Serieller Port – Florian Boemmel

1. Generelles

In unserem Projekt nutzen wir eine serielle USB-Verbindung zwischen Arduino und Raspberry Pi, um Daten und Befehle zwischen den beiden Geräten auszutauschen. Dieser Abschnitt beschäftigt sich ausschließlich nur mit dem Seriellen Port für die USB-Verbindung zwischen Raspberry Pi und Arduino.

2. Grundlagen

Die Grundlage jeder seriellen Kommunikation auf einem linuxbasiertem Betriebssystem ist das Öffnen und Konfigurieren eines Seriellen Ports. Serielle Ports werden unter Linux durch eine Datei repräsentiert.

3. Seriellen Port bestimmen

Zunächst muss der Port festgestellt werden, an dem der Arduino am Pi erkannt wird. Dazu kann entweder die Arduino IDE benutzt werden, oder über das Terminal.

1. Möchte man das Terminal nutzen, muss die Verbindung zum Arduino unbedingt getrennt werden und folgendes Kommando ausgeführt werden:

   ```
   pi@raspberrypi:~$ ls /dev/
   ```

   Nun muss zunächst überprüft werden, ob bereits ein ttyUSB oder ttyACM existiert. Jetzt muss der Arduino verbunden werden. Eine erneute Ausführung des Kommandos sollte jetzt einen weiteren Eintrag liefern (z.B. ttyUSB0). Dieser Eintrag ist nun der Serielle Port zu unserm Arduino.

2. Möchte man die Arduino IDE benutzen, öffnet man diese und verbindet den Arduino mit dem Pi. Anschließend wählt man im Menü:

   ```
   Tools → Serieller Port
   ```

   Hier wird nun der Port angezeigt. Jedoch muss beachtet werden, dass weitere angeschlossene Geräte unter Umständen auch angezeigt werden.

4. Seriellen Port implementieren

Das Implementieren des Seriellen Ports erfolgt mit C und unter der Verwendung der terminos API. Die terminos API unterstützt unterschiedliche Modi um einen Seriellen Port anzusprechen. Die zwei wichtigsten sind:

- **Cannonical Mode:** Dieser Modus ist Zeilenorientiert. Dies bedeutet, dass Eingaben gepuffert und durch den Benutzer bearbeitet werden können, bis ein carriage return (unter Linux CTRL-C) oder ein line feed (Zeilenumbruch) erkannt wird. Anschließend kann ein read(2) ausgeführt werden. Wird von Terminals verwendet.

- **NonCannonical Mode:** Dieser Modus ist im Gegensatz zum Cannonical Mode weder Zeilenorientiert noch werden Eingaben gepuffert oder können vom Benutzer bearbeitet werden. Dies bedeutet, dass ein Input sofort zur Verfügung steht. Zusätzlich muss hier eine Einstellung vorgenommen werden, unter welchen Umständen ein read(2) aufgerufen wird und wie sich dieses verhält.

Ausführliche Informationen über die Seriellen Ports und deren Programmierung können im **"The Serial Programming Guide for POSIX Operating Systems"** nachgelesen werden.

5. Seriellen Port öffnen und schließen

Zum Öffnen eines Seriellen Ports unter Linux wird der Systemaufruf open(2) verwendet:

```
int fd;
fd = open("/dev/ttyUSB0", O_RDWR | O_NOCTTY);
```

fd:	File-Deskriptor
/dev/ttyUSB0:	Serieller Port im Verzeichnis /dev
O_RDWR:	Serieller Port wird geöffnet für schreiben und lesen
O_NOCTTY:	Kein Terminal wird das öffnen kontrollieren

Wurde der Port erfolgreich geöffnet, erhält fd einen positiven Wert. Im Fehlerfall liefert open -1 zurück.

Zum schließen wird close(2) verwendet:

```
close(fd);
```

6. Seriellen Port konfigurieren

Zum Konfigurieren des Seriellen Ports wird, wie schon beschrieben, die terminos API benutzt. Die terminos Struktur sieht wie folgt aus:

```
struct termios
  {
    tcflag_t c_iflag;           /* input mode flags */
    tcflag_t c_oflag;           /* output mode flags */
    tcflag_t c_cflag;           /* control mode flags */
    tcflag_t c_lflag;           /* local mode flags */
    cc_t c_line;                        /* line discipline */
    cc_t c_cc[NCCS];            /* control characters */
    speed_t c_ispeed;           /* input speed */
    speed_t c_ospeed;           /* output speed */
#define _HAVE_STRUCT_TERMIOS_C_ISPEED 1
#define _HAVE_STRUCT_TERMIOS_C_OSPEED 1
  };
```

Nun werden die spezifischen Einstellungen für unser Projekt gesetzt.

```
struct termios SerialPortSettings;

tcgetattr(fd, &SerialPortSettings);             // Get the current attributes of the Serial port

cfsetispeed(&SerialPortSettings,B115200);        // Set Read Speed as 115200
cfsetospeed(&SerialPortSettings,B115200);        // Set Write Speed as 115200

SerialPortSettings.c_cflag &= ~PARENB;          // Disables the Parity Enable bit(PARENB),So No Parity
SerialPortSettings.c_cflag &= ~CSTOPB;          // CSTOPB = 2 Stop bits, here it is cleared so 1 Stop bit
SerialPortSettings.c_cflag &= ~CSIZE;           // Clears the mask for setting the data size ... enables set own data bits... see next line
SerialPortSettings.c_cflag |=  CS8;             // Set the data bits = 8
SerialPortSettings.c_cflag |= (CREAD | CLOCAL); // Enable receiver,Ignore Modem Control lines

cfmakeraw(&SerialPortSettings);                 // Setup Raw-Mode automatically
```

Für weitere Informationen und einer detaillierten Beschreibung der verwendeten sowie möglichen weiteren Einstellungen kann das unter Punkt 4 referenzierte Dokument verwendet werden.

Ein letzter Schritt setzt die Einstellungen in der terminos Struktur zu dem Seriellen Port:

```
tcsetattr(fd,TCSANOW,&SerialPortSettings));
```

Die Funktion liefert im Erfolgsfall eine 0 zurück. Danach ist der Serielle Port konfiguriert und für die Übertragung und das Empfangen von Daten eingerichtet.

7. Serieller Port schreiben

Das Schreiben auf dem Seriellen Port, wird durch den Systemaufruf write(2) realisiert.

```
write(fd, data, size);   // use write() to send data to port
                         // "fd"                  - file descriptor pointing to the opened serial port
                         // "write_buffer"        - address of the buffer containing data
                         // "sizeof(write_buffer)" - No of bytes to write
                         // returns the actual bytes written to the port
```

Dabei wird write() der File-Deskriptor, einem Puffer vom Typ const void * und eine Größe der zu schreibenden Daten in Bytes übergeben.
Die Größe der zu schreibenden Daten in Bytes gibt an, wie viele Bytes auf dem Puffer geschrieben werden sollen.

write() liefert im Erfolgsfall die Anzahl der geschriebenen Bytes zurück. Im Fehlerfall wird -1 zurückgegeben und 0 bedeutet, dass keine Daten geschrieben wurden.

8. Serieller Port lesen

Das Lesen auf dem Seriellen Port, wird durch den Systemaufruf read(2) realisiert.

```
read(fd, data , maxsize);        // Read the data
                                     // Write it to the location at <data>
                                     // Write <maxsize> bytes at maximum to not overrun buffersizes
```

Dabei wird read() der File-Deskriptor, einem Puffer vom Typ void * und eine Größe der zu lesenden Daten in Bytes übergeben. Die Größe der zu lesenden Daten in Bytes gibt an, wie viele Bytes aus dem File-Deskriptor in den übergebenen Puffer gelesen und anschließend geschrieben werden sollen.

read() liefert im Erfolgsfall die Anzahl der gelesenen Bytes zurück. Im Fehlerfall wird -1 zurückgegeben und 0 bedeutet, das Ende der Datei ist erreicht.

9. Probleme

1. Wechsel des Seriellen Ports

Zu Beginn des Projekts verwendeten wir einen Arduino Uno. Im späteren Verlauf wechselten wir jedoch auf einen Arduino Mega. Während der Debug-Tätigkeiten fiel immer wieder das Problem mit dem Seriellen Port auf. Manchmal wurde der Arduino unter ttyUSB0 erkannt und ein paar Tage später wieder unter ttyACM0 oder andersherum.

Schlussendlich stellte sich heraus, dass der Grund hierfür der Wechsel der Arduinos war. Einmal wurde mit dem Uno gearbeitet und ein anderes Mal mit dem Mega. Der Uno bekommt durch den Raspberry Pi den Port ttyUSB0 zugewiesen. Im Gegensatz dazu bekommt der Mega den Port ttyACM0 zugewiesen. Mögliche Gründe hierfür konnte ich noch nicht ermitteln.

2. Öffnen des Seriellen Ports

Zu Begin der Implementierung des Seriellen Ports, kam es sporadisch vor, dass der Port zwar ordnungsgemäß geöffnet wurde, jedoch anschließend nicht fehlerfrei arbeitete. Nach intensiver Recherche wurde ich fündig. In einigen Fällen, kann es vorkommen, dass das Programm zu schnell weiterarbeitet und z.B. ein write(2) zu früh ausführt. Ein usleep(3) von 200 Millisekunden direkt nach dem Öffnen löste das Problem dauerhaft.

3. Neustart des Arduino

Während der ersten Tests viel auf, dass keine Daten an den Arduino gesendet werden konnten. Ich untersuchte dies ausgiebig und stellte anschließend fest, dass nach dem unter Punkt 6 beschriebenen Tätigkeiten der Arduino neustartet. Genauer gesagt geschieht dies direkt nach dem Funktionsaufruf:

```
tcsetattr(fd,TCSANOW,&SerialPortSettings));
```

Eine Lösung hierfür ist denkbar einfach und mehrfach in den Foren als einzige Lösung bekannt. Man muss auf den Neustart des Arduinos warten. Eine sichere Zeitspanne ist dabei drei Sekunden. Realisierbar unter Linux mit sleep(3).

4. Schließen des Seriellen Ports

Es ist unabdingbar den Seriellen Port am Ende des Programms wieder zu schließen, um Ihn anschließend beim erneuten Starten des Programms wieder öffnen zu können. Das bereitete gerade während der Entwicklungsphase der Benutzeroberfläche einige Probleme. Stürzte die Benutzeroberfläche während eines Testlauf ab, konnte der Serielle Port nicht mehr geöffnet werden und es blieb nichts anderes übrig, als jedes Mal die USB-Verbindung zum Arduino zu trennen und wiederherzustellen.

10. Ausblick

Die geforderten Anforderungen wurden umgesetzt und der Serielle Port erweist sich als robust und bereitet keine Probleme. Dies wurde durch ausgiebige Tests bestätigt.

Jedoch wäre eine genauere Untersuchung der gesamten terminos Struktur ein weiterer möglicher Schritt. In diesem könnten eventuelle Verbesserungen in dem Gebiet der Performance erreicht werden.

Zusätzlich könnte noch eine Logik eingebaut werden, die die vorhandenen Ports überprüft und so selber den richtigen für den Arduino auswählt. Aktuell ist dieser im Code fest implementiert und muss bei einem Wechsel des Ports geändert und neu kompiliert werden.

Grafische Benutzeroberfläche – Florian Boemmel

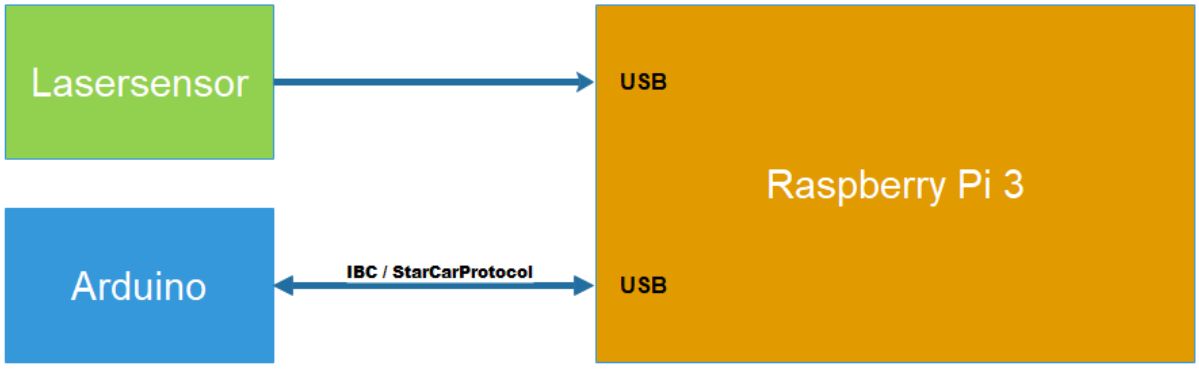
1. Generelles

Die grafische Benutzeroberfläche (im folgenden GUI bezeichnet) stellt, abstrakt dargestellt, dass Bindungsglied zwischen Benutzer und dem Fahrzeug da. Über diese, soll die Steuerung des Fahrzeugs erfolgen.

Die GUI soll unter den Aspekten der Skalierbarkeit und der einfachen Erweiterung durch andere Projektmitglieder entwickelt werden. Aus diesem Grund, einigte sich das Projektteam darauf, dass alle Module auf dem Raspberry Pi 3 Model B (Pi) in C++ entwickelt werden.

Module stellen hierbei externe Klassen da, diese unabhängig von der GUI entwickelt werden und anschließend in die GUI eingebunden werden müssen.
Folgende Module existieren:

- Lasersensor
- IBC / StarCarProtocol



2. Entwicklungsumgebung

Der Pi bietet eine große Auswahl an Möglichkeiten, jedoch ist seine Rechenleistung begrenzt und für einige Tätigkeiten, wie z.B. eine umfangreiche GUI direkt auf ihn zu programmieren eher ungeeignet.

Aus dem oben genannten Gründen wird die GUI in der Sprache C++ und dem GUI-Toolkit Qt5 realisiert. Qt bietet eine plattformunabhängige Programmierung. Dies bedeutet, dass die GUI auf einem stärkeren Rechner entwickelt und plattformunabhängige Funktionalitäten getestet werden können. Ein weiterer Vorteil liegt darin, dass die Zielplattform variabel ist, somit bleibt die GUI, selbst bei einem Wechsel des Betriebssystems auf dem Zielrechner einsetzbar. Lediglich die betriebssystemspezifischen Erweiterungen und Funktionen müssen ersetzt werden. Letztlich kann der mühsame Weg einer Cross-Kompilierung mit Qt umgangen werden. Das Projekt und alle seine Dateien können auf dem Pi kopiert und dort kompiliert werden.

3. Anforderungen

/G0101/ **Automatischer Start der Benutzeroberfläche:** Verbindet der Benutzer das Fahrzeug mit einer von Ihm gewählten Stromquelle, bootet der Raspberry Pi direkt in die Benutzeroberfläche des Fahrzeugs und verhindert so eine falsche Bedienmöglichkeit des Fahrzeugs.

/G0102/ **Initialisierung des Fahrzeugs:** Der Benutzer kann über einen Button das Fahrzeug initialisieren. Das bedeutet im konkreten Fall, dass zunächst ein Serieller Port geöffnet wird und das Inter Board Protocoll (IBC) gestartet wird. Weiterführende Steuerungsmöglichkeiten dürfen dem Benutzer zu diesem Zeitpunkt nicht zu Verfügung stehen.

/G0103/ **Modi Auswahl:** Der Benutzer hat die Möglichkeit zwischen zwei Betriebsmodi auszuwählen:

- Uhrsteuerung
- Controllersteuerung

Zusätzlich muss der Benutzer, ohne einen Modus auszuwählen, die Möglichkeit erhalten, sich die aktuellen Sensorwerte ansehen zu können.

/G0104/ **Neustart der Benutzeroberfläche:** Der Benutzer muss über ein Menü die Möglichkeit erhalten, die Benutzeroberfläche neu zu starten. Dies ist insbesondere bei Verbindungsproblemen zum Mikrocontroller unabdingbar.

/G0105/ **Beenden des Systems:** Der Benutzer muss über ein Menü die Möglichkeit erhalten, die Benutzeroberfläche sowie den Raspberry Pi ordnungsgemäß herunterfahren zu können.

/G0106/ **Uhrsteuerung:** Wählt der Benutzer den Modus Uhrsteuerung, muss diesem zunächst eine kurze Anleitung dargestellt werden, wie er die Uhren anzulegen hat. Hat der Benutzer diese Information verstanden, muss er diese bestätigen. Nach der positiven Bestätigung, muss dem Benutzer die Steuerung anhand von Bildern und Animationen verständlich erklärt werden. Zudem muss der Benutzer über einen Button die Möglichkeit gegeben werden, den Raumscan zu starten /F0111/.

/G0107/ **Controllersteuerung**: Wählt der Benutzer den Modus Controllersteuerung, wird dieser aufgefordert, den Controller griffbereit zu halten. Hat der Benutzer diese Information verstanden, muss er diese bestätigen. Nach der positiven Bestätigung, muss dem Benutzer die Steuerung anhand von Bildern und Animationen verständlich erklärt werden. Zudem muss der Benutzer über einen Button die Möglichkeit gegeben werden, den Raumscan zu starten /F0111/.

/G0108/ **Navigation:** Der Benutzer muss jederzeit die Möglichkeit erhalten, zur Modusauswahl /F0103/ zurückzukehren und einen anderen Modus wählen zu können. Dabei ist die Navigationstiefe in einem Modus unrelevant.

/G0109/ **Darstellung der Sensorwerte:** Dem Benutzer muss nach der Wahl, sich die Sensorwerte anzeigen zu lassen, eine Übersicht der vorhandenen Sensoren und deren aktuellen Werte dargestellt werden.

/G0110/ **Fehleranzeige:** Dem Benutzer muss eine Fehleranzeige bereitgestellt werden. Diese muss unabhängig von allen Darstellungen und Benutzereingaben jederzeit gut sichtbar sein. Weiterhin müssen dem Benutzer spezifische Details über einem Fehlerfall dargestellt werden.

/G0111/ **Raumscan**: Der Benutzer muss die Möglichkeit erhalten, nach der Wahl eines Modi, den Raumscan zu starten. Während der Raumscan läuft, werden dem Benutzer die Sensordaten dargestellt /F0109