Gerätetreiber unter Linux

Konstantin Veltmann

22.Jan.2026

Der Quelltext dieser Arbeit sowie Beispielcode und das vollständige Quellenverzeichnis sind unter https://git.veltko.de/Weckyy702/uc-ausarbeitung-linux-treiber° mit der GPL lizensiert aufzufinden.

Inhaltsverzeichnis

1. Einleitung 1
   1.1. Gerätezugriff per Dateisystem 1
      1.1.1. Device-Dateien 1
      1.1.2. Zugriff über Sysfs 2
2. Beschreibung hardwarespezifischer Schnittstellen unter Linux 3
   2.1. I²C 3
   2.2. GPIO 3
   2.3. ADC 5
3. Design einer Hardwareschnittstelle für AT91SAM7-Timer 6
   3.1. Features 6
      3.1.1. Capture-Modus 6
      3.1.2. Waveform-Modus 6
   3.2. Umsetzung 6
4. Scheduling bei geteilten Bussystemen 8
   4.1. Lösungsansatz 8
Bibliografie 10

1. Einleitung

Damit Computersysteme mit ihrer Umwelt interagieren können, ist die Kommunikation mit externen Sensoren und Aktoren erforderlich. Eine der Aufgaben eines Betriebssystems besteht darin, gemeinsame Schnittstellen für verschiedene Geräte bereitzustellen[1]. Unter Linux werden hierfür sogenannte *device files*[2] verwendet, die den Zugriff auf diese Geräte über klassische Dateioperationen ermöglichen.

1.1. Gerätezugriff per Dateisystem

Nach der UNIX-Philosophie „Everything is a file“[3] melden Gerätetreiber spezielle Dateien im virtuellen Dateisystem an (in der Regel im Verzeichnis `/dev` oder `/sys`). Wird auf einer solchen Datei ein Systemaufruf wie `write()` ausgeführt, so wird eine Funktion im Kerneltreiber aufgerufen, die die ‚geschriebenen‘ Daten an das physische Gerät weiterleitet.

1.1.1. Device-Dateien

Folgender Beispielcode[4] zeigt die Kommunikation mit einem BME280-Sensor mithilfe des Userspace I²C-Treibers auf einem Raspberry Pi:

```
int main() {
  // Öffne die Device-Datei
  int driver = open("/dev/i2c-1");                    // (1)

  // Setze die Slave-Adresse für nachfolgende Transaktionen
  ioctl(driver, I2C_SLAVE, 0x76);                     // (2)

  // Schreibe die Adresse für das ID-Register
  uint8_t const write_buf[] = {0xd0};
  write(driver, write_buf, sizeof(write_buf));        // (3)

  // Lese aus dem ID-Register. NOTE: Das ist eine separate I²C-Transaktion!
  uint8_t id;
  read(driver, &id, 1);                               // (4)

  printf("ID: %"PRIu8, id);
}
```

Der Beispielcode zeigt die vier typischen Datei-Operationen bei der Arbeit mit Device-Dateien:

1. Wie jede andere Datei muss die Device-Datei geöffnet werden. Der Treiber richtet dabei die notwendigen Verwaltungsstrukturen für die weitere Nutzung durch die Anwendung ein.
2. Der `ioctl`-Systemaufruf dient dazu, Treibereinstellungen zu ändern oder Operationen auszuführen, die nicht über `read` oder `write` abgebildet werden können. Die Flags und Parameter für `ioctl` sind treiberabhängig.
3. Der `write`-Systemaufruf sendet eine schreibende Transaktion auf den I²C-Bus. In der Regel wird `write` verwendet, um Daten auf Busse zu schreiben oder Ausgänge zu schalten.
4. `read` führt eine lesende Transaktion auf dem I²C-Bus aus. `read` wird typischerweise verwendet, um Gerätedaten auszulesen oder von Bussen zu empfangen.

Typischerweise stellt der Kernel eine Begleitbibliothek wie `libi2c` für I²C-Operationen bereit, um versionsabhängige Unterschiede zu abstrahieren. Diese Bibliotheken werden hier nicht näher beschrieben, stellen jedoch die empfohlene Schnittstelle dar.

1.1.2. Zugriff über Sysfs

Einige Gerätetreiber registrieren keine Dateien in `/dev`, sondern werden über Dateien im virtuellen Dateisystem unter `/sys` kontrolliert. Die Konvention dafür ist, dass für Eingabe-/Ausgabe-Operationen mit Geräten die Dateien in `/dev` verwendet werden, während `/sys` für strukturierte Zugriffe und Konfiguration verwendet wird[5].

Geräte-Dateien in `/sys` haben eine String-basierte Schnittstelle, es werden also menschenlesbare Werte in verschiedenen Dateien geschrieben. Das macht die Interaktion mit `/sys` -Dateien in der Shell attraktiv.

Folgender Shell-Code liest die momentane Batteriespannung meines Laptops aus.

```
# ADC-Dateien finden, haben i.d.R voltage im Namen
find /sys -iname "*voltage*"
BAT='/sys/devices/[...]/BAT1/voltage_now' # Pfad gekürzt
cat $BAT
# > 12832000 [µV]
```

Folgender Shell-Code stellt auf einem Banana Pi R3° die Drehgeschwindigkeit des CPU-Kühlers auf 40%:

```
echo 40 > /sys/devices/platform/pwm-fan/hwmon/hwmon1/pwm1
```

2. Beschreibung hardwarespezifischer Schnittstellen unter Linux

2.1. I²C

Wie in der Einleitung beschrieben stellt der I²C-Treiber device-Dateien unter `/dev/i2c-*` bereit.

- Implementiert in `drivers/i2c/i2c-dev.c`°
- Offizielle Dokumentation°

Implementierte Systemaufrufe:

Syscall	Funktion
`ioctl(<file>, I2C_SLAVE, <addr>)`	Setzt die Slave-Adresse für alle folgenden I²C-Transaktionen
`write(<file>, <buf>, <len>)`	Sendet die Daten aus `buf` in einer einzelnen I²C-Schreib-Operation an die gesetzte Slave-Adresse
`read(<file>, <buf>, <len>)`	Liest `len` Bytes vom ausgewählten Slave in `buf`
`ioctl(<file>, I2C_RDWR, <msgset>)`	Sendet mehrere Schreibe- und Lese-Operationen an den ausgewählten Slave in einer Transaktion ohne Stop-Conditions

Tabelle 1: Systemaufrufe des I²C-Treibers

In modernen Computersystemen wird der mit I²C kompatible SMBus verwendet. Daher stellt der Treiber noch weitere `ioctls` bereit, die hier jedoch nicht besprochen werden.

2.2. GPIO

Der GPIO-Treiber stellt zwei Schnittstellen bereit, eine unter `/dev` und eine veraltete unter `/sys`. Hier wird die aktuelle empfohlene Variante beschrieben.

- Implementiert in `drivers/gpio/gpiolib-cdev.c`°
- Offizielle Dokumentation°.

Implementierte Systemaufrufe:

Syscall	Funktion
`ioctl(<file>, GPIO_GET_CHIPINFO_IOCTL, <chip_info>)`	Informationen über einen Gpio-Chip holen
`ioctl(<file>, GPIO_GET_LINEINFO_UNWATCH_IOCTL , <line_offset>)`	Stoppt das Beobachten

Syscall	Funktion
	eines GPIO-Pins
`ioctl(<file>, GPIO_V2_GET_LINEINFO_IOCTL, <line_info>)`	Beschafft Informationen über einen spezifischen GPIO-Pin
`ioctl(<file>, GPIO_V2_GET_LINEINFO_WATCH_IOCTL, <line_info>)`	Beschafft Informationen über einen GPIO-Pin und macht nachfolgende Änderungen über `read` verfügbar
`ioctl(<file>, GPIO_V2_GET_LINE_IOCTL, <line_request>)`	Reserviert und konfiguriert einen GPIO-Pin für das aufrufende Programm
`ioctl(<file>, GPIO_V2_LINE_SET_CONFIG_IOCTL, <line_config>)`	Setzt Attribute für einen Pin, zum Beispiel Input/Output oder active LOW/HIGH
`ioctl(<file>, GPIO_V2_LINE_GET_VALUES_IOCTL, <line_values>)`	Liest Werte von mehreren Eingangs-Pins
`ioctl(<file>, GPIO_V2_LINE_SET_VALUES_IOCTL, <line_values>)`	Setzt/Cleared mehrere Ausgangs-Pins

Tabelle 2: Systemaufrufe des GPIO-Treibers

2.3. ADC

ADCs werden in Linux nicht direkt als eigene Geräteklasse verwaltet, sondern sind in der Regel als *Hardware Monitoring* (Überwachung) oder *Industrial I/O* (iio) gelistet.

Als Beispiel wird hier der Kernel-eigene Treiber für den ADC des Cirrus Logic EP93xx SoC[6] genutzt. Dabei wird für jeden der ADC-Pins ein eigener Eintrag unter `/sys/bus/iio/devices/iio:device<N>/` angelegt, wobei N die Geräte-ID ist:

Sysfs-Eintrag	Name des gesampleten Pins
in_voltage0_raw	`Y-`
in_voltage1_raw	`sX+`
in_voltage2_raw	`sX-`
in_voltage3_raw	`sY+`
in_voltage4_raw	`sY-`
in_voltage5_raw	`X+`
in_voltage6_raw	`X-`
in_voltage7_raw	`Y+`

Tabelle 3: Sysfs-Einträge des ADC-Treibers

Das Auslesen einer dieser Datein führt synchron eine ADC-Umwandlung durch.

Aus der Dokumentation anderer ADC-Treiber[7] geht hervor, dass der `in_voltageX_raw` -Wert der unskalierte Bitwert des ADC ist. Die Datei `/sys/[...]/in_voltage_scale` beinhaltet den Umrechnungswert vom Rohwert zu Millivolt. Manche Treiber stellen zusätzlich die Datei `/sys/[...]/in_voltage_offset` bereit, die einen konstanten Fehlerwert enthält. Die vollständige Umrechnung ist dann:

$$U_{[\mathrm{mV}]} = (\mathrm{voltage\_raw} \cdot \mathrm{voltage\_scale}) + \mathrm{voltage\_offset}$$

3. Design einer Hardwareschnittstelle für AT91SAM7-Timer

Der AT91SAM7-Mikrocontroller[8] stellt das *Timer Counter Peripheral* bereit; Drei unabhängige 16-bit Zähler, Kanäle genannt, mit einstellbaren Taktgeschwindigkeiten, Überlaufgrenzen und *Triggern*.

3.1. Features

Jeder Kanal kann sich in einem der folgenden Modi befinden:
- *Capture* zum Festhalten von Zeitpunkten, zu denen Eingänge geschaltet wurden
- *Waveform* zum Erzeugen von einstellbaren Rechtecksignalen

Außerdem hat jeder Kanal drei Eingangssignale `XC0-2`, zwei Ausgangssignale `A/B` und kann einen von fünf Vorteilern wählen.

3.1.1. Capture-Modus

Im *Capture*-Modus zählt der Zähler kontinuierlich und es wir bei einem konfigurierbaren *Event* (eine Flanke auf `TIOA` oder `TIOB`) der Zählerstand in eins der Register geschrieben.

Dieser Modus ist unter anderem für die Bestimmung von Frequenz, Pulszeit und Pahsenbestimmung eins oder mehrerer anliegender Signale gedacht.

3.1.2. Waveform-Modus

Dieser Modus ist für die Erzeugung von Rechtecksignalen gedacht. Es gibt vier Untermodi:

Modus	Zählrichtung	Verhalten wenn $= \mathrm{RC}$
`00`	Hoch	Nichts, nur durch Überlauf zurückgesetzt
`10`	Hoch	Zurücksetzen auf 0
`01`	Hoch, dann Runter	Nichts, Richtungswechsel wenn $= 0$ oder $= 0\mathrm{xFFFF}$
`11`	Hoch, dann Runter	Richtungswechsel

Tabelle 4: Wellenmodi im Waveform-Modus

Außerdem wird der Zählerwert immer mit den Werten in den Registern `RA/RB/RC` auf Gleichheit verglichen. Die daraus entstehenden Trigger-Signale können dann die Ausganspins `A/B` jeweils entweder einschalten, ausschalten oder umschalten.

3.2. Umsetzung

Die API ist an der Struktur der GPIO-API orientiert.

Jeder Kanal muss mit `REQ_CHANNEL` vom Kernel angefragt werden, damit ein Kanal von genau einem Prozess verwaltet wird. Mithilfe der `SET_MODE_CAPTURE` und `SET_MODE_WAVE` `ioctl`s wird der Kanal in den jeweiligen Modus versetzt und konfiguriert. Der `TIMER_START` -Befehl startet einen einzelnen Kanal. Wenn der aufrufende Prozess alle Kanäle kontrolliert, kann `TIMER_START` auf dem Timer selbst aufgerufen werden, was das SYNC-Signal für alle Kanäle setzt.

Folgend eine Beispielanwendung:

```
int main() {
  int timer_fd = open("/dev/timer0");

  int ch0 = ioctl(timer_fd, REQ_CHANNEL_IOCTL, 0);
  int some_free_channel = ioctl(timer_fd, REQ_CHANNEL_IOCTL, -1);

  struct capture_config capture_config = {
    .clock = CLOCK_1,   // = TIMER_CLOCK1
    .clock_burst = CLOCK_BURST_NONE, // Oder CLOCK_BURST_TIOA0/1/2
    .clock_invert = false,
    .a_edge = EDGE_RISING,
    .b_edge = EDGE_NONE,
    .external_trigger = EXT_TRIGGER_A,
    .interrupt_on = INT_LDRA | INT_LDRB | INT_OVF, // Aktivierte interrupts
    .compare = -1, //Deaktiviert CPCTRG, >0 aktiviert CPCTRG
  };

  ioctl(ch0, SET_MODE_CAPTURE, &capture_config);

  struct wave_config wave_config = {
    .clock = CLOCK_TIOA2,   //-EINVAL wenn nicht verfügbar
    .clock_invert = true,
    .wave_mode = WAVE_MODE_UP_RC_TRIGGER, // WAVSEL = 10
    .ra = 100,
    .rb = 0x4000,
    .rc = 0x9fff,
    .tioa = (struct mtio) {
      .a_mode = MTIO_MODE_SET,
      .b_mode = MTIO_MODE_CLEAR,
      .c_mode = MTIO_MODE_TOGGLE,
      .sw_mode = MTIO_MODE_NONE,
    }
    .tiob = (struct mtio) {0}, // TIOB ist deaktiviert
  };
  ioctl(some_free_channel, SET_MODE_WAVE, &wave_config);

  //Würde mit dem SYNC-Signal alle Kanäle starten,
  //allerdings hat dieser Prozess nicht alle Kanäle angefragt.
  //Der Aufruf würde also fehlschlagen
  //ioctl(timer_fd, TIMER_START);

  ioctl(ch0, TIMER_START); //Setzt SWTRG
  struct capture_event capture_event;
  // Blockiert bis mindestens eins der Signale in interrupt_on ausgelöst
wurde
  read(ch0, &capture_event, sizeof(capture_event));
}
```

4. Scheduling bei geteilten Bussystemen

Angenommen man habe ein smartes Thermostat mit folgenden Sensoren und Aktoren, alle angeschlossen über einen geteilten I²C-Bus:

- OLED-Display
- BME280 Temperatur- und Luftfeuchtigkeitsmesser

Jedes der Geräte hat einen eigenen Treiber, der über die kerneleigenen I²C-Funktionen auf den Bus zugreift.

Da Displays typischerweise hohe Datenraten brauchen, verbringt der Display-Treiber viel Zeit auf dem Bus. Zudem sollen in regelmäßigen Abständen Temperatur und Luftfeuchtigkeit vom Sensor angefragt werden.

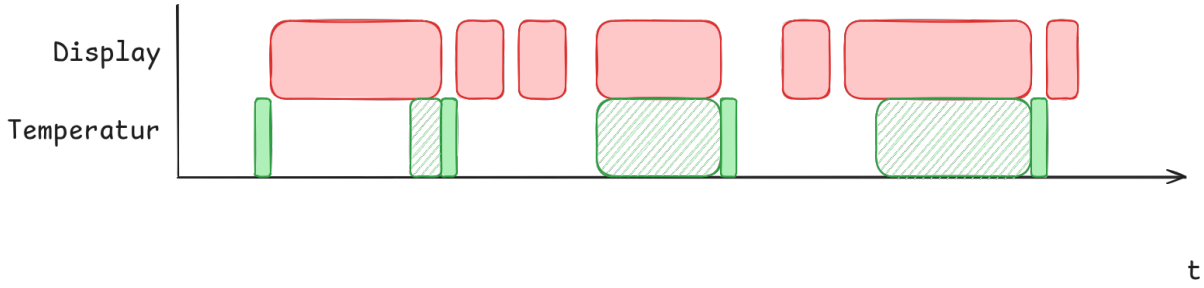

Abbildung 1: Gantt-Diagramm des smarten Thermostats

Wie in Abbildung 1 gezeigt, wird durch die häufigen Display-Übertragungen der Temperatur-Sensor „ausgehungert“ (schraffierter Hintergrund) und kann seine Daten nicht rechtzeitig übertragen.

Dieses Problem gehört zur Klasse der *Scheduling*-Aufgaben. Ein klassischer Lösungsansatz wird im nächsten Abschnitt besprochen

4.1. Lösungsansatz

Da hier eine geteilte Ressource (der Bus) **fair** zwischen mehreren Clients (den Treibern) verteilt werden soll, bietet sich ein *Scheduling*-Verfahren[9] an.

Fragt ein Client einen I²C-Transfer an, so wird er nicht direkt ausgeführt, sondern mit anderen ausstehenden Anfragen in einer *Queue* (dt. Warteschlange)[10] gespeichert. Nun kann der I²C-Scheduler die nächste anstehende Transaktion nach einem Scheduling-Verfahren wie dem *Completely Fair Scheduler*[11] aussuchen und durchführen, um Aushungern zu vermeiden.

Nachfolgend ist der Ablauf mit dem simplen *Round-Robin-Verfahren*[12] gezeigt, das **keine** Fairness garantiert:

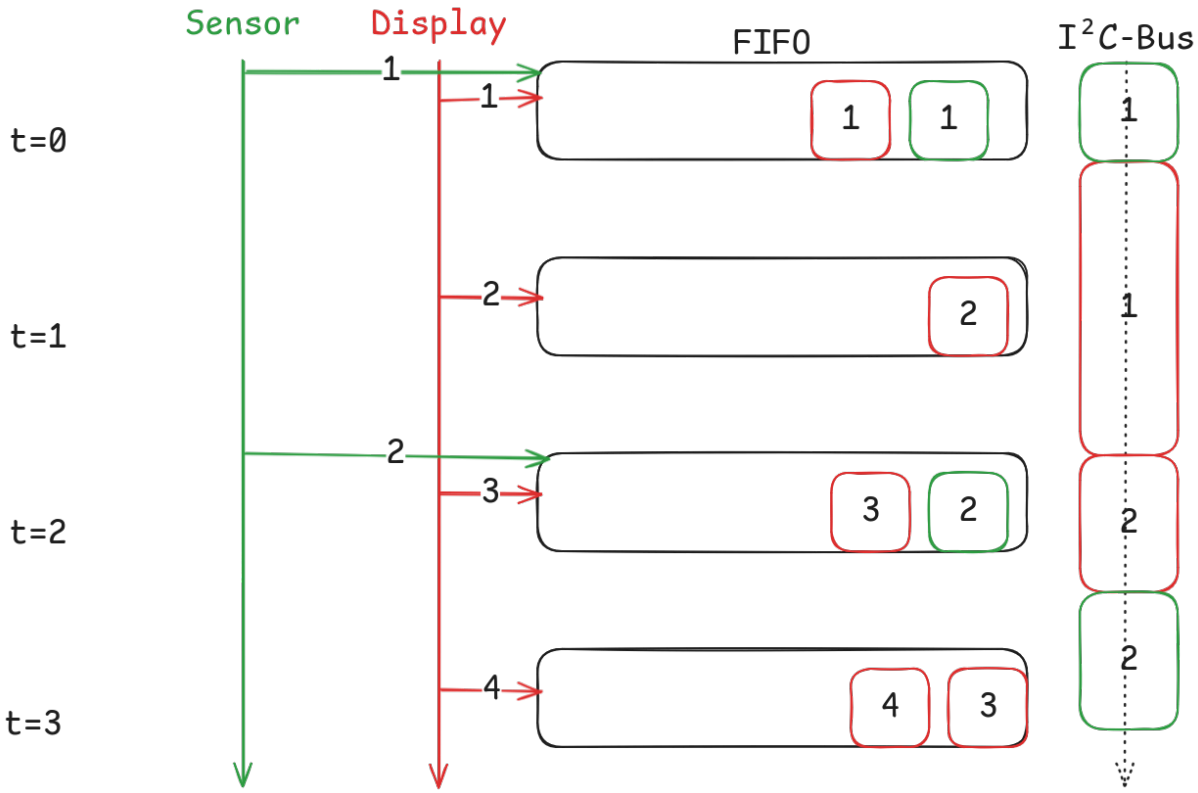

Abbildung 2: I²C-Scheduling mit Round Robin

Bibliografie

[1] J. Aiman, „ 5 Functions of an Operating System “. 2025.

[2] linux kernel contributors, „Character device drivers“.

[3] N. Brown, „Ghosts of Unix Past: a historical search for design patterns“. 2010.

[4] K. Veltmann, 2026.

[5] Y. Salem, „The Difference Between /dev and /sys/class“. 2024.

[6] the kernel development community, „Cirrus Logic EP93xx ADC driver“.

[7] „How to Use the SAMA5D2 ADC Under Linux®“. 2355 West Chandler Blvd. Chandler, Arizona, USA, 2019.

[8] „AT91SAM ARM-based Flash MCU“. 2355 West Chandler Blvd. Chandler, Arizona, USA, 2012.

[9] wikipedia contributors, „Prozess-Scheduler“. 2024.

[10] wikipedia contributors, „Queue (abstract data type)“. 2026.

[11] wikipedia contributors, „Completely Fair Scheduler“. 2025.

[12] wikipedia contributors, „Round Robin (Informatik)“. 2024.