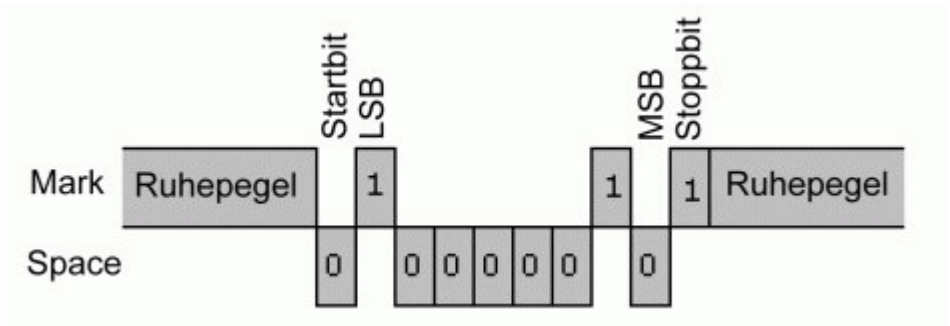
Raspberry Pi: Serielle Schnittstelle

Allgemeines

Bei einem seriellen, asynchronen Datentransfer werden die einzelnen Bits eines Datenbytes nacheinander über eine Leitung übertragen (siehe Bild). Der Ruhezustand der Übertragungsleitung, der auch mit "Mark" bezeichnet wird, entspricht dem Pegel einer logischen 1. Die zur Übertragung verwendeten Spannungs- bzw. Strompegel können Sie der Beschreibung der einzelnen Schnittstellen entnehmen. Die Übertragung eines Bytes beginnt mit einem vorangestellten Startbit, das als logische 0 ("SPACE") gesendet wird. Anschließend werden nacheinander - je nach eingestelltem Format - fünf bis acht Datenbits, beginnend mit dem niederwertigen Bit (least significant bit, LSB), ausgegeben. Dem letzten Datenbit kann ein Paritätsbit folgen, das zur Erkennung von Übertragungsfehlern dient. Das Paritätsbit ergänzt das Datenbyte auf eine gerade (gerade Parität, even parity) oder ungerade (ungerade Parität, odd parity) Anzahl von 1-Bits. Das Ende des Zeichens wird durch ein oder zwei Stoppbits gebildet. Alle Bits werden sequenziell gesendet.



Ein Byte besteht dann aus einer Folge von acht Datenbits, die von Start- und Stoppbit eingerahmt werden. Zwischen zwei aufeinanderfolgenden Bytes können sich auch beliebig lange Pausen befinden, da der Beginn eines Zeichens am Startbit eindeutig erkannt wird. Daher nennt man diese Form der Übertragung "asynchron". Durch die asynchrone Übertragung wird die Übertragungsrate gesenkt, da für z. B. 8 Informationsbits 10 Bits über die Leitung gesendet werden. Nach dem Stoppbit kann sofort wieder eine neue Übertragung mit einem Startbit beginnen.

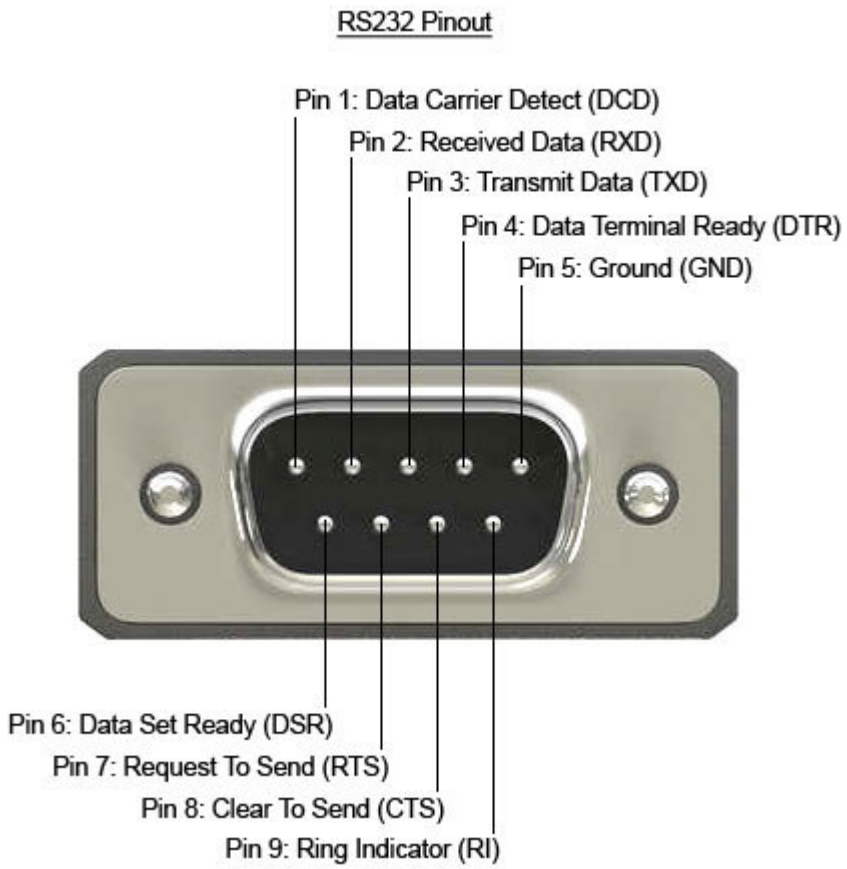
Die Datenrate wird in *Bit pro Sekunde (bps)* bzw. *Baud* (nach dem französischen Ingenieur und Erfinder Jean Maurice Émile Baudot) angegeben. Dabei werden alle Bits (auch Start- und Stoppbit) gezählt und Lücken zwischen den Bytetransfers ignoriert. Deshalb ist die Baudrate der reziproke Wert der Länge eines Bits. Als Datenraten sind folgende Werte üblich:

```
150, 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600 und 115200
```

Da die Pause zwischen zwei aufeinanderfolgenden Datenbytes beliebig lang sein darf, spricht man von einer "asynchronen" Kommunikation. Für den Datenverkehr synchronisieren sich Sender und Empfänger bei der asynchronen Übertragung für jedes einzelne Zeichen neu. Vor jedem Zeichentransfer liegt auf der Übertragungsleitung das Signal auf 1-Pegel. Soll nun ein Zeichen übertragen werden, so wird dies dem Empfänger vom Sender durch ein Startbit angezeigt, indem für einen Taktzyklus das Signal auf 0 gelegt wird. Anhand der 0-1-Flanke kann der Empfänger den Beginn eines Datenbytes exakt erkennen. Damit sind

Sender und Empfänger für dieses Zeichen synchronisiert. Anhand der Stoppbits erkennt der Empfänger das Ende des Zeichens, damit dient das Stoppbit ebenfalls der Synchronisation. Sender und Empfänger müssen sich zuvor auf die Anzahl der Stoppbits, der Datenbits, der Berechnung der Paritätsbits und auf die Frequenz des Übertragungstaktes (Baudrate) verständigen. Diese Parameter werden zumeist einmal in den Schnittstellen einprogrammiert und bleiben für die gesamte Dauer der Kommunikation unverändert.

Für das Verständnis der seriellen Schnittstelle ist es wichtig zu wissen, dass es je eine Datenleitung für das Senden und Empfangen der Daten (TD/RD), zwei Paare von Handshake-Leitungen RTS und CTS bzw. DTR und DSR, zwei Statusleitungen CD und RI sowie GND (Massepotenzial) gibt. Vom Computer aus gesehen sind die Leitungen TD, RTS, DTR Ausgänge, die Leitungen RD, CTS, DSR, CD, RI Eingänge. RTS und DTR können also programmgesteuert aktiviert und deaktiviert werden, CTS, DSR, CD und RI können nur gelesen werden. Die folgende Grafik zeigt die übliche Belegung der Schnittstelle beim 9-poligen RS-232-Stecker.



Die serielle Schnittstelle des Raspberry Pi ist über GPIO14 (TXD) und GPIO15 (RXD) erreichbar. Günstigerweise liegen die entspechenden Pins auf der Steckerleiste P1 untereinander (6 - GND, 8 - TXD, 10 - RXD), sodass Sie auch mit einer dreipoligen Pfostenbuchse abgegriffen werden können (derartige Buchsen findet man auch of in PC-Bastelkiste). Aber immer daran denken: *Auch diese Schnittstelle arbeitet mit nur 3,3 Volt.*

Beim Raspberry Pi sind keine speziellen Leitungen für den Hardware-Handshake vorgesehen. Falls ein hardwarebasierter Handshake benötigt wird, muss man auf freie GPIO-Pins zurückgreifen.

Serielle Schnittstelle freischalten

Per Voreinstellung ist auf der Schnittstelle, die unter Linux auf `/dev/ttyAMA0` angesprochen wird, eine serielle Login-Konsole konfiguriert, auf der auch der gesamte Bootvorgang protokolliert wird. Deshalb kann man diese Schnittstelle nicht so ohne Weiteres für andere Zwecke verwenden. Das bedeutet, dass Sie zuerst die serielle Konsole abschalten müssen. Maßgeblich dafür sind zwei Dateien:

- `/boot/cmdline.txt`
- `/etc/inittab`

Bearbeiten `/boot/cmdline.txt`

Die Datei `/boot/cmdline.txt` regelt den Bootvorgang des Pi. Dort werden diverse Boot-Optionen eingestellt, so auch die serielle Konsole. Per Default steht relativ wenig in der Datei, wobei es bei kommenden Versionen schon wieder anders aussehen kann (*für die Darstellung hier wurde die Zeile umbrochen!*):

```
dwc_otg.lpm_enable=0 console=ttyAMA0,115200 kgdboc=ttyAMA0,115200
 console=tty1 root=/dev/mmcblk0p2 rootfstype=ext4 elevator=deadline rootwait
```

Aus diesen Optionen müssen Sie nun die Angaben zur Konsole "ttyAMA0" löschen, aber alles andere unbedingt unverändert lassen. Am Besten machen Sie zuerst ein Backup der Datei (`cp /boot/cmdline.txt /boot/cmdline.bak`). Dann ändern Sie in der Originaldatei die Zeile:

```
dwc_otg.lpm_enable=0 console=ttyAMA0,115200 kgdboc=ttyAMA0,115200
 console=tty1 root=/dev/mmcblk0p2 rootfstype=ext4 elevator=deadline rootwait
```

in die folgende Zeile (oben sind die zu löschenden Teile farbig hervorgehoben):

```
dwc_otg.lpm_enable=0
 console=tty1 root=/dev/mmcblk0p2 rootfstype=ext4 elevator=deadline rootwait
```

Bearbeiten `/etc/inittab`

Nun wird die Datei `/etc/inittab` bearbeitet. In ihr ist die serielle Schnittstelle als Login-Schnittstelle definiert. Dazu Öffnen Sie die Datei und bearbeiten den folgenden Eintrag:

```
#Spawn a getty on Raspberry Pi serial line
T0:23:respawn:/sbin/getty -L ttyAMA0 115200 vt100
```

Ändern Sie die Zeile durch ein davor gestelltes Kommentarzeichen in

```
#Spawn a getty on Raspberry Pi serial line
# T0:23:respawn:/sbin/getty -L ttyAMA0 115200 vt100
```

Damit ist die Login-Funktion abgeschaltet. Wenn Sie den Pi nun neu starten, können Sie die serielle Schnittstelle beliebig nutzen.

Damit der User pi auch auf die Schnittstelle zugreifen darf, muss er zusätzlich in die Gruppe "dialout" aufgenommen werden. Das erreichet man mit dem Kommando

```
sudo usermod -a -G dialout pi
```

Bei neueren Versionen des Raspian-Betriebssystems kann die Aktivierung/Deaktivierung der Schnittstelle auch über `raspi-config` erfolgen. Damit läßt sich die Schnittstelle aber nur komplett aktivieren oder deaktivieren (nach dem Deaktivieren ist `/dev/ttyAMA0` dann nicht mehr vorhanden und steht auch fürdie Programmierung nicht mehr zur Verfügung.

Fehlt also bei Ihrem System die Datei `/etc/inittab`, kann die Deaktivierung des Login-Terminals auf der seriellen Schnittstelle über `systemctl` mit den beiden folgenden Befehlen vorgenommen werden. Die Schnittstelle bleibt dann für die Programmierung aktiv:

```
sudo systemctl stop serial-getty@ttyAMA0.service
sudo systemctl disable serial-getty@ttyAMA0.service
```

Beim Raspberry Pi, Model 3, existieren zwei serielle Schnittstellen (siehe folgenden Abschnitt), weshalb bei diesem Board die Befehle geändert werden müssen:

```
sudo systemctl stop serial-getty@ttyS0.service
sudo systemctl disable serial-getty@ttyS0.service
```

Mit der aktuellen Raspbian-Version müssen Sie sich nicht mehr mit den beiden Dateien herumschlagen. Im Konfigurationstool `raspi-config` gibt es einen Menüpunkt zum Ein- und Ausschalten der seriellen Konsole. Sie rufen `raspi-config` auf und wählen dann die Menüoptionen "Advanced Options" → "Serial" → "Nein" (bzw. "No" in der englischen Sprachvariante) zum Deaktivieren der seriellen Konsole.

Änderungen beim Raspberry Pi Modell 3

Das BCM2837 auf dem Raspberry Pi3 hat 2 UARTs (wie auch seine Vorgänger), jedoch zur Unterstützung Die Bluetooth-4.2-Funktionalität des voll ausgestatteten PL011-UART wurde von den Header-Pins auf den Bluetooth-Chip verschoben und der Mini-UART auf den Header-Pins 8 und 10 verfügbar gemacht. Der Mini-UART ist eine kleine Variante des ursprünglichen UART, die neben einem geringen Durchsatz auch nicht mehr so stabil ist wie zuvor. Die Baud-Rate des Mini-UART wird vom System-Takt (Videocore IV) abgeleitet. Das bedeutet, dass die Baud-Rate nicht mehr stabil bleibt, sondern je nach CPU-Auslastung stark schwankt. Deshalb schlagen die Entwickler vor, den Core-Takt fest auf 250 MHz einzustellen (siehe unten). Insgesamt hat dies eine Reihe von Konsequenzen für Benutzer der seriellen Schnittstelle. Zum einen muss beim Abschalten der seriellen Konsole `ttyS0` anstelle von `ttyAMA0` angegeben werden (siehe oben) und zum anderen sind weitere Anpssungen notwendig.

Das Device `/dev/ttyAMA0`, das zuvor für den Zugriff auf den UART verwendet wurde, stellt jetzt eine Verbindung mit Bluetooth her. Der miniUART ist jetzt auf `/dev/ttyS0` verfügbar. In der neuesten Betriebssystem-Software (ab "Jessie") gibt es ein Symlink `/dev/serial0`, das beim RasPi 3 auf `/dev/ttyS0` verweist und bei den anderen Modellen auf `/dev/ttyAMA0`. Beim RasPi 3 gibt es dann noch `/dev/serial1` als Verweis auf `/dev/ttyAMA0`. So kann man sich beim Programmieren mittels `/dev/serial0` auf die serielle Schnittstelle am GPIO beziehen und das Programm läuft auf allen Modellen ohne Anpassung.

Der Mini Uart hat folgende Eigenschaften:

- 7 oder 8 Bit Daten
- 1 Startbit und 1 Stoppbit
- Keine Parität
- Break-Generierung
- 8 Bytes tiefe FIFOs für Empfang und Senden
- Registersatz wie beim IC 16550
- Baudrate, abgeleitet vom System Clock

Durch das Ändern der Datei `/boot/config.txt` kann die Abhängigkeit vom System Clock durch Hinzufügen der folgende Zeile am Ende der Datei fest eingestellt werden:

```
core_freq=250
```

Dies behebt das Timing-Problem und scheint wenig Einfluss auf andere Dinge zu haben. Die SPI-Taktfrequenz und ARM Timer sind ebenfalls abhängig vom System Clock.

Aus irgendeinem seltsamen Grund ist ab dem Kernel 4.4.9 "DISABLE UART" die Standard-Voreinstellung - vermutlich, um Konflikte mit Bluetooth zu verhindern. Um die serielle Schnittstelle zu aktivieren, müssen Sie `enable_uart=1` in der Datei `/boot/config.txt` setzen. Dies behebt angeblich auch die `core_freq`-Problematik, so dass der obige Eintrag ggf. nicht mehr notwendig ist. Wer sicher gehen will, schreibt auf jeden Fall beides in die Datei `/boot/config.txt`:

```
core_freq=250
enable_uart=1
```

Wenn Sie Bluetooth nicht verwenden, ist es möglich, die beiden seriellen Ports im Devicetree zu vertauschen oder Bluetooth mittels Devicetree-Overlay ganz abzuschalten. Zum Vertauchen fügen Sie am Ende der Datei `/boot/config.txt` die Zeile

```
dtoverlay=pi3-miniuart-bt
```

hinzu. Nach einem Neustart funktioniert die UART-Zuordnung wieder wie in den Vorgängerversionen. Mittels `pi3-disable-bt` kann man Bluetooth komplett abschalten. Dann muss aber auch noch das bluetooth-Modem deaktiviert werden, was über `systemctl` mit dem folgenden Kommando erreicht wird:

```
sudo systemctl disable hciuart
```

Zum Testen, ob die serielle Schnittstelle auch wunschgemäß arbeitet, verbinden Sie die GPIO-Ports TX und RX des UART miteinander (Pins 8 und 10 der Stiftleiste). Dmit wird jedes gesendete Byte gleich wieder auf den Empfang zurückgegeben. Dann starten Sie das Teminalprogramm Minicom (ggf. muss das erst nachinstalliert werden) im lokalen Modus, d. h. ohne Senden von Init-Strings etc.:

```
minicom -b 9600 -o -D /dev/ttyAMA0
```

Jedes Zeichen, das Sie eintippen wird nun als Echo auf dem Bildschirm angezeigt (sofern alles in Ordnung ist). Einen Zeilenvorschub erreichen Sie übrigens mit "Strg-J" (Linefeed). Beenden läßt sich Minicom mit der Tastenfolge "Strg-A" und "q".

Verwendung eines USB-Seriell-Adapters

Will man Bluetooth behalten, hilft ein USB-seriell-Adapter. Dieses sieht meist aus wie ein Kabel mit einem USB-A-Stecker an einem Ende und einem 9-poligen DB9-Stecker auf der anderen Seite. In den seriellen Anschluss ist ein Hardware-Chipsatz eingebaut, der die notwendige USB-seriell-Umwandlung durchführt. Stecken Sie den Adaper am RasPi ein und wartet Sie einige Sekunden. Dann starten Sie in der Konsole das Kommando `dmesg`. Am Ende des Outputs sollte nun das Protokoll zum Adapter erscheinen, z. B.:

```
  ...
[ 2329.049523] usb 1-1.4: new full-speed USB device number 7 using dwc_otg
[ 2329.178464] usb 1-1.4: New USB device found, idVendor=0403, idProduct=6001
[ 2329.178489] usb 1-1.4: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[ 2329.178502] usb 1-1.4: Product: USB Serial Converter
[ 2329.178514] usb 1-1.4: Manufacturer: FTDI
[ 2329.178526] usb 1-1.4: SerialNumber: FT9GC7Y9
[ 2329.189325] ftdi_sio 1-1.4:1.0: FTDI USB Serial Device converter detected
[ 2329.189470] usb 1-1.4: Detected FT232RL
[ 2329.190563] usb 1-1.4: FTDI USB Serial Device converter now attached to ttyUSB0
```

Bei diesem Beispiel handet es sich im Adapter um den bekannten Chip von Future Technology Devices International (FTDI). Wichtig für später sind die Hersteller- und Geräte-Id (idVendor=0403, idProduct=6001). Diese können Sie auch später noch mit dem Kommando `lsusb` ermitteln, z. B.:

```
   ...
Bus 001 Device 008: ID 0403:6001 Future Technology Devices International, Ltd FT232 USB-Serial (UART) IC
Bus 001 Device 003: ID 0424:ec00 Standard Microsystems Corp. SMSC9512/9514 Fast Ethernet Adapter
Bus 001 Device 002: ID 0424:9514 Standard Microsystems Corp.
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
   ...
```

Wenn alles gut gegangen ist, ist Ihr Adapterkabel jetzt installiert und einsatzbereit, aber welches ist unter Linux der zugehörige serielle Port? Wie bei allen Geräten unter Linux, befindet sich die Gerätedatei unterhalb von `/dev/`. Da es sich um einen USB-basierten seriellen Port handelt, heißt er in der Regel `/dev/ttyUSB0`. Er kann aber, je nach Treiber, auch andere Namen haben, z. B. `/dev/ttyACM0`.

Um den Adapter auf einen festen Namen "festzunageln", kann man eine `udev`-Regel verwenden. Dazu erstellen Sie an besten eine neue Datei im Verzeichnis `/etc/udev/rules.de`, die Sie beispielsweise `seriell.rules` nennen (wichtig ist nur die Endung "`.rules`"). Dort tragen Sie für den Adapter eine Zeile nach

dem Muster

```
SUBSYSTEM=="tty", ATTRS{idVendor}=="vvvv", ATTRS{idProduct}=="pppp", SYMLINK+="geraete_name"
```

ein. Für "vvvv" wird die Vendor-Id und für "pppp" die Product-Id eingesetzt. Als Gerätenamen wählen Sie irgend etwas Aussagekräftiges, z. B. "Seriell" oder auch "COM1". Sie können dann immer über das symbolische Link `/dev/Seriell` auf die Schnittstelle zugreifen. Hier noch einige Beispiele für gängige Produkte:

```
SUBSYSTEM=="tty", ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6001", SYMLINK+="FTDI"
SUBSYSTEM=="tty", ATTRS{idVendor}=="19d2", ATTRS{idProduct}=="0031", SYMLINK+="ONDA"
SUBSYSTEM=="tty", ATTRS{idVendor}=="12d1", ATTRS{idProduct}=="14ac", SYMLINK+="HUAWEI"
SUBSYSTEM=="tty", ATTRS{idVendor}=="067b", ATTRS{idProduct}=="2303", SYMLINK+="PROLIFIC"
```

Nach einem Reboot steht das Symlink dann zur Verfügung. Um im laufenden Betrieb, etwa nach dem Erstellen der Rules-Datei, zu testen, ob es klappt, können Sie die Datei mittels

```
sudo udevadm trigger
```

einbinden.

Wenn Sie wollen, dass alle User auf die Schnittstelle zugreifen können (nicht nur diejenigen, die zur Gruppe dialout gehören), hängen Sie noch ', MODE="0666"' an die Regel an.

Programmierung der seriellen Schnittstelle in C

Die Programmierung der seriellen Schnittstelle funktioniert im Prinzip wie bei jedem UNIX- oder Linux-Rechner und ist kein Hexenwerk. Natürlich sollte auf der Gegenseite auch ein System sitzen, dessen serielle Schnittstelle funktioniert - und die auf die gleiche Datenrate eingestllt ist. Übrigens kann analog der "echten" seriellen Schnittstelle ein USB-Seriell-Adapter an der USB-Schnittstelle angesprochen werden - oder ein anderes USB-Gerät, was sich auf ein Pseudo-Seriell-Device stützt. Denken Sie auch daran, dass so eine Schnittstelle sich nicht unbedingt so "brav" wie eine Festplatte verhält. Unter Umständen antwortet das angeschlossene Gerät nicht wie erwartet oder gar nicht. Manche Mess-Systeme spucken auch ständig Daten auf die Leitung oder benötigen die Ansteuerung bestimmter Handshake-Leitungen. Auch die Buchsenbeschaltung selbst kann nicht so sein, wie Sie das erwarten.

Zum Testen Ihres Programms können Sie die Schnittstelle mit sich selbst "reden" lassen, indem Sie TXD und RXD über einen Widerstand von ca. 1,5 - 2 Kiloohm verbinden. Der Widerstand ist im Prinzip nicht nötig, schützt aber die Leitungen des GPIO, falls sie versehentlich mal beide als Ausgang programmiert wurden (z. B. durch eine in Vergessenheit geratene Init-Routine).

Für alle Programme werden einige Standard-Include-Dateien benötigt, die Sie am Besten immer gleich in Ihre Programme einbinden:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <termios.h>
```

Für die Angabe der Baudrate gibt es in den Include-Dateien definierte Konstante, die Sie auch verwenden sollten. Die langsameren Übertragungsgeschwindigkeiten orientieren sich an den inzwischen schon historischen Fernschreibern, wobei die ursprüngliche Rate von 150 Bit/s (bps) immer wieder verdoppelt wurde:

```
B1200, B2400, B4800, B9600, B19200, B38400
```

Beim PC kamen dann noch hinzu (Modem bzw. maximale Geschwindigkeit der seriellen PC-Schnittstelle):

```
B57600, B115200
```

Mit USB ginge es natürlich noch schneller, so dass der Raspberry Pi auch noch die folgenden Angaben kennt:

```
B230400, B460800, B500000, B576000, B921600, B1000000, B1152000,
B1500000, B2000000, B2500000, B3000000, B3500000, B4000000
```

Es gibt aber nur sehr selten Anlass, Datenraten über 115200 bps zu verwenden. Serielle Geräte sind meist relativ langsam und liefern auch nicht allzuviele Daten. Zudem ist es auch schon für die Software höchst anspruchsvoll, die ankommenden (oder abgehenden) Daten mit hoher Geschwindigkeit zu verarbeiten. Ganz abgesehen davon, dass "normale" serielle Kabel etc. gar nicht für solche Raten spezifiziert sind. Daher mein Tipp: Bleiben Sie bei 9600 bps oder 19200 bps, das reicht in fast allen Fällen aus und Sie sind auf der sicheren Seite.

Ein wichtiger Punkt betrifft die Zugriffsrechte. Normalerweise ist der User "pi" in die Gruppe der seriellen Schnittstelle (anfangs "dialout", jetzt "tty") eingetragen und kann somit darauf zugreifen. Arbeiten Sie mit einer anderen Userkennung, muss ggf. der User in die entsprechende Gruppe eingetragen werden:

```
sudo usermod -a -G dialout <Username>
        bzw.
sudo usermod -a -G tty <Username>
```

Oder sie ändern gleich die Zugriffsrechte für die Schnittstelle:

```
sudo chmod a+rw /dev/ttyAMA0
```

Die Änderungen der Gruppe werden erst nach dem nächsten Login des Users wirksam.

Die folgenden Links beschreiben die serielle Programmierung recht umfangreich, weshalb im Anschluss nur einige Tipps und Handreichungen gegeben werden.

- Programmierung der seriellen Schnittstelle beschreibt den Zugriff in C, Bearbeiten von längeren Eingabepuffern, Behandlung von Timeouts und IO-Controls. Das Listung zum Öffen der Schnittstelle unten stammt aus der Webseite.
- Serial Port programming in C (PDF) behandelt die C-Programmierschnittstlle unter POSIX - das sollten alle Linux-Systeme beherrschen.
- Serial-Programming-HOWTO behandelt ebenfalls die Programmierung der seriellen Schnittstelle, jedoch auf PC-Basis. Das Dokument ist sehr knapp gehalten, die letzte Version stammt von 2001.

Die im folgenden gezeigte Schnittstellenprogrammierung funktioniert analog natürlich auch bei USB-seriellen Schnittstellen (`/dev/ttyUSB0` etc.), es muss nur beim Öffnen der Schnittstelle das entsprechende Device angegeben werden.

Öffnen der Schnittstelle

Die folgende C-Funktion stellt eine allgemein verwendbare Routine für das Öffnen der Schnittstelle dar. Da es zahllose Optionen gibt (siehe Headerdateien bzw. obige Dokumentnation), werden hier die notwendigsten Optionen so gesetzt, dass sie die häufigsten Praxisfälle abdecken. Auf zwei besondere Optionen, `options.c_cc[VMIN]` und `options.c_cc[VTIME]`, wird weiter unten noch genauer eingegangen. Das Datenformat wird auf 8 Datenbist, 1 Stoppbit, keine Parität gesetzt. In der Praxis kommen eigentlich nur drei Formate vor:

- No parity (8N1):

```
    options.c_cflags &= ~PARENB
    options.c_cflags &= ~CSTOPB
    options.c_cflags &= ~CSIZE;
    options.c_cflags |= CS8;
```

- Even parity (7E1):

```
    options.c_cflags |= PARENB
    options.c_cflags &= ~PARODD
    options.c_cflags &= ~CSTOPB
    options.c_cflags &= ~CSIZE;
    options.c_cflags |= CS7;
```

- Odd parity (7O1):

```
    options.c_cflags |= PARENB
    options.c_cflags |= PARODD
    options.c_cflags &= ~CSTOPB
    options.c_cflags &= ~CSIZE;
    options.c_cflags |= CS7;
```

```
int open_serial(void)
  {
  /*
   * Oeffnet seriellen Port
   * Gibt das Filehandle zurueck oder -1 bei Fehler
   *
   * RS232-Parameter:
   * 19200 bps, 8 Datenbits, 1 Stoppbit, no parity, no handshake
   */

   int fd;                     /* Filedeskriptor */
   struct termios options;     /* Schnittstellenoptionen */

   /* Port oeffnen - read/write, kein "controlling tty", Status von DCD ignorieren */
   fd = open("/dev/ttyAMA0", O_RDWR | O_NOCTTY | O_NDELAY);
   if (fd >= 0)
     {
     /* get the current options */
     fcntl(fd, F_SETFL, 0);
     if (tcgetattr(fd, &options) != 0) return(-1);
     memset(&options, 0, sizeof(options)); /* Structur loeschen, ggf. vorher sichern
                                              und bei Programmende wieder restaurieren */
     /* Baudrate setzen */
     cfsetispeed(&options, B19200);
     cfsetospeed(&options, B19200);

     /* setze Optionen */
     options.c_cflag &= ~PARENB;         /* kein Paritybit */
     options.c_cflag &= ~CSTOPB;         /* 1 Stoppbit */
     options.c_cflag &= ~CSIZE;          /* 8 Datenbits */
     options.c_cflag |= CS8;

     /* 19200 bps, 8 Datenbits, CD-Signal ignorieren, Lesen erlauben */
     options.c_cflag |= (CLOCAL | CREAD);

     /* Kein Echo, keine Steuerzeichen, keine Interrupts */
     options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG);
     options.c_iflag = IGNPAR;           /* Parity-Fehler ignorieren */
     options.c_oflag &= ~OPOST;          /* setze "raw" Input */
     options.c_cc[VMIN]  = 0;            /* warten auf min. 0 Zeichen */
     options.c_cc[VTIME] = 10;           /* Timeout 1 Sekunde */
     tcflush(fd,TCIOFLUSH);              /* Puffer leeren */
     if (tcsetattr(fd, TCSAFLUSH, &options) != 0) return(-1);

     }
  return(fd);
  }
```

Besonders wichtig sind die beiden Optionen `c_cc[VTIME]` und `c_cc[VMIN]`. In `c_cc[VTIME]` wird die Wartezeit in Zehntelsekunden und in `c_cc[VMIN]` das Minimum der zu lesenden Bytes angegeben. Die folgenden vier

Konstellationen sind denkbar:

- **1. Fall: c_cc[VTIME] > 0 und c_cc[VMIN] > 0**
  `read()` liefert MIN Bytes, bevor die Zeit TIME abläuft oder `read()` liefert weniger als MIN Bytes, weil die Zeit TIME abgelaufen ist. Sind noch keine Daten empfangen worden, wartet `read()` auf min. ein Byte. Wenn das erste Byte gelesen wurde, läuft der Timer los, wobei jedes ankommende Byte den Timer wieder neu startet. Diese Methode ist günstig, wenn man große Datenmengen lesen, aber auch auf einzelne Zeichen reagieren muss. Aber es kann eine Blockierung stattfinden.
- **2. Fall: c_cc[VTIME] = 0 und c_cc[VMIN] > 0**
  `read()` liefert mindestens MIN Bytes, sobald diese eingetroffen sind. Dieser Modus ist günstig, wenn möglichst viele Bytes mit einem `read()` gelesen werden sollen. Andererseits kann man auch auf ein einziges Byte reagieren (MIN = 1). Ist MIN größer als die Anzahl der bei `read()` angegebenen Zeichen, wird gewartet, bis MIN Bytes gelesen, aber nur n Bytes an `read()` geliefert wurden; ein zweites `read()` liefert dann den Rest. Auch hier kommt es zur Blockierung, wenn nicht genügend Bytes eintreffen.
- **3. Fall: c_cc[VTIME] > 0 und c_cc[VMIN] = 0**
  Diese Einstellung erlaubt es, das Lesen mit Timeout zu programmieren. Sobald ein Byte eintrifft, liefert `read()` dieses ab. Wenn die Zeit TIME seit dem Aufruf von `read()` verstrichen ist, liefert `read()` 0 (gelesene Bytes) zurück.
- **4. Fall: c_cc[VTIME] = 0 und c_cc[VMIN] = 0**
  `read` liefert die Anzahl Bytes, die anliegen. Sind keine Daten vorhanden, wird sofort 0 (gelesene Bytes) zurückgegeben. Der Treiber wartet also niemals auf Daten, sondern kehrt immer sofort zurück.

Wenn das Programm nicht ewig auf eine Eingabe warten soll, nimmt man also am besten den dritten Fall.

Bytes senden

Für das Senden wird in der Regel die Funktion `write()` verwendet, deren erster Parameter der Filedeskriptor ist. Weitere Parameter sind die Adresse des Sendepuffers und die Anzahl der zu sendenden Bytes. Es muss auf jeden Fall überprüft werden, wieviele Bytes gesendet wurden (Rückgabewert von `write()`) und ob auch alle Bytes gesendet wurden.

```
int sendbytes(char * Buffer, int Count)
/* Sendet Count Bytes aus dem Puffer Buffer */
  {
  int sent;  /* return-Wert */
  /*  Daten senden */
  sent = write(fd, Buffer, Count);
  if (sent < 0)
    {
    perror("sendbytes failed - error!");
    return -1;
    }
  if (sent < Count)
    {
    perror("sendbytes failed - truncated!");
    }
  return sent;
  }
```

Bytes empfangen

Für das Empfangen wird in der Regel die Funktion `read()` verwendet, deren erster Parameter der Filedeskriptor ist. Weitere Parameter sind die Adresse des Sendepuffers und die maximale Anzahl der zu empfangenden Bytes. Die Funktion gibt die Anzahl der empfangenen Bytes zurück, wobei dieser Wert auch 0 sein kann. Das Verhalten von `read()` hängt von den Konfigurationswerten `c_cc[VTIME]` und `c_cc[VMIN]` ab.

Bei der in `open_serial()` getroffenen Einstellung kehrt `read()` auf jeden Fall nach einer Sekunde zurück, ggf. ohne Zeichen empfangen zu haben. Dies ist bei der Programmierung zu berücksichtigen.

Das erste Programmfragment liest bis zu 100 Zeichen in einen Puffer:

```
char buf[101];   /* Eingabepuffer */
int anz;         /* gelesene Zeichen */
   ...

anz = read(fd, (void*)buf, 100);
if (anz < 0)
  perror("Read failed!");
else if (anz == 0)
   perror("No data!");
else
  {
  buf[anz] = '\0';      /* Stringterminator */
  printf("%i Bytes: %s", anz, buf);
  }
   ...
```

Das Verfahren eignet sich insbesondere dann, wenn Sie wissen, wieviele Bytes zu erwarten sind. Andernfalls gehen Sie vorsichtiger vor und lesen zeichenweise. Diese Methode eignet sich auch gut, wenn auf ein bestimmtes empfangenes Byte reagiert werden soll (Enter, Newline etc.):

```
char buf[101];   /* Eingabepuffer für die komplette Eingabe */
int anz;         /* gelesene Zeichen */
char c;          /* Eingabepuffer fuer 1 Byte */
int  i;          /* Zeichenposition bzw Index */
   ...

i = 0;
do               /* Lesen bis zum Carriage Return, max. 100 Bytes */
  {
  anz = read(fd, (void*)&c, 1);
  if (anz > 0)
    {
    if (c != '\r')
      buf[i++] = c;
    }
  }
while (c != '\r' && i < 100 && anz >= 0);

if (anz < 0)
  perror("Read failed!");
else if (i == 0)          /* Nur zur Demo, dass auch mal nix kommt */
   perror("No data!");    /* im Normalbetrieb loeschen!            */
else
  {
  buf[i] = '\0';        /* Stringterminator */
  printf("%i Bytes: %s", i, buf);
  }
   ...
```

Sie sehen schon, das Empfangen wirft mehr Probleme auf, als das Senden. Hier muss immer eine speziell an die Kommunikation angepasste Lösung entwickelt werden.

Programmieren der seriellen Schnittstelle in Python

Für das Ansprechen der seriellen Schnittstelle mit Python wird das Modul `pySerial` benötigt. Die Installation erfolgt mit der Paketverwaltung ganz einfach:

```
sudo apt-get -y install python-serial
```

Für Python, Version3, ersetzen Sie "python-serial" durch "python3-serial". Die Dokumentation von pySerial findet man auf https://github.com/pyserial/pyserial. Dort gibt es unter "pySerial API" eine

vollständige Beschreibung der Klassen. Einige davon sind allerdings plattformspezifisch (Linux, Windows). Mit folgendem Programm werden auf der Tastatur eingegebene Zeichen an ein externes Device gesendet und von dort empfangene Zeichen auf der Konsole ausgegeben. Zum Test kann man beispielsweise zwei RasPis über ein Kabel mit gekreuzten Verbindungen zwischen RXD (Receive Data) und TXD (Transmit Data) miteinander verbinden und auf beiden das Programm laufen lassen. Bei nur einm Raspberry Pi kann man auch nur diese beiden Pins miteinander verbinden. Dann werden alle gesendeten Zeichen sofort wieder empfangen.

Beim Öffnen der seriellen Schnittstelle kann man, wie auch bei C, die Einstellungen vornehmen. Diese Einstellungen lassen sich aber auch nachträglich festlegen oder ändern. Eine typische Initialisierung könnte folgendermaßen aussehen:

```
  ser = serial.Serial(
                port="/dev/ttyS0",
                baudrate = 9600,
                parity=serial.PARITY_NONE,
                stopbits=serial.STOPBITS_ONE,
                bytesize=serial.EIGHTBITS,
                timeout=30
           )
```

Die Standard-Voreinstellung ist 8 Datenbist, 1 Stoppbit, keine Parität - diese Angaben dürfen daher auch fehlen. Obligatorisch sind die Angabe von Port und Baudrate. Zum Lesen der empfangenen Zeichen muss man eine nicht-blockierende Funktion verwenden, wenn das Programm auch laufend prüfen muss, ob eine Taste gedrückt wurde etc. Die Methode `ser.read()` blockiert nicht, wenn im Konstruktor der timeout-Parameter auf 0 gesetzt wird. Mit der Methode `ser.isopen()` kann man feststellen, ob der Zugriff auf die Schnittstelle geklappt hat. Die Methoden `serialPort.flushInput()` und `serialPort.flushOutput()` dienen zum Löschen von Empfangs- und Sendepuffer.

Einige Eigenschaften für den Handshake kann man bei Linux-PCs setzen, bei denen die Steuerleitungen der seriellen Schnittstelle vorhanden sind. Dazu dienen folgende Variablen:

- `ser.xonxoff`: Einschalten (True) oder Abschalten (False) des Software-Handshake
- `ser.rtscts`: Einschalten (True) oder Abschalten (False) des Hardware-Handshake mit RTS/CTS
- `ser.dsrdtr`: Einschalten (True) oder Abschalten (False) des Hardware-Handshake mit DTR/DSR

Die Steuerleitungen lassen sich auch Setzen (RTS, DTR, Break=TXD) oder Lesen (CTS, DSR, RI). Beim Setzen sind die Wete 0 und 1 für `val` möglich - entsprechend dem gewünschten Zustand der Ausgangsleitungen:

```
ser.setRTS(val)
ser.setDTR(val)
ser.setBreak(val)
val = ser.getCTS()
val = ser.getDSR()
val = ser.getRI()
```

Beim Raspberry Pi muss man ggf. die Steuerleitungen über GPIO-Pins realisieren.

Zum Senden von Bytes/Zeichen wird die Methode `write()` verwendet, zum Beispiel:

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

from serial import Serial

ser = Serial("/dev/ttyUSB0", baudrate=9600, timeout=1)
for i in range(10):
  ser.write(b"Hello World\n")
ser.close()
```

In Python 3 werden unterschiedliche Typen für Text (string) und binäre Daten (bytes) benutzt. Bei der seriellen Schnittstelle muss man binäre Daten versenden. Der Text muss also in einen Binärstring umgewandelt werden. Dies kann entweder mit einem vorangestellten 'b' geschehen (z. B.: `ser.write(b"Hello World\n")`) oder mit der Methode `encode()` (z. B.: `ser.write("Hello World\n".encode())`). Als Parameter kann bei `encode()` die Kodierung angegeben werden (z. BV. utf-8, utf-16, latin-1). Ohne Parameter wird die Default-Codierung des Betriebssystems verwendet. Bei Python 2 ist die Umcodierung nicht nötig.

Zum Einlesen von Daten kann man eine der Methoden `read(n)`, `readline(nmax)` oder `readlines()` verwenden. In Klammern wird die Anzahl der zu lesenden Bytes bzw. die maximale Anzahl der zu lesenden Bytes angegeben. Wird das Timeout erreicht, owohl noch nicht die angegebene Anzahl von Zeichen gelesen wurde, so bricht die Funktion ab. Invers zu `encode()` wandelt die Methode `decode()` den Bytestring wieder in einen Textstring zurück. Mittels `inWaiting()` kann man ermitteln, wieviele Zeichen noch im Puffer warten. Bei den folgenden Beispielen wurden am USB-Seriell-Adapter einfach die Pins TXD und RXD gebrückt. Das erste Programm schickt einfach die eingegebenen Zeichen zurück. #!/usr/bin/env python # -*- coding: utf-8 -*- import serial ser = serial.Serial("/dev/ttyUSB0", baudrate=19200, timeout=0.1) if (ser.isOpen() == True): ser.close() ser.open() try: while True: ch = ser.read() ser.write(ch) # Keyboard Interrupt abfangen, zum beenden mit [STRG]+[C]. except(KeyboardInterrupt, SystemExit): print("Abbruch") ser.close()

Beim nächsten Programm wird der ASCII-Zeichensatz ausgegeben und wieder eingelesen.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import serial
import time

outStr = ''
inStr = ''

ser = serial.Serial("/dev/ttyUSB0", 19200, timeout=2)
if (ser.isOpen() == True):
  ser.close()
ser.open()

for i, a in enumerate(range(33, 126)):
  outStr += chr(a)
  ser.write(outStr)
  time.sleep(0.05)
  inStr = ser.read(ser.inWaiting())

print "inStr =  " + inStr
print "outStr = " + outStr
if(inStr == outStr):
  print "Bingo!"
else:
  print "Ooops!"
ser.close()
```

Im folgenden Beispiel kommunizieren zwei Programme über die serielle Schnittstelle. Für den Test kann man beide Programme in verschiedenen Shell-Fenstern laufen lassen. `sender.py` sendet einen String ("Zählerstand: nn"), wobei der Zähler dauern inkrementiert wird. `receive.py` empfängt den String und gibt ihn am Bildschirm aus. Zuerst das Sende-Programm `sender.py`:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import time
import serial

counter = 0

ser = serial.Serial(port='/dev/ttyUSB0', baudrate = 19200, timeout = 1)
if (ser.isOpen() == True):
  ser.close()
```

```
ser.open()

try:
  while 1:
    ser.write('Zählerstand: %d \n'%(counter))
    time.sleep(1)
    counter += 1

# Keyboard Interrupt abfangen, zum beenden mit [STRG]+[C].
except(KeyboardInterrupt, SystemExit):
  print("Abbruch")
  ser.close()
```

Und nun das Empfangsprogramm `receive.py`:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import time
import serial

ser = serial.Serial(port='/dev/ttyUSB0', baudrate = 19200, timeout=1)
if (ser.isOpen() == True):
  ser.close()
ser.open()

try:
  while 1:
    x = ser.readline().rstrip()  # LF am Ende entfernen
    print x

# Keyboard Interrupt abfangen, zum beenden mit [STRG]+[C].
except(KeyboardInterrupt, SystemExit):
  print("Abbruch")
  ser.close()
```

Interessant ist die serielle Kommunikation auch, wenn man den Raspberry Pi mit einem Arduino verbinden will. Der Arduino hängt dann per USB einerseits an der seriellen Schnittstelle und wird andererseits über die USB-Buchse mit Energie versorgt. Je nach Modell wird die Schnittstelle als `/dev/ttyAMC0` (Arduino UNO) oder als `/dev/ttyUSB0` (Arduino NANO) identifiziert. Hier kann eine automatische Ermittlung des Ports nach folgendem Muster helfen (die sich auch auf weitere serielle Komponenten erweitern liesse):

```
   ...
# Arduino NANO detektieren
cmd = "dmesg | grep ttyUSB | grep FTDI | head -1 | sed -e 's/.* //'"
process = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE)
ArduinoNano = str.strip(process.stdout.read())
if ArduinoNano:
  PORT = '/dev/tty'+ArduinoNano
else:
  # Arduino UNO detektieren
  cmd = "dmesg | grep ttyACM | head -1 | sed -e 's/.*tty//' -e 's/:.*//'"
  process = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE)
  ArduinoUno = str.strip(process.stdout.read())
  if ArduinoUno:
    PORT = '/dev/tty'+ArduinoUno
  else:
    exit(1)
  ...
```

Die Verbindung zum Arduino wird nun mit dem folgenden Befehl aufgebaut, egal welcher Arduino angestöpselt ist.

```
ser = serial.Serial(port=PORT, baudrate = 9600, timeout=30)
```

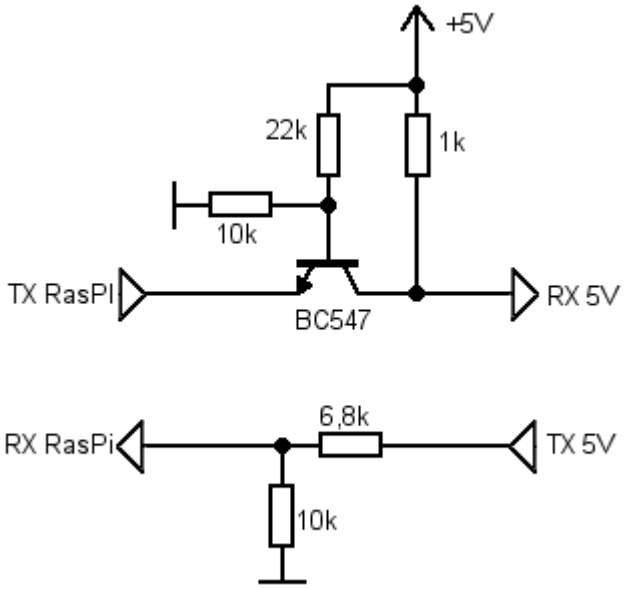
Links

- pySerial Dokumentnation

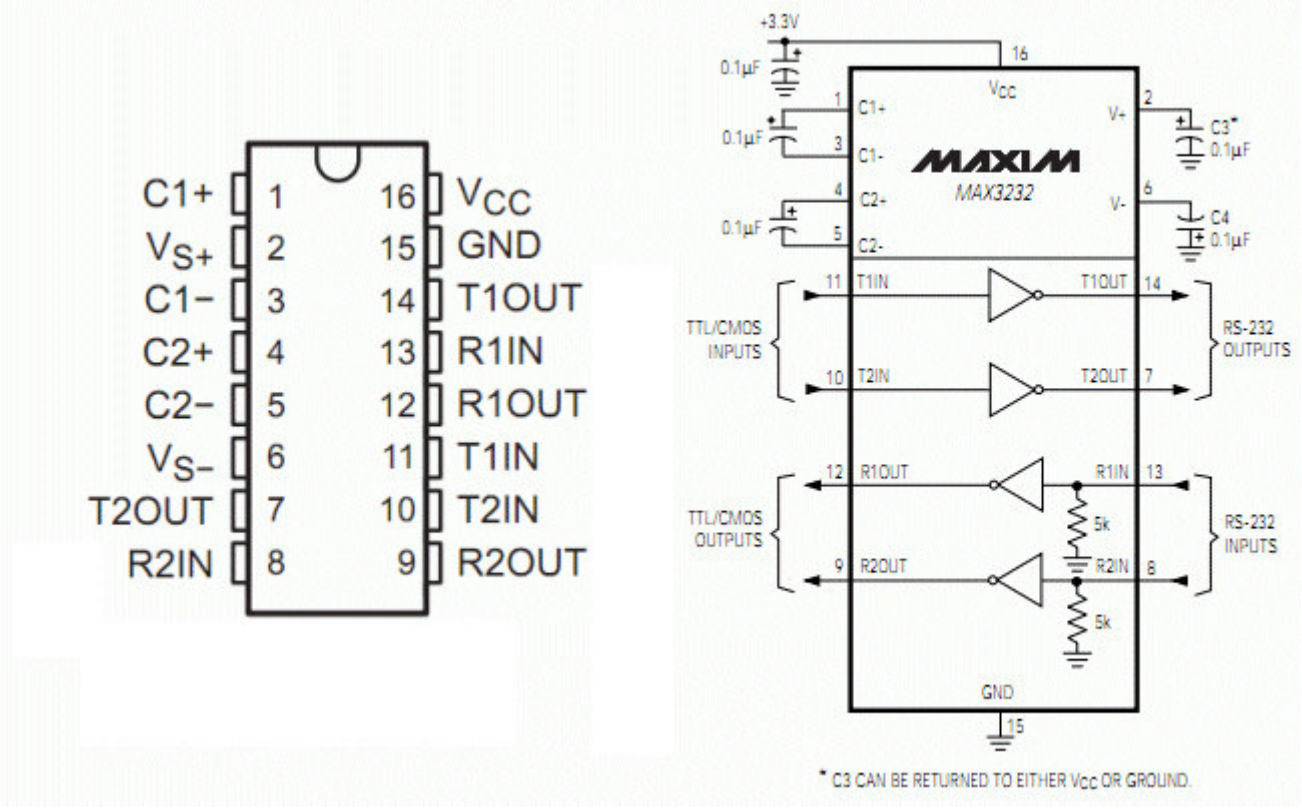
- pySerial

Pegelanpassung der seriellen Schnittstelle

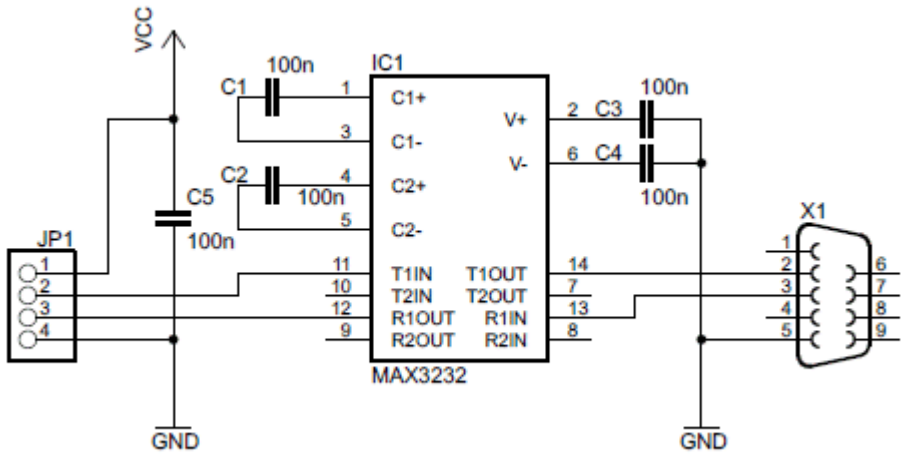
Da der Raspberry Pi mit nur 3,3 V arbeitet, muss fast immer eine Pegelanpassung vorgenommen werden. Handelt es sich beim Kommunikationspartner ebenfalls um ein Mikrocontrollersystem, das aber mit 5 V arbeitet, ist die Anpassung relativ einfach. Der Sendeausgang TX des Raspberry mit ca. 2,8 V reicht manchmal schon aus, um den 5-V-Controller anzusteuern. Jedoch liegt diese Ausgangsspannung im Grenzbereich - mal klappte es mal nicht. Die Schaltung im Bild oben dient dazu, das Digitalsignal des Raspberry Pi auf 5 V zu verstärken. Wenn am Eingang mehr als etwa 1,5 V anliegen, sperrt der Transistor, der in BAsisschaltung betrieben wird. Am Ausgang liegt über den 1-Kiloohm-Widerstand eine Spannung von 5 V an. Führt der Eingang Masse-Potential, ist der Transistor durchgeschaltet und der Ausgang liegt ebenfalls auf nahezu Masse-Potential. Auf der Eingangsseite genügt für die Wandlung ein Spannungsteiler, der das 5-V-Signal auf 3,3 V herunterteilt:



Sollen echte RS232-Pegel erzeugt werden, hilft ein entsprechender Konverterbaustein. Der MAX232 (Maxim) ist einer der ersten und beliebtesten TTL-zu-RS232-Konverter, nur arbeit er leider mit 5 V. Es gibt jedoch eine pinkompatible 3,3-V-Version, den MAX3232 (der sogar mit Spannungen zwischen 3 und 5,5 V arbeitet):



Zum Betrieb sind nur noch fünf Kondensatoren zu je 100 nF nötig und schon werden die 3,3-V-Pegel in ±12 V umgesetzt. Da keine Steuerleitungen verwendet werden, benötigen Sie nur jeweils einen der beiden Sende- oder Empfangskreise.

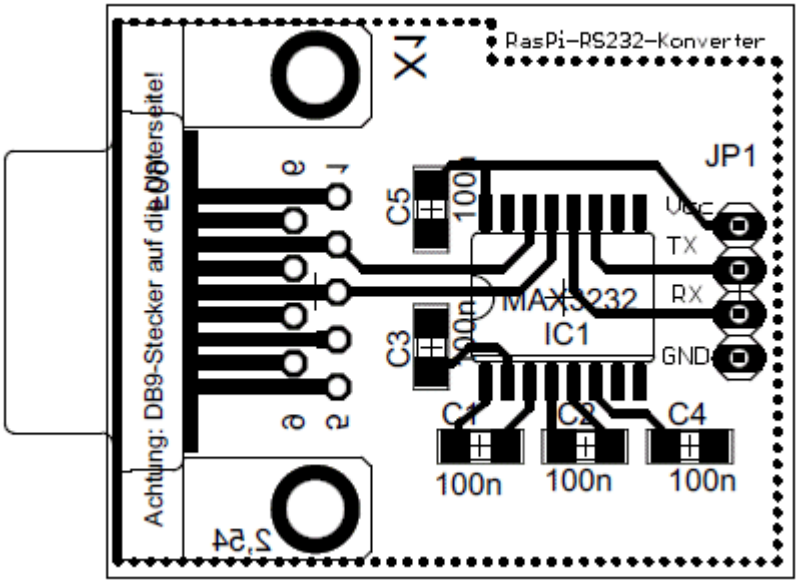


Die Anschlüsse des Raspberry Pi, TXD und RXD werden mit der "TTL/CMOS"-Seite des IC verbunden und eine 9-polige Sub-D-Buchse mit der RS232-Seite. Je nach Beschaltung der Gegenseite müssen eventuell die Pins 2 und 3 der Sub-D-Buchse vertauscht werden (Hier schlägt der "Fluch der seriellen Schnittstelle" zu). Die Zuordnung von JP1 und RasPi-Steckerleiste ist:

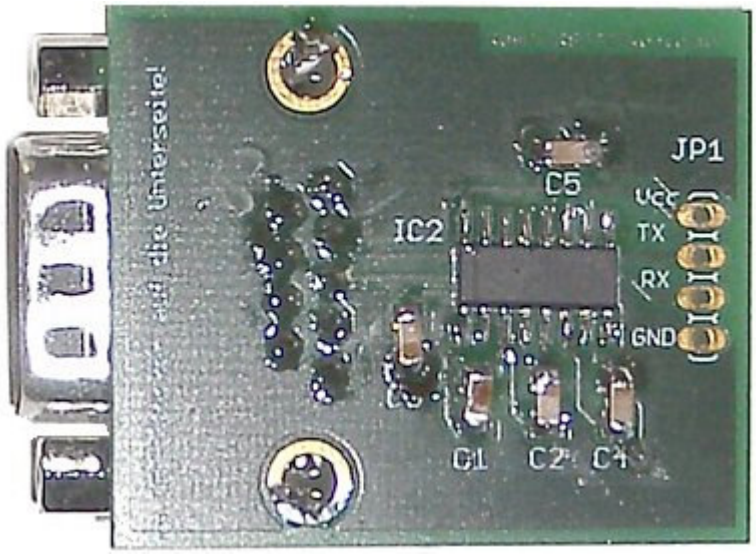
JP1	RasPi
1 (Vcc)	1
2 (TX)	8
3 (RX)	10
4 (GND)	6

Für den Anschluss einer RS232-Schnittstelle wurde entsprechend dem Schaltplan oben eine Platine entwickelt, die mit SMD-Bauteilen bestückt wird. Der MAX3232 ist im SO16-Gehäuse lieferbar, das wie auch die Kondensatoren der Größe 1206 ohne Fummelei zu verlöten ist. Die Platine kann Dank der SMD-

Technik einseitig bleiben, nur die Bestückungseite hat Kupferbahnen. Das folgende Bild zeigt den Bestückungsplan, wobei aus Gründen der Übersichtlichkeit die Masseflächen nicht gezeigt sind (bei EAGLE genügt der "ratsnest"-Befehl, um sie hervorzuzaubern).



Beachten Sie, dass der 9-polige Sub-D-Stecker auf der **Unterseite** der Platine sitzt und dessen Anschlusspins dann auf der Oberseite (Bestückungsseite) verlötet werden. Achtung: Vom MAX3232 gibt es zwei Bauformen unterschiedlicher Breite - bitte beim Einkauf beachten.



Die entsprechenden Dateien finden Sie unten bei den Links.

Links

- EAGLE-Schaltplan Adapterboard
- EAGLE-Board Adapterboard

Letzte Aktualisierung: 07/18/2018 16:26:29