schule':'école'}
```

Der Zugriff mit dem Schlüssel erfolgt mit eckigen Klammern. Der Befehl

```
print(wb['angewandt'])
```

druckt das Wort "appliqué".

Wir nutzen das Dictionary hier um die Buchstaben zu kodieren. Für das 14-Segment Display werden pro Stelle (***digit***) 2 Byte benötigt. 14 Bit dienen der Ansteuerung der Segment-LEDS und das 15. Bit (Bit14) zur Ansteuerung des Dezimalpunktes. Um eine Eins

darzustellen müssen nur die Segmente 1 (Bit1) und 2 (Bit 2) eingeschaltet werden. Dies ergibt für das HByte `0x00` und für das LByte `0b00000110 = 0x06`, also findet man im Dictionary das Wertepaar `'1':0x0006`.

In der Funktion `dispInit()` werden einige Register des Chip HT16K33 initialisiert, damit dessen Oszillator schwingt und das Display eingeschaltet ist. Dazu wird der smbus Befehl

```
bus.write_byte(dispAddr,0xE7)
```

verwendet, der nur ein Byte vom Master zum Slave schickt. Der Chip HT16K33 erkennt an den oberen 4 Bit, welches Register adressiert werden soll. Im einzelnen handelt es sich um folgende Register: ***Dimming Setup Register:*** Kommando (HNibble) = 0xE Mit den unteren 4 Bit kann eine Pulsweite in 16er-Schritten eingestellt werden. LByte = 0x0: minimale Helligkeit. LByte = 0xF: maximale Helligkeit ***System Setup Register:*** Kommando (HNibble) = 0x2 Bit 0 schaltet Systemtakt ein (default 20H: system oscillator off) ***Display Setup Register:*** Kommando (HNibble) = 0x8 Bit 0 schaltet das Diplay ein.(default 80H: no blink Display off) Bit 2 und Bit 1 legen die Blinkfrequenz fest. (00 no blink, 01: 2Hz, 10: 1Hz, 11: 0,5Hz) Weitere Informationen kann man dem Datenblatt entnehmen (z.B. S 30): http://www.adafruit.com/datasheets/ht16K33v110.pdf

Die Funktion `dispWrite(dispString)` schreibt die Daten (darzustellende Zeichen) ins RAM des Chip. Dies passiert mit dem Kommando

```
bus.write_word_data(dispAddr,0,d[dispString[0]])
```

welches ein Wort (2 Byte) zum Slave schickt. Der 2 Parameter übergibt hier wie beim RTC den Adresszeiger fürs RAM. Er muss also jeweils um 2 erhöht werden. Mit Hilfe des Dictionary wird das richtige Word für jedes der 4 Zeichen des Strings ermittelt.

Die Funktion `dispTicker(tString,ttime)` erzeugt eine Laufschrift indem sie über die gesamte Zeichenkette iteriert und immer die vier Zeichen nächsten Zeichen ausgibt. Mit ttime kann die Geschwindigkeit der Laufschrift eingestellt werden.

Aufgabe I11:

Ändere das obige Programm, so dass das Datum, die Uhrzeit und der Wochentag des RTC in Laufschrift ausgegeben werden.

Aufgabe I12:

Schließe den Temperatursensor (DS18B20) wieder an und gib zusätzlich die aktuelle Temperatur aus.

Aufgabe I13: (für Fleißige)

Der 8-Bit I/O Port-Expander-Chip PCF8574 ermöglicht es den Raspi um 8 digitale Ein- bzw. Ausgänge zu erweitern. Es handelt sich um einen Seriell/Parallel- Wandler. Ein über den I²C-Bus gesendetes Byte wird parallel an 8 Pins ausgegeben. Verbinde den Chip mit dem Raspi. Schließe zwei LEDs (Vorwiderstände!) und zwei Taster an. Die

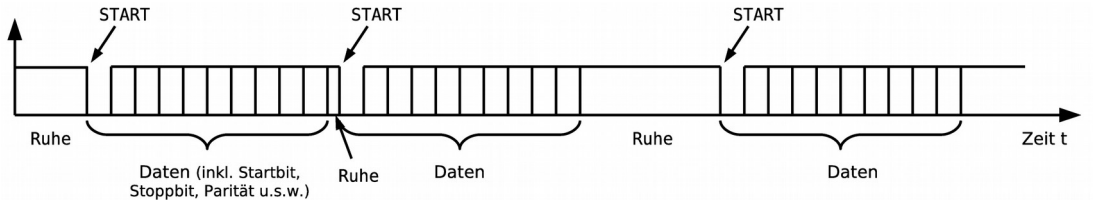
Taster sollen die LEDs abwechselnd ein und ausschalten. Weitere Informationen zum PCF8574: http://www.nxp.com/documents/data_sheet/PCF8574.pdf

Die asynchrone serielle Schnittstelle EIA232

Die EIA-232-Schnittstelle ist seit fast 50 Jahren standardisiert.Ebenso häufig wie die aktuelle Bezeichnung EIA-232 (**EIA** für ***Electronic Industries Alliance***) findet man die alte Bezeichnung RS-232 (*RS für Radio Sector bzw. Recommended Standard*) oder die Bezeichnung V24. EIA-232 definiert die Verbindung zwischen einem Terminal (DTE) und einem Modem (DCE), was Timing, Spannungspegel, Protokoll und Stecker betrifft. Auch wenn diese Schnittstelle schon viele Jahre besteht, reicht ihre Geschwindigkeit für übliche Anwendungen oft aus. Sie ist äußerst robust und erlaubt auch größere Kabellängen. USB-EIA-232-Wandler ermöglichen die Verbindung mit dem PC falls keine serielle EIA-232-Schnittstelle mehr verfügbar ist.

Das asynchrone Verfahren

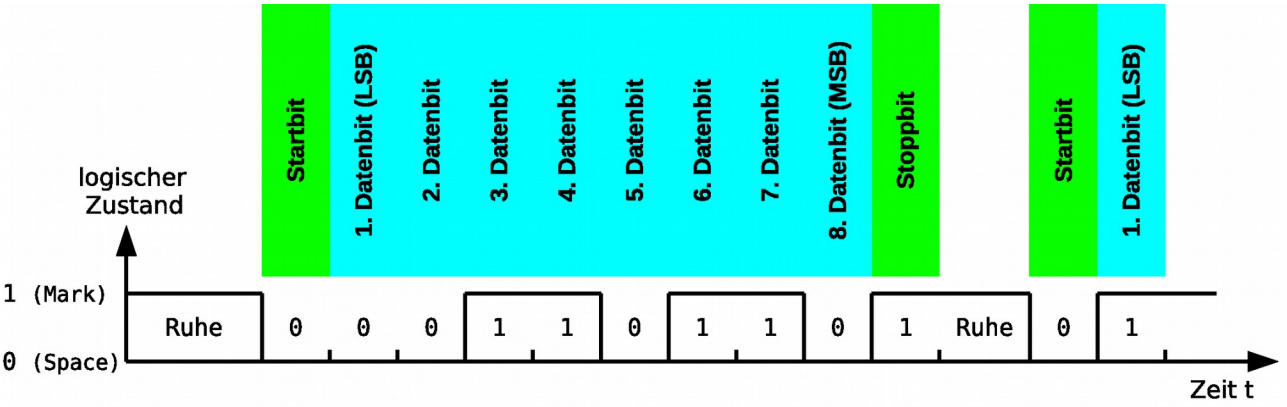
Bei der asynchronen Datenübertragung kann die Informationsübertragung zeichenweise zu einem beliebigen Zeitpunkt erfolgen. Ein an einem PC arbeitender Benutzer sendet z.B. diese Zeichen in zufälligen unvorhersehbaren Intervallen. Sender (Tastatur) und Empfänger (PC) sind daher nur während der Übermittlung eines einzelnen Zeichens synchronisiert. Die Synchronisation ist durch eine festgelegte Bitrate, ein festgelegtes Datenformat sowie die Verwendung von Start- und Stoppbits möglich (keine Taktleitung).



Der Zeichenrahmen (SDU, Serial Data Unit)

Jedes einzelne Zeichen wird innerhalb eines Zeichenrahmens (*frame*, **SDU**, *Serial Data Unit*) zwischen Steuerbits eingefasst. Im inaktiven Zustand (Ruhezustand, es wird kein Zeichen übertragen) wird die Übertragungsleitung auf logisch 1 (Mark) gehalten. Der Beginn der Datenübertragung und damit auch die Synchronisation erfolgt mit Hilfe eines Startbits (logisch 0, Space), das an den Anfang eines jeden Zeichens gesetzt wird. Anschließend werden die Datenbits ausgesendet. Je nach gewähltem Code können dies 5, 6, 7 oder 8 Bits sein. Am häufigsten ist die Übertragung mit 8 Bit. Man beachte, dass das niederwertigste Datenbit (**LSB**, *Least Significant Bit*, D0) zuerst übertragen wird! Nach den Daten wird ein Paritätsbit und ein, anderthalb oder zwei Stoppbits (logisch 1) übertragen. Das Paritätsbit ist ein Kontrollbit, dient der Fehlererkennung, und bezieht sich nur auf die Datenbits. Heute wird es meist weggelassen. **Heutige Software arbeitet meist mit 8**

Datenbit, ohne Parität und einem Stoppbit. Kurzschreibweise: 8N1 Beispiel für die Übertragung des Buchstaben `'l'`:

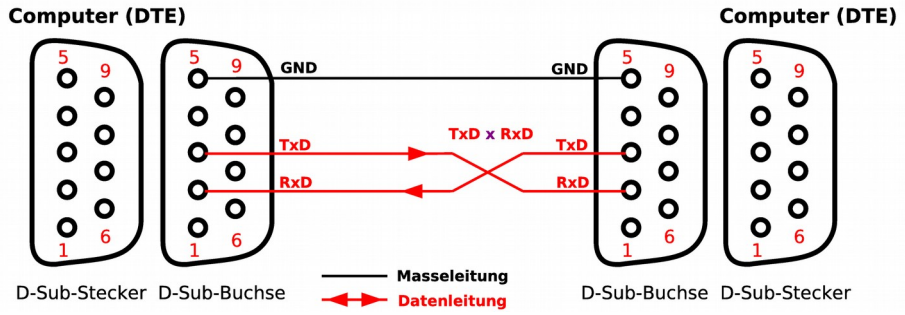


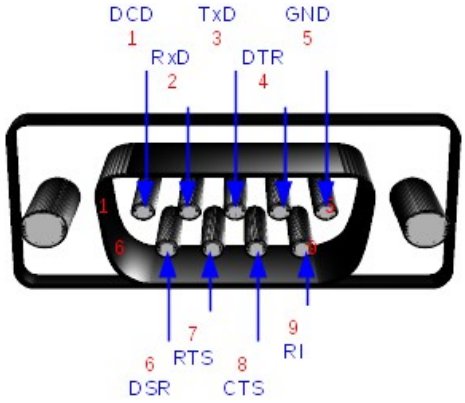
Die Übertragungsgeschwindigkeit der EIA-232 Schnittstelle liegt zwischen 300 bit/s und 115200 bit/s. Bei jeder Signaländerung wird nur ein Bit übertragen. Die Übertragungsgeschwindigkeit ist somit der Baudrate (Signaländerung/Sekunde) bei EIA-232 gleichzusetzen. Heute werden durchweg höhere Bitraten als früher eingesetzt. Häufige Bitraten sind 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600 und 115200 bit/s (bzw. Baud (Bd)). Höhere Bitraten verringern die maximale Kabellänge (siehe: Die Reichweite von EIA-232)! Mit Standardkabeln ist bei einer Baudrate von 19200 Baud eine Reichweite von um die 15 m möglich1. Mit Kabeln, welche eine besonders niedriger Kapazität aufweisen (z.B. nicht geschirmtes Netzwerkkabel UTP CAT-5), lassen auch 45 m erreichen.

Empfänger und Sender müssen auf die gleiche Geschwindigkeit (Bit- bzw. Baudrate) und das gleiche Datenformat eingestellt werden (unter Datenformat versteht man die Zahl der Datenbits und Stoppbits sowie die Parität, Bsp.: 8N1).

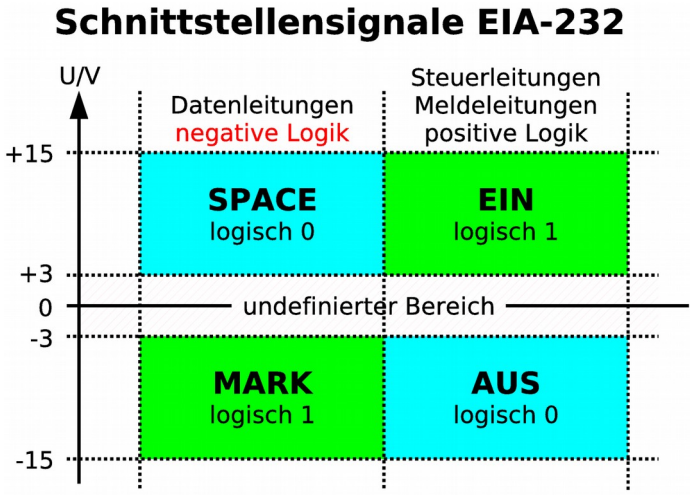
Schnittstellensignale und Hardware

Bei der seriellen Schnittstelle können bis zu neun Daten, Steuer und Meldeleitungen verwendet werden. Es sind dabei viele unterschiedliche Kombinationen möglich. Bevor man zwei Geräte verbindet sollte man sich also genaustes Informieren wie die Schnittstelle eingesetzt wird. Die allerwichtigsten Schnittstellensignale sind die beiden Datenleitungen "***Transmitter Data***" (**TxD**) und "***Receiver Data***" (**RxD**) sowie die Masseleitung (GND). Mit diesen drei Leitungen ist eine bidirektionale Datenübertragung möglich. Heute werden glücklicherweise meist einfache **Null-Modem**-Verbindungen mit nur diesen drei Leitungen eingesetzt. Null-Modem bedeutet, dass die Leitungen gekreuzt werden müssen, so dass die Sendeleitung mit der gegenüberliegenden Empfangsleitung verbunden ist.





Um akzeptable Reichweiten zu erhalten reicht die Spannungsdifferenz zwischen 0 un 5V bzw. 3,3V nicht aus. Auch ist Masse als logischer Low-Pegel nicht günstig, da hier ein Leitungsbruch nicht erkannt werden kann. EIA-232 arbeitet daher mit Spannungen zwischen 3V und 15V bzw. -3V un -15V. **!!!!Niemals den RASPI sofort mit der seriellen Schnittstelle eines PC verbinden!!!!!!** PCs arbeiten üblicherweise mit ±12 V, Notebooks mit ±7-8 V. Die Datenleitungen arbeiten mit negativer Logik! +12V entspricht also logisch *Low*, -12V entspricht logisch *High*.



Eine digitale Schaltung mit TTL-Pegeln darf nicht ohne Pegelwandler an die serielle Schnittstelle angeschlossen werden, da sie sonst zerstört wird!

Will man mit TTL-Bausteinen die serielle Schnittstelle benutzen, so muss eine TTL/EIA-232-Pegelanpassung vorgenommen werden. Die Halbleiterindustrie bietet solche Pegelwandler als integrierte Schaltkreise an. Sie werden als "232-ICs" bezeichnet. Je nach Hersteller, sind verschiedene Buchstaben voran gesetzt (z.B. MAX232). Dies gilt natürlich nicht, wann man zum Beispiel den Raspi mit einem Mikrocontroller mit gleicher Versorgungsspannung (3,3V) verbindet.

Weitere Informationen zur seriellen Schnittstelle:
http://weigu.lu/a/pdf/MICEL_B5_Serielle_Schnittstelle.pdf

EIA-232 mit dem Raspberry Pi

Im Normalfall nutzt der Raspberry Pi die serielle Schnittstelle um mit der Außenwelt zu kommunizieren. Auch ohne Ethernet-Verbindung kann man sich so einloggen und Terminalbefehle ausführen. Die Verbindung zum Raspi erfolgt über ein 3,3V-USB-EIA232-Kabel von adafruit (http://www.adafruit.com/product/954). Im Kabel ist ein Pegelwandler eingebaut (Profilic oder FTDI). Der Treiber des Betriebssystems stellt eine virtuelle serielle Schnittstelle zur Verfügung. Der **weiße Draht** ist mit der Sendeleitung **TxD** zu verbinden, der **grüne Draht** mit der Empfangsleitung **RxD** und der **schwarze Draht** mit **GND** (Masse). **Der rote Draht (5V vom PC) darf nicht angeschlossen werden!!** (Außer man möchte den Raspi über diese Verbindung mit Spannung versorgen (USB2: max 500mA); dann ist aber kein USB-Netzteil anzuschließen.) Beim Kauf des USB-EIA232-Kabels muss darauf geachtet werden, dass dieser auf der Sende- und Empfangsleitung einen maximalen Pegel von 3,3V liefert!

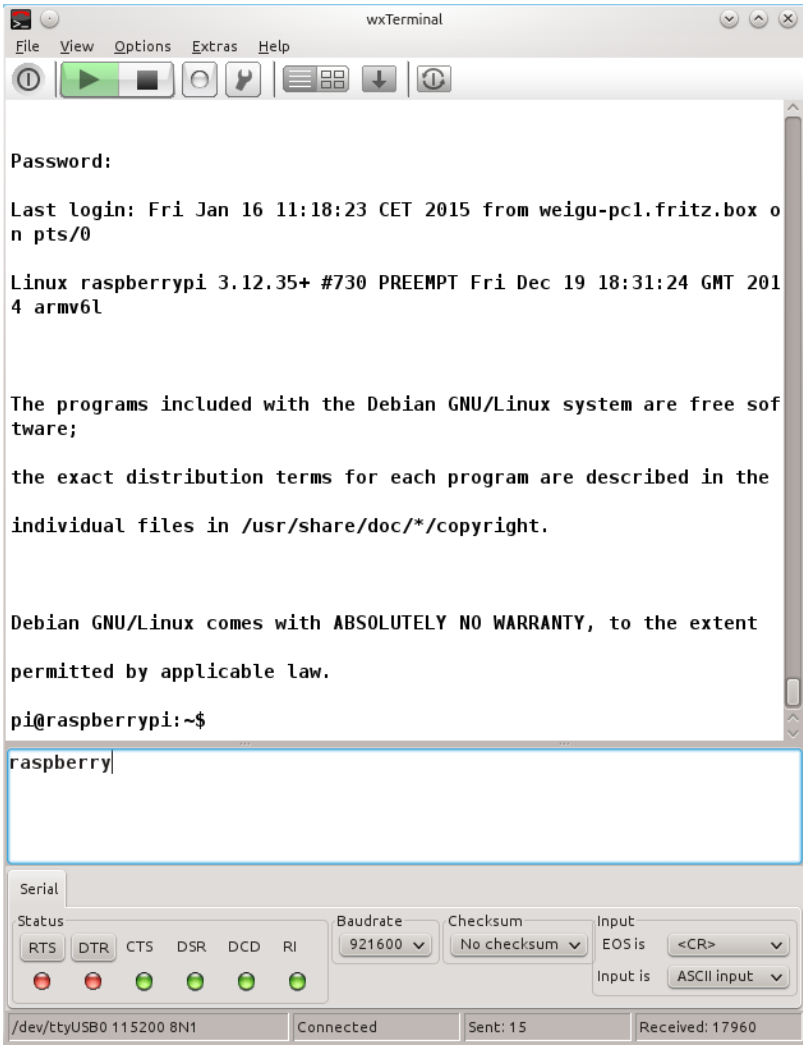
!!! Achte beim Verdrahten darauf keine Kurzschlüsse zu verursachen. Die GPIO- Pins des Raspi sind nicht so robust wie die verschiedener Mikrocontroller!
Wird der rpibuffer-Adapter (weigu.lu/c/rpibuffer) genutzt, so ist der Jumper auf 3.3V zu setzen.

Auf dem PC wird eine Terminalsoftware benötigt. Ein quelloffenes freies Terminalprogramm findet man bei IFTOOLS https://iftools.com/download/index.en.php. Das Terminalprogramm `wxterm` läuft auf Linux und Windows und bietet alles was man von einem Terminalprogramm erwartet. Es läßt sich auch leicht auf einem USB-Stick installieren (einzelne `.exe`-Datei). Über einen Klick auf den Schraubenschlüssel kommt man zu den "*`RS232 Connection Settings`*" wo man die Schnittstelle auswählen kann und die Baudrate und mit 115200 bit/s einstellen muss (Datenformat 8N1, kein Handshake (beide Auswahlfelder deaktiviert)). Mit einem Klick auf den Pfeil wird die Verbindung hergestellt. Oben is das Empfangsfenster, unten das Sendefenster.

Aufgabe I14:

Verbinde den Raspi mit der seriellen Schnittstelle. Installiere das Terminalprogramm und stelle eine Verbindung mit dem Raspi her. Boote den Raspi neu und beobachte

den Bootvorgang. Logge dich mit Hilfe des Terminalprogramms ein. Teste die Befehle `ls -l` und `sudo reboot`.



Das Einloggen mit Hilfe eines Terminalprogramm funktioniert zwar, ist aber nicht sehr komfortabel. Wir wollen die Schnittstelle für andere Zwecke verwenden, und müssen deshalb die Terminalfunktion abstellen. Das geht sehr einfach mit dem Befehl:

`sudo raspi-config`

Im achten Menupunkt "*`8 Advanced Options`*" wählen wir Punkt "*`A8 Serial Enable/Disable shell and kernel messages on the serial connection`*" und schalten die Option mit *`No`* ab. Dann verlassen wir das Programm mit *`Finish`*. Der Raspi führt einen Neustart durch.

Um die serielle Schnittstelle mit Python zu nutzen benötigen wir das Python Modul `python-serial`:

`sudo apt-get install python-serial`

falls es noch nicht installiert ist. Jetzt lässt sich die serielle Schnittstelle für unsere Zwecke nutzen. Hier ein erstes Testprogramm:

```
#!/usr/bin/env python3
# interface_eia232_p2_1.py (python 2!)
#
# if not installed, install python-serial: sudo apt-get install python-serial
# run: sudo raspi-config
# disable shell and kernel messages on serial connection (8 Advanced Options,
# A8 serial)      run: sudo reboot
# Wiring: Raspi TxD: white Raspi RxD: green GND: black

import serial

ser = serial.Serial("/dev/ttyAMA0", baudrate=115200, timeout=1.0)

while True:
    ser.write("Please type something:\n")
    mytext = ser.read(30)
    while mytext == '':
        mytext = ser.read(30)
    ser.write("You sent: " + mytext)
```

Zuerst wird ein serielles Objekt mit dem (beliebigen) Namen `ser` erstellt. Hierbei müssen als Parameter die serielle Schnittstelle (beim Raspi `/dev/ttyAMA0`), die Baudrate und das Timeout angegeben werden. Das Timeout gibt an wie viele Sekunden beim Einlesen gewartet wird ob Daten vorhanden sind. Nach dieser Zeit wird der Lese-Vorgang abgebrochen. Dann kann mit der Schreib-Methode (`write()`) Text über die serielle Schnittstelle ausgegeben werden. Bei der Lese-Methode (`read()`) ist anzugeben wie viele Zeichen maximal eingelesen werden sollen. In einer Schleife wird (im Sekundenrythmus: siehe Timeout) darauf gewartet, dass Text eingegeben wurde, bevor dieser dann ausgegeben wird.

In Python 2 wurde der `str`-Typ für Text und für binäre Daten verwendet. Da dies zu Problemen führte (besonders bei der Verwendung von Unicode) wurde in Python 3 wie auch in anderen Sprachen 2 unterschiedliche Typen für Text (`string`) und binäre Daten (`bytes`) geschaffen. Arbeitet man nur mit Text oder nur mit binären Daten, so stellt dies kein Problem dar.

Bei der seriellen Schnittstelle jedoch möchte man Text als binäre Daten versenden. Es ist also nötig der Text in einen Binärstring überzuführen und umgekehrt. Dazu nutzt man die beiden Methoden `encode()` und `decode()`. Als Parameter kann die Kodierung angegeben werden (`utf-8`, `utf-16`, `latin-1`). Ohne Angabe wird die default-Codierung des Laufzeit-Systems verwendet. Beim Raspi kann man diese mit dem Befehl

`locale charmap`

ermitteln. Hier das entsprechende Programm für Python 3:

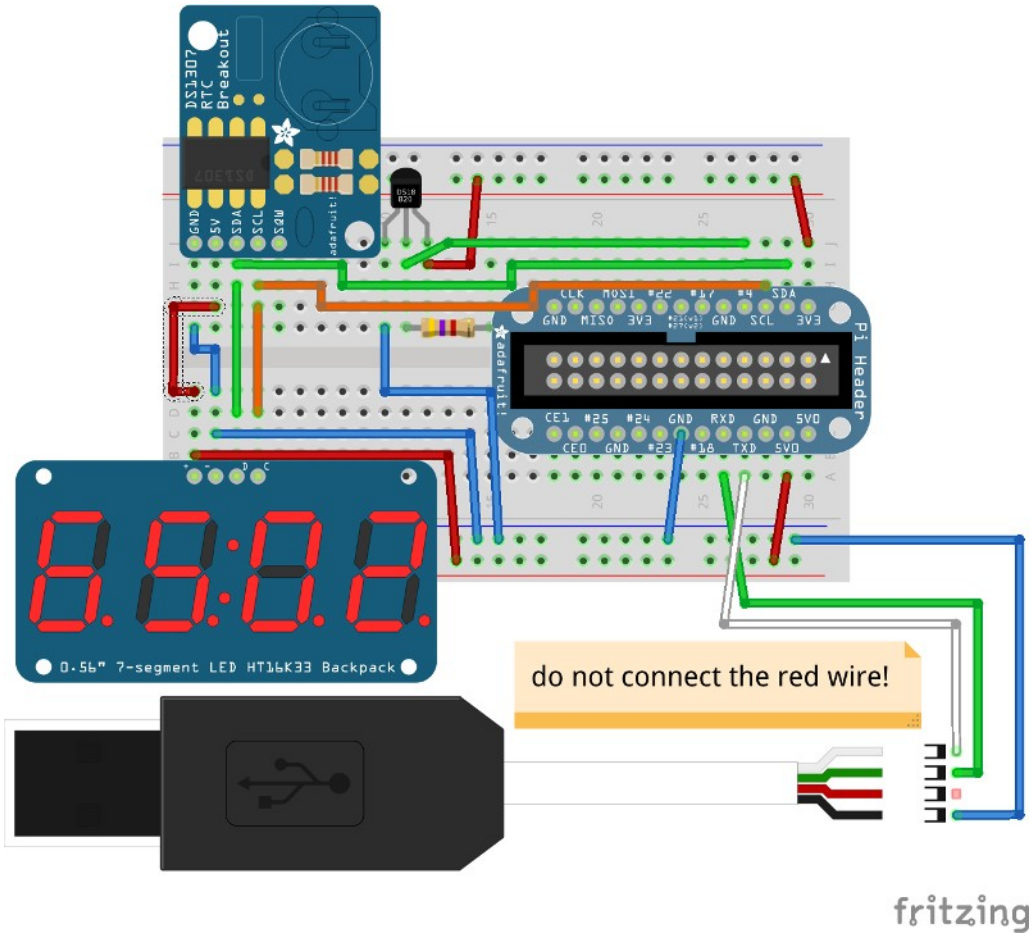
```
#!/usr/bin/env python3
# interface_eia232_p3_1.py (python3!)
#
# if not installed, install python-serial: sudo apt-get install python-serial
# run: sudo raspi-config
# disable shell and kernel messages on serial connection (8 Advanced Options,
# A8 serial)      run: sudo reboot
# Wiring: Raspi TxD: white Raspi RxD: green GND: black

import serial

ser = serial.Serial("/dev/ttyAMA0", baudrate=115200, timeout=1.0)

while True:
```

```
    ser.write("Please type something:\n".encode())
    mytext = ser.read(30)
    while mytext.decode() == '':
        mytext = ser.read(30)
    ser.write("You typed: ".encode() + mytext)
```

Das obige Programm läuft in einer Endlosschleife. Es ist also nicht möglich die Datei `ser` wieder sauber zu schliessen (`ser.close()`). Glücklicherweise wird die Datei automatisch geschlossen, wenn das Programm mit `Ctrl+c` abgebrochen wird. Das gleiche gilt auch wenn ein Gui-Programm geschlossen wird (siehe Aufgabe weiter unten).

Aufgabe I15:

Erweitere die Aufgabe I12, so dass Datum, Uhrzeit, Wochentag und Temperatur auch über die serielle Schnittstelle versendet werden.



Aufgabe I16:

Der Raspi soll den Text "Please type something:" zum PC senden. Der vom PC zurückgesendete Text soll dann auf dem Display (Laufschrift) dargestellt werden (Achtung! keine kleinen Buchstaben oder Sonderzeichen verwenden, die nicht auf dem Display dargestellt werden können).

Aufgabe I17:

Verbinde zwei Raspis über die serielle Schnittstelle miteinander (Sendeleitung an Empfangsleitung und umgekehrt). An beiden Raspis sollen auch 2 Schalter und 2 LEDs angeschlossen werden. Die beiden Schalter sollen jetzt die beiden LEDs des anderen Raspi schalten. Die Informationen wann welche LED ein ist soll über die serielle Schnittstelle versendet werden.

Aufgabe I18:

Die gleiche Aufgabe nur dass statt des Raspi ein Mikrocontroller verwendet wird (http://www.weigu.lu/a Kapitel serielle Schnittstelle).

Aufgabe I19: (für Freaks)

Statt des Terminalprogramms soll ein eigenes Python-Programm mit Tkinter programmiert werden, mit dem Daten von der seriellen Schnittstelle empfangen und gesendet werden können. Das Senden stellt kein Problem dar. Der Empfang muss aber die Mainloop von Tkinter unterbrechen können. Dies ist möglich mit der Methode `after()`. Die Parameter sind eine Zeit in Millisekunden und die aufzurufende Funktion. Sie wird im Hauptprogramm aufgerufen.

```
mainWin.after(10,receive)
mainWin.mainloop()
```

Die Empfangsfunktion kann dann folgendermaßen aussehen:

```
def receive():
   if (ser.inWaiting() != 0):
       sertextrec.set(ser.read(30))
   mainWin.after(10,receive)  # reschedule event in 10 ms
```

Weitere Infos: http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/universal.html
Eine mögliche Lösung findet man unter: http://www.weigu.lu/c/python/download