

CAN-Bus mit Linux und Python

Grazer Linxstage 2014

Jörg Faschingbauer

Inhaltsverzeichnis

1 Basics

2 CAN Interfaces
3 Programmierung

4 Hardware, Kernel
5 Schluss

Übersicht

1 Basics

2 CAN Interfaces
3 Programmierung

4 Hardware, Kernel
5 Schluss

Warum dieser Vortrag?

- Schamlose Werbung
 - Jörg Faschingbauer
 - jf@faschingbauer.co.at
 - www.faschingbauer.co.at
- Es ist alles sehr kompliziert (©Fred Sinowatz)
... wenn man alles vermischt
- CAN-Bus wird *nicht* mit Einfachheit in Verbindung gebracht
- Linux bringt Einfachheit rein
 - Netzwerkprogrammierung → Allgemeinwissen
 - “SocketCAN”: gespendet von Volkswagen
 - Keine Bindung der Applikation an Hardware
 - Freie Wahl der Programmiersprache → natürlich Python
- Spezielles Goodie: Virtueller CAN-Bus
→ Entwickeln und Testen am PC, ganz ohne Hardware

Halbherzig: Geschichte aus Wikipedia

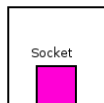
jjj hier noch was her

Netzwerkprogrammierung: Datagrammverkehr

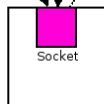
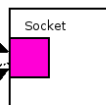
Paketvermittlung → Datagramme

- Am Internet: User Datagram Protokoll (UDP)
- Keine Point-to-Point Verbindung wie TCP
- Broadcasts möglich
- Paketgrenzen

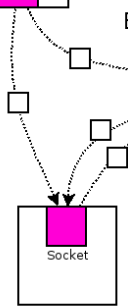
Eine Applikation



Eine andere Applikation

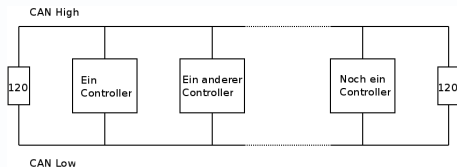


Noch eine Applikation



Und CAN?

- Sehr kleine Pakete (bis 8 Bytes Nutzdaten)
- Bus-Arbitrierung, Priorität und Applikationszuordnung anhand von "Paket-IDs"
- Keine Adressen → nur Broadcasts
- ⇒ sieht aus wie ein Netzwerk, ist ein Netzwerk



Ein Netzwerkpaket ...

- Definiert als C-Struct
- Host-Byteorder
- Python: `struct.pack()`, `struct.unpack()`

```
#include <linux/can.h>
```

```
struct can_frame {  
    canid_t can_id; /* 32 bit CAN_ID + EFF/RTR/ERR flags */  
    __u8     can_dlc; /* frame payload length */  
    __u8     data[8] __attribute__((aligned(8)));  
};
```


Übersicht

1 Basics

2 CAN Interfaces
3 Programmierung

4 Hardware, Kernel
5 Schluss

Das CAN-Interface: Konfiguration

CAN ist ein Netzwerk ...

```
# ip link show
... alle interfaces hier ...
# ip link show can0
3: can0: <NOARP,ECHO> mtu 16 qdisc noop state DOWN mode DEFAULT
    link/can
# ip link set can0 type can help
# ip link set can0 bitrate 500000
# ip link set can0 bitrate 500000 listen-only on
# ip link set can0 up
```

CAN Utils (1)

Nette kleine Utilities ...

- `git@gitorious.org:linux-can/can-utils.git`
- Überbleibsel des SocketCAN Projektes (→ “CAN als Netzwerk”)
- `cansend`: Frame senden
- `candump`: Aufzeichnen aus 1.. Interfaces
- `canplayer`: Replay aus Logfile
- ... und einige andere
- → Testen der Konfiguration
- ... und mit Fantasie mehr

CAN Utils (2)

Frames generieren:

```
$ cansend can0 42#deadbeef
$ cangen -D deadbeef -L 4 can0
$ cangen -D deadbeef -L 4 -I 42 can0
$ cangen -D i -I 42 -L 8 -g 100 -p 100 can0
```

Frames mitschnüffeln:

```
$ candump can0
$ candump can0 can1 ...
```

CAN Utils (3)

Record, Replay:

```
$ candump -L can0 > can0.log  
$ canplayer < ./can0.log
```

Wechseln des Interfaces:

```
$ canplayer can1=can0 < ./can0.log
```

vcan: Virtuelles CAN-Interface

Problem:

- CAN-Programmierung braucht Hardware
- ... mindestens zwei Teilnehmer (oder Loopback über Controller)
- → Programmieren und Testen so schwer wie möglich

Lösung:

```
# modprobe vcan  
# ip link add dev mein-test-can type vcan  
# canplayer mein-test-can=can0 < ./can0.log
```

Fantasie:

- Programmieren und Testen zuhause am PC
- Continuous Integration
- ...

Übersicht

1 Basics

2 CAN Interfaces

3 **Programmierung**

4 Hardware, Kernel

5 Schluss

CAN Programmierung

CAN geht über Sockets ...

- Paketvermittlung → ähnlich wie UDP
- Neue *Protocol Family*: PF_CAN
- Keine Adressen → Binden per “Interface Index”
- Pakete (“Frames”) von fixer Größe

CAN in C — Socket, “Interface Index”

Documentation/networking/can.txt

```
int s;
struct sockaddr_can addr;
struct ifreq ifr;

s = socket(PF_CAN, SOCK_RAW, CAN_RAW);

strcpy(ifr.ifr_name, "can0");
ioctl(s, SIOCGIFINDEX, &ifr);

addr.can_family = AF_CAN;
addr.can_ifindex = ifr.ifr_ifindex;

bind(s, (struct sockaddr *)&addr, sizeof(addr));
```

CAN in Python — Socket, “Interface Index”

```
can_socket = socket.socket(  
    socket.PF_CAN, socket.SOCK_RAW, socket.CAN_RAW)  
can_socket.bind(('can0',))
```

CAN in C — Frames (1)

Kernel liefert (und akzeptiert) “Netzwerkpakete” fixer Größe → CAN Frames

```
#include <linux/can.h>

struct can_frame {
    canid_t can_id; /* 32 bit CAN_ID + EFF/RTR/ERR flags */
    __u8 can_dlc; /* frame payload length */
    __u8 data[8] __attribute__((aligned(8)));
};
```

CAN in C — Frames (2)

```
struct can_frame frame;  
read(s, &frame, sizeof(struct can_frame));  
  
/* do something with frame */  
  
write(s, &frame, sizeof(struct can_frame));
```

CAN in Python — Frames

```
struct can_frame {
    canid_t can_id; /* 32 bit CAN_ID + EFF/RTR/ERR flags */
    __u8    can_dlc; /* frame payload length */
    __u8    data[8] __attribute__((aligned(8)));
};
```

```
frame_layout = "=IB3x8s"
frame_size = struct.calcsize(frame_layout)
frame = can_socket.recv(frame_size)
can_id, can_dlc, data = struct.unpack(frame_layout, frame)

/* do something with frame */

frame = struct.pack(frame_layout, can_id+1, len(data), data)
can_socket.send(frame)
```

Was gibts noch zu sagen?

- Interface Index 0 (Python: leerer Interface-Name, '') → *alle* Interfaces
- Alternative System Calls: `recvfrom()`, `sendto()`, wenn man am Interface interessiert ist
- Der Rest ist Unix
 - Filedescriptoren
 - Event Loops
 - → Client-Server Techniken, die altbekannt sind
- Realtime ... hat nix damit zu tun
 - Kann man machen, wenn man will/muss
 - *Vorsicht*: soll man nicht, wenn man nicht muss

Übersicht

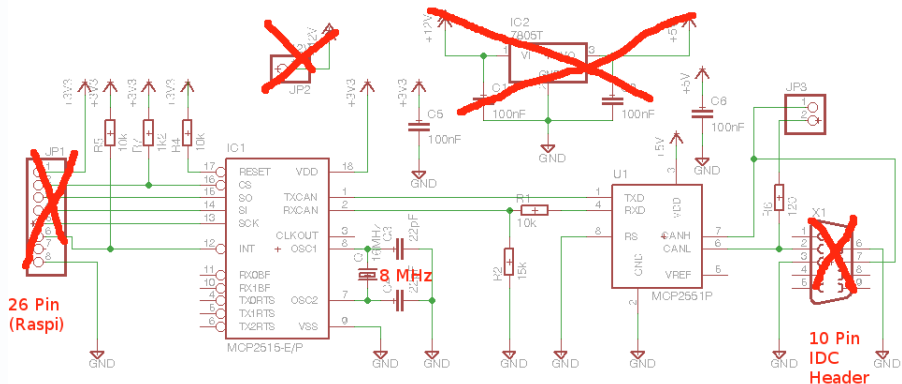
1 Basics

2 CAN Interfaces
3 Programmierung

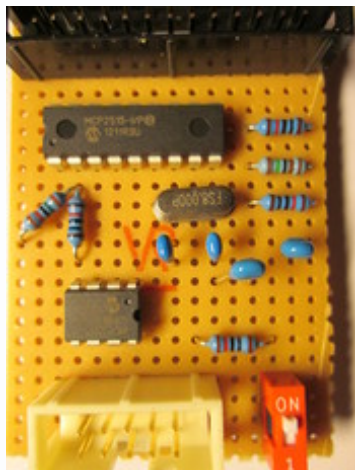
4 **Hardware, Kernel**
5 Schluss

MCP2515: Schaltplan

<http://lrxpps.de/can2udpe/openwrt/>



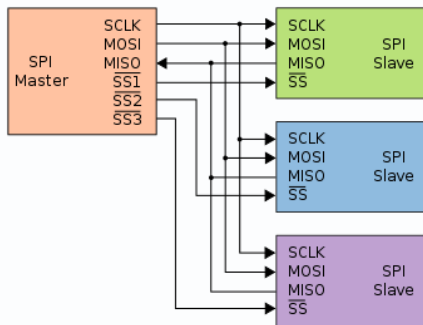
MCP2515: Löterei



MCP2515: SPI

- SPI: asymmetrisch (Master/Slave)
 - MOSI, MISO, SCLK: SPI an sich
 - “Chip Select” (“Slave Select”)
- Benachrichtigung an Master
 - Interrupt
 - Raspi: beliebiger GPIO

Serial Peripheral Interface — SPI



MCP2515: Raspberry

- Broadcom BCM2835
 - 2 SPI Master, davon einer über Header P1 erreichbar
 - 2 Chip-Select → CE0 (“Chip Enable”)
 - Haufenweise IO → GPIO25
- Stromversorgung
 - 3.3V für MCP2515
 - 5V für Transceiver

Raspberry P1 Pinout



Arbeiten am Kernel

- **“Board File”**: Verdrahtung der Software mit Hardware
- C Code
- Hauptsächlich Strukturen in Analogie zur Hardware
- Raspberry Board File: `arch/arm/mach-bcm2708/bcm2708.c`
- MCP2515 Driver: `drivers/net/can/mcp251x.c`
- <http://github.com/jfasch/linux/tree/rpi-mcp2515>

Übersicht

- 1 Basics
- 2 CAN Interfaces
- 3 Programmierung
- 4 Hardware, Kernel
- 5 **Schluss**

Weiterführendes

- SocketCAN: <http://en.wikipedia.org/wiki/SocketCAN>
- CAN-Utills: <https://gitorious.org/linux-can/can-utils>
- Kernel Doc: `Documentation/networking/can.txt` (im Kernel Source)
- Python-CAN: <http://python-can.readthedocs.org>
- OBD II: <http://de.wikipedia.org/wiki/On-Board-Diagnose>

Viel Spass

