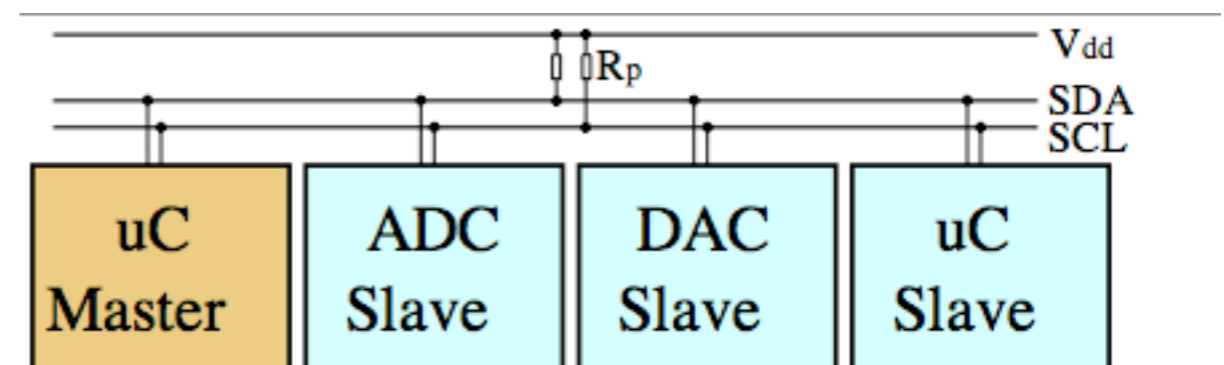


# I2C Low Level in der Praxis

Nikolaus Schaller, Golden Delicious Computers  
Open Hard&Software Workshop  
München 4. 12. 2011

# I2C wozu?

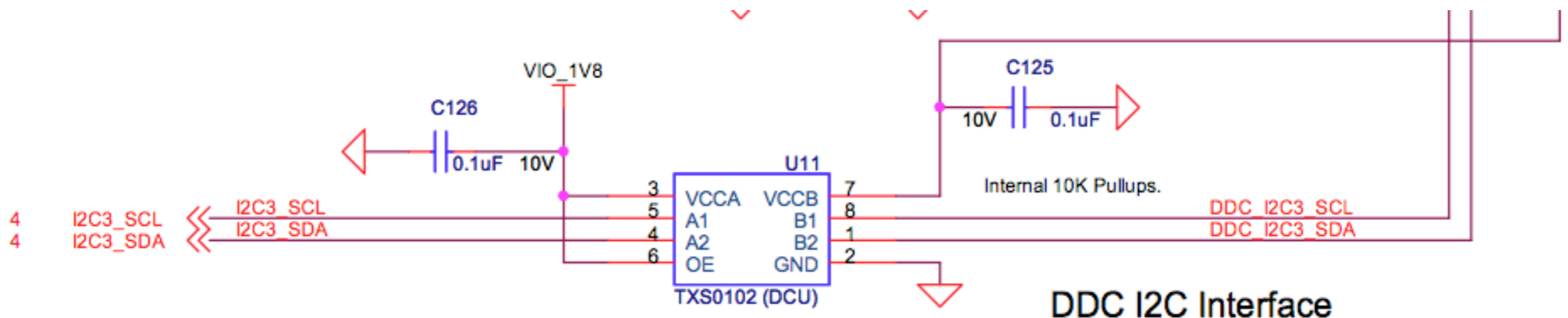
- viele kleine Chips mit wenigen Leitungen über Bus an CPU anschließen
- Sensoren, ADC, Timer, EEPROM, usw.
- andere Namen: 2-wire, SMBus
- Beispiel: TSC2007 (Touch Screen controller)





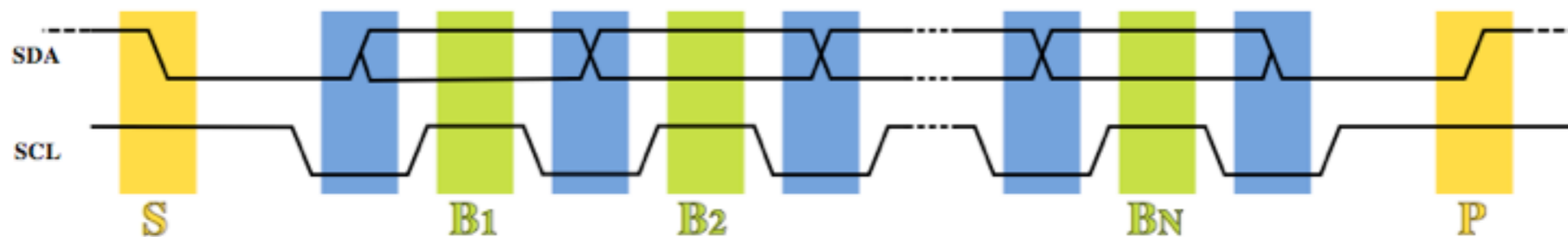
# I2C-Signale

- 2 Leitungen SDA (Data), SCL (Clock)
- Open Collector
- Meist 3.3V, viele auch 1.8V oder 5V (Bereich beachten)
- Level Shifter (Ausschnitt Schaltplan BeagleBoard)



# Signal-Zustände

- 2 Leitungen - 4 Zustände
- Wichtig sind auch Zustandswechsel!
- Wechsel von SDA während SCL=H  
=> Framing (Start, Stop)
- Bits werden nach SCL H->L geschrieben und vom Empfänger bei SCL L->H gelesen
- Jeder Teilnehmer kann SCL=L verlängern (Daten ausbremsen)



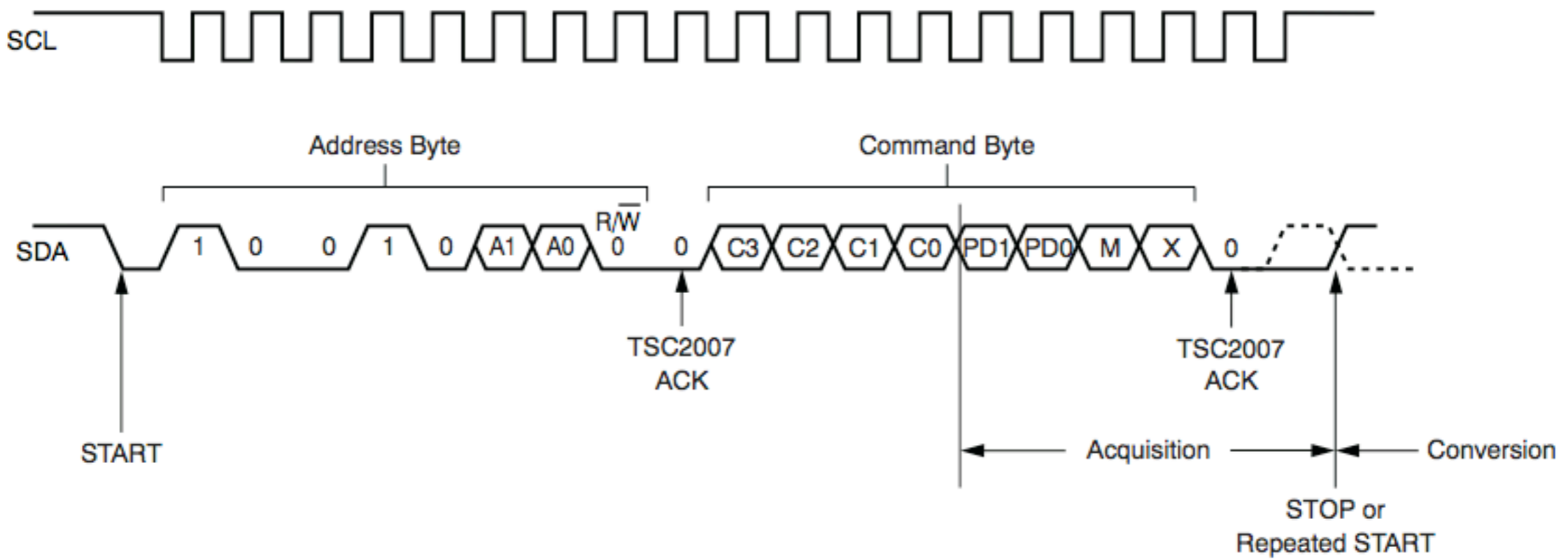
# Datenrate

- ist durch den langsamsten Teilnehmer begrenzt
- abhängig von Pull-Up, Leitungskapazitäten, Leitungslänge (Wellenwiderstand)
- aber auch von Chips (haben oft langsamen, internen Clock-Generator für State-Machine)
- typ. 100 kHz, 400 kHz, >1 MHz

# Signalisierung

- START Condition
- 8 Datenbits
- R/W-Bit (0 = Write, 1 = Read)
- Acknowledge-Bit (Master sendet H, Empfänger L)
- STOP condition

# Beispiel



**Figure 31. Complete I<sup>2</sup>C Serial Write Transmission**

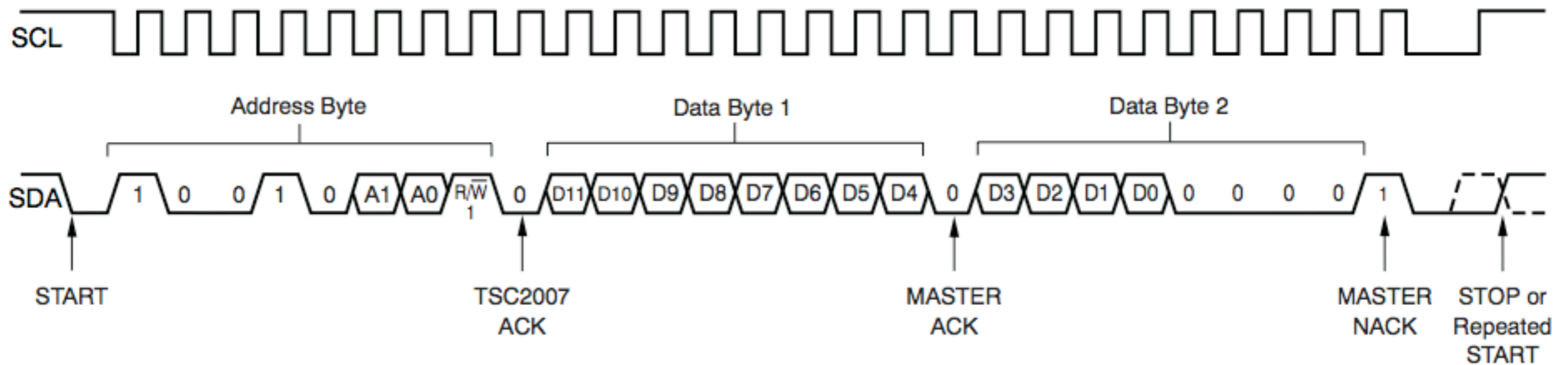
# Adressierung

- jedes Device hat eine Adresse
- das erste Byte das der Master schreibt ist immer die Adresse
- Bit 0 definiert ob es ein Schreib- oder Lesebefehl wird
- weitere Bytes werden geschrieben oder gelesen (Master treibt beim Lesen nur noch den Clock)
- Master beendet den Vorgang (ggf. vorzeitig bei variabler Länge des Datenblocks!) durch eine STOP-Condition

# Weitere Bytes

- Nach der Adresse folgen Bytes die chipspezifisch interpretiert werden
- z.B. Befehle, Subadressen, Adressen von Registern
- und Daten
- typisch: Chipadresse + Registeradresse + zu schreibendes Byte

# Lesen



**Figure 32. Complete I<sup>2</sup>C Serial Read Transmission**

# Bit-Bang vs. Controller

- Bitbang: CPU nimmt zwei GPIO-Leitungen und schreibt/liest bzw. schaltet sie hoch/niedrig und führt alle Schieberegisteroperationen durch
- Controller: auf dem SoC sitzt ein Controller dem man Bytes in einen FIFO steckt/abholt

# API

- Initialisierung: Clockfrequenz festlegen
- Low Level: Byte senden/lesen mit ACK/NACK/STOP
- High Level: Block schreiben/lesen (ggf. aneinandergereiht schreiben+lesen)

# U-Boot-Befehle

- `i2c dev 1 -- Bus 2 auswählen (OMAP: I2C2)`
- `i2c probe -- gefundene Devices anzeigen`

# U-Boot-API

```
i2c_set_bus_num(1); // I2C2
printf("TSC2007:      %s\n", !i2c_probe(0x48)?"found":"-");
```

```
int tsc2007_cmd(int cmd)
{ // send command
    unsigned char buf[16];
    buf[0]=cmd;
    if (i2c_write(TSC2007_ADDRESS, cmd, 1, buf, 0)) // write 1 byte command
    {
        printf ("Error writing the TSC.\n");
        return 1;
    }
    return 0;
}
```

# Wie funktioniert “Probe“?

- Trick:
- man schreibt nur die Adresse,
- wartet den ACK ab (mit Timeout) und
- beendet den Vorgang durch einen STOP
- d.h. es wird weder geschrieben noch gelesen

# Kernel-Treiber

- Registrieren im Boardfile
- Treiber-Probing
- Daten übertragen

# Registrieren im Boardfile\*

```
static struct i2c_board_info __initdata gta04_i2c2_boardinfo[] = {  
#if defined(CONFIG_TOUCHSCREEN_TSC2007) || defined  
(CONFIG_TOUCHSCREEN_TSC2007_MODULE)  
{  
    I2C_BOARD_INFO("tsc2007", 0x48),  
    .type          = "tsc2007",  
    .platform_data = &tsc2007_info,  
    .irq          = OMAP_GPIO_IRQ(TS_PENIRQ_GPIO),  
},  
#endif  
};
```

Busadresse

```
static int __init gta04_i2c_init(void)  
{  
    omap_register_i2c_bus(1, 2600, gta04_i2c1_boardinfo,  
        ARRAY_SIZE(gta04_i2c1_boardinfo));  
    omap_register_i2c_bus(2, 400, gta04_i2c2_boardinfo,  
        ARRAY_SIZE(gta04_i2c2_boardinfo));  
    omap_register_i2c_bus(3, 100, gta04_i2c3_boardinfo,  
        ARRAY_SIZE(gta04_i2c3_boardinfo));  
    return 0;  
}
```

Busnummer (I2C2)

Clock in kHz

# Treiber-Probing\*

```
static int __devinit tsc2007_probe(struct i2c_client *client,
                                  const struct i2c_device_id *id)
{
    struct tsc2007 *ts;
    struct tsc2007_platform_data *pdata = pdata = client->dev.platform_data;
    struct input_dev *input_dev;
    int err;

    if (!pdata) {
        dev_err(&client->dev, "platform data is required!\n");
        return -EINVAL;
    }

    if (!i2c_check_functionality(client->adapter,
                                 I2C_FUNC_SMBUS_READ_WORD_DATA))
        return -EIO;
    ...

    static struct i2c_device_id tsc2007_idtable[] = {
        { "tsc2007", 0 },
        { }
    };

    MODULE_DEVICE_TABLE(i2c, tsc2007_idtable);

    static struct i2c_driver tsc2007_driver = {
        .driver = {
            .owner    = THIS_MODULE,
            .name     = "tsc2007"
        },
        .id_table = tsc2007_idtable,
        .probe     = tsc2007_probe,
        .remove    = __devexit_p(tsc2007_remove),
    };
};
```

# Daten lesen

```
/* i2c/smbus access */
static inline int tsc2007_xfer(struct tsc2007 *ts, u8 cmd)
{
    s32 data;
    u16 val;

    data = i2c_smbus_read_word_data(ts->client, cmd);
    if (data < 0) {
        dev_err(&ts->client->dev, "i2c io error: %d\n", data);
        return data;
    }

    /* The protocol and raw data format from i2c interface:
     * S Addr Wr [A] Comm [A] S Addr Rd [A] [DataLow] A [DataHigh] NA P
     * Where DataLow has [D11-D4], DataHigh has [D3-D0 << 4 | Dummy 4bit].
     */
    val = swab16(data) >> 4;

    dev_dbg(&ts->client->dev, "data: 0x%x, val: 0x%x\n", data, val);

    return val;
}
```

Linux hat spezielle "SMBus"-Befehle:  
1 Byte CMD senden, 1 Word (2 Byte) lesen

# Achtung!

- SMBus ist nur eine Teilmenge aller I2C-Möglichkeiten; manche Chips können nur über volles I2C-API gesteuert werden
- I2C-Kommunikation nicht in Interrupt-Handler möglich => struct work

# Linux User-Space

- ähnlich wie U-Boot
- Debian: apt-get install i2c-tools
- i2cdetect
- i2cdump
- i2cget
- i2cset