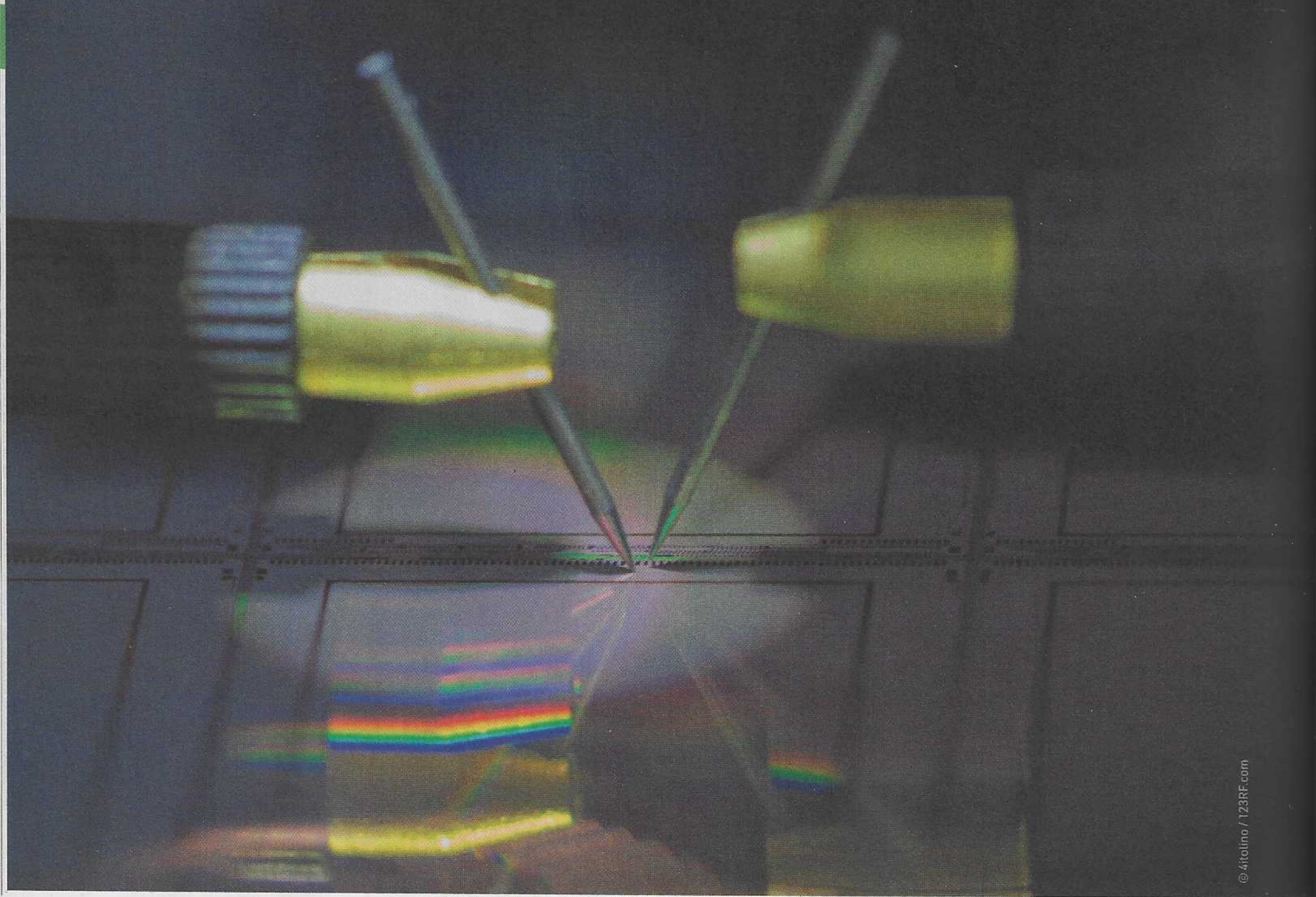


Kernel- und Treiberprogrammierung unter Linux – Folge 116

Präzise ausgelesen

Damit Anwender intuitiv auf Sensoren und Aktoren zugreifen können, müssen Entwickler einiges beachten. Der Treiber für ein virtuelles Industrial-IO-Gerät zeigt, wie es geht.

Eva-Katharina Kunst, Jürgen Quade

Seitdem das Industrial-IO-Subsystem vor neun Jahren produktiv im Linux-Kernel verankert wurde, hat es mächtig an Funktionalität, aber auch an Komplexität gewonnen. Der Benutzer liest damit Sensorwerte und steuert Aktoren über gut spezifizierte und dokumentierte IO-Kanäle. Die bildet das Subsystem für den intuitiven Zugriff praktischerweise auf virtuelle Dateien ab.

Dabei kann der Anwender sich darauf verlassen, dass diese Kanäle die angeforderten Daten in der definierten Auflösung liefern, beispielsweise in Millivolt oder in Hektopascal. Falls vom Treiber unterstützt, konfiguriert der Nutzer sich eine automatisierte Messwerterfassung, die unterschiedliche IO-Kanäle zeitgleich beackert und auf diese Weise konsistente Datensätze liefert.

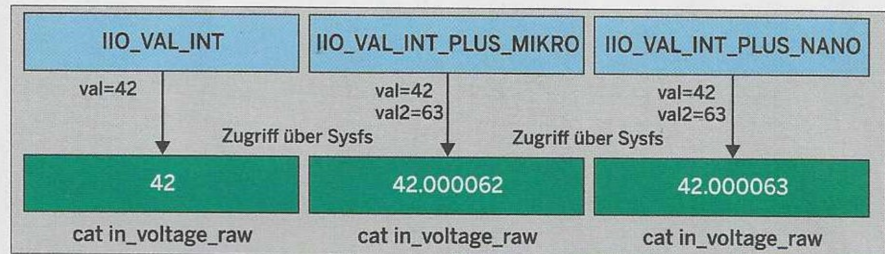
So praktisch derartige sauber definierte Interfaces zu IIO-Devices für den Nutzer auch sind, so unübersichtlich gerät zunächst für den Entwickler die Programmierung der benötigten Treibersoftware. Es gilt, den Treiber als solchen im System zu verankern, die unterstützten Geräte detailliert zu spezifizieren, Ressourcen

Die Autoren

Eva-Katharina Kunst ist schon seit den Anfängen von Linux Fan von Open Source. **Jürgen Quade**, Professor an der Hochschule Niederrhein, bietet auch für Unternehmen Schulungen zu den Themen Treiberprogrammierung und Embedded Linux an.

zu reservieren und zu guter Letzt die eigentlichen Zugriffsfunktionen auf die Hardware zu implementieren.

Dass mit parallelen Zugriffen zu rechnen ist, macht den Code nicht wirklich übersichtlicher. Auf der anderen Seite sorgt das Industrial-IO-Subsystem für die Ablaufsteuerung und die Interaktion mit dem Anwender und entlastet damit den Entwickler ganz entscheidend.



1 Fließkommawerte werden im Kernel auf Integer-Variablen abgebildet.

Tradition und Moderne

Der wesentliche Unterschied zwischen einem traditionellen Treiber und einem für IIO-Devices besteht darin, dass beim traditionellen Treiber der Zugriff über eine oder mehrere Gerätedateien stattfindet. Der Entwickler verankert den Treiber also im Kernel und schreibt geeignete `driver_read()`- und `driver_write()`-Funktionen. Dabei hat er alle Freiheiten.

Listing 1: Einfache Lesefunktion

```
static int foo_iio_read_raw(
  struct iio_dev *indio_dev,
  struct iio_chan_spec const
*channel,
  int *val, int *val2, long mask)
{
  if (channel->channel==0) {
    read_voltage_from_hardware(
val, val2 );
    return IIO_VAL_INT_PLUS_
MICRO;
  }
  return -EINVAL;
}
```

Insbesondere legt er die Repräsentation der mit den Applikationen ausgetauschten Daten fest.

Bei der Implementierung als IIO-Device hingegen gilt es, die Geräte respektive die ausgetauschten Daten detailliert in Datenstrukturen zu beschreiben. Statt der beiden Funktionen `driver_read()` und `driver_write()` gibt es in der einfachsten Variante dazu die Routinen `read_raw()` und `write_raw()`, die die Hardwaredaten über vordefinierte Variablen austauschen.

Solche Funktionen lassen sich für einfache Anwendungsfälle schnell implementieren. Ein typisches Beispiel zeigt Listing 1: Innerhalb der Funktion `read_raw()` findet der tatsächliche Zugriff auf die Hardware statt, wobei das Ergebnis in zwei Variablen (`val` und `val2`) landet, deren Adressen als Parameter übergeben wurden. Die erste Variable enthält typischerweise den ganzzahligen Anteil, die zweite Variable falls benötigt einen Nachkommaanteil, zum Beispiel einen ganzzahligen Nachkommawert in der Größenordnung Micro oder Nano.

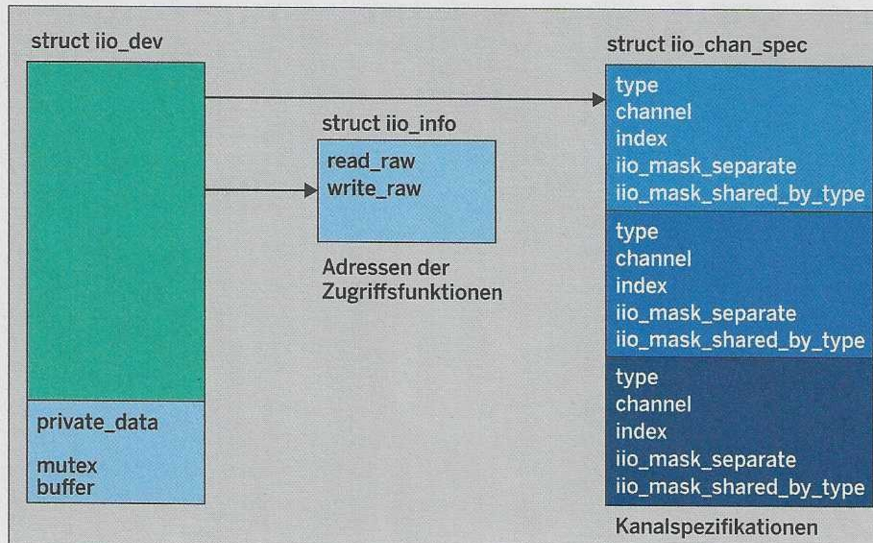
Die Funktion `read_raw()` gibt im Erfolgsfall den Typ der abgelegten Daten zurück. `IIO_VAL_INT` beispielsweise besagt, dass sich in `val` der Vorkommawert als Integer befindet. `IIO_VAL_INT_PLUS_NANO` signalisiert, dass in `val2` der Nachkommaanteil in einer Auflösung von 10^{-9} liegt **1**. Das IIO-Subsystem bereitet das Ergebnis beim Zugriff durch Nutzer entsprechend dieses Rückgabetyps als Fließkommawert auf.

Komplizierter wird es erst dadurch, dass es für alle Kanäle (und deren Attribute) eines IIO-Geräts nur diese eine Lesefunktion gibt. Folglich muss die Funktion sowohl die als Parameter übergebene Kanalnummer (Parameter `channel`) auswerten als auch die Anfrage nach einem potenziellen Attribut (Parameter `mask`). Abhängig davon gilt es anschließend, die Daten zu lesen, aufzubereiten und an den per `val` und `val2` übergebenen Adressen abzulegen (Listing 2). Zu guter Letzt gibt die Funktion zurück, welche Daten (mit oder ohne Nachkomma) sie an den Speicheradressen abgelegt hat.

Die Schreibfunktion `write_raw()` muss der Entwickler analog zu `read_raw()` implementieren. Sie bekommt die zu schreibende Größe ebenfalls über die beiden Variablen `val` und `val2` übergeben. Auch hier gilt es wieder, die Parameter `channel` und `mask` auszuwerten,

Listing 2: Kanalauswertung inklusive Attributtypen

```
static int foo_iio_read_raw(
  struct iio_dev *indio_dev,
  struct iio_chan_spec const
*channel,
  int *val, int *val2, long mask)
{
  switch (mask) {
  case IIO_CHAN_INFO_SCALE:
    *val = FOO_SCALE;
    return IIO_VAL_INT;
  case IIO_CHAN_INFO_OFFSET:
    *val = 0;
    *val2= FOO_OFFSET;
    return IIO_VAL_INT_PLUS_MICRO;
  case IIO_CHAN_INFO_PROCESSED:
    if (channel->channel==1) {
      *val = count*FOO_SCALE+
FOO_OFFSET;  // fake data
      *val2 = count*FOO_SCALE+
FOO_OFFSET; // fake data
      return IIO_VAL_INT_PLUS_
NANO;
    }
    return -EINVAL;
  }
  return -EINVAL;
}
```



2 IIO-Geräte werden über drei Datenstrukturen beschrieben.

falls der Treiber mehrere Kanäle und unterschiedliche Kanalattribute unterstützt.

Einmal Init, bitte!

Neben den Zugriffsfunktionen muss der Entwickler noch die Einbindung des IIO-Treibers in den Kernel durch Aufruf der Funktion `iio_device_register()` implementieren. Dieser Funktion übergibt er eine Datenstruktur des Typs `struct iio_dev` mit der Beschreibung des IIO-Geräts 2. Die gilt es zu initialisieren.

Eine zentrale Rolle spielen dabei die Unterstrukturen `struct iio_info` und `struct iio_chan_spec`. Erstere enthält die Adressen der eigentlichen Zugriffsfunktionen, die die Interaktion mit der Hardware realisieren. Die nicht minder wichtige Struktur `iio_chan_spec` definiert über ein Array die Kanäle, also die vom Treiber zur Verfügung gestellten Daten (Listing 3), gern auch als Makro (siehe Kasten Kanäle über Makros).

Neben dem Typ (zum Beispiel Spannung, Entfernung, Luftfeuchte) werden die Kanalnummer und – über ein Bitfeld – die unterstützten Kanalattribute angegeben. Dabei entscheidet der Entwickler, welche Attribute für jeden Kanal separat (Variable `info_mask_separate`) und welche für alle Kanäle gemeinsam (`info_mask_shared_by_type`) gelten. Zurzeit stehen 27 unterschiedliche Attribute zur Verfügung. Neben den Klassikern *Rohwert*, *aufbereiteter Wert*, *Offset* und *Skalierung* tummeln sich darunter auch Attribute wie *Phase*, *Hysterese* oder *Frequenz*.

Als Letztes steht noch die Antwort auf die Frage aus, wo und wann die Anmeldung des IIO-Geräts innerhalb eines Gerätetreibers stattfindet. Das hängt von der Art der Anbindung der Hardware ab.

Bei einem per I²C angebundenen Sensor beispielsweise meldet sich der Gerätetreiber beim I²C-Subsystem an und registriert das IIO-Gerät, sobald I²C die zugehörige Hardware erkannt respektive als vorhanden gemeldet bekommen hat.

Für Hardware, die sich nicht direkt einem klassischen Bussystem zuordnen lässt, hat Kernel-Chef Torvalds den Plattformbus integriert. Den verwenden wir als Basis in unserem Beispielcode und definieren daher einen Plattformtreiber samt Plattformgerät.

Plattform-Device

Das Plattform-Device, das eine Probe- sowie eine Remove-Funktion zur Verfügung stellt, wird beim Laden des Treibers instanziiert, was einen Aufruf der Probe-Funktion auslöst. Innerhalb dieser Funktion wird das virtuelle IIO-Device beim

Kanäle über Makros

Programmierer sind bekanntlich faule Zeitgenossen, die unnötige Redundanzen gern vermeiden. So findet man in vielen Treibern des Industrial-IO-Subsystems die Definition eines Kanals per Makro. Das bietet sich immer dann an, wenn ein Sensor mehrere gleichartige Ressourcen bietet, wie etwa die acht Kanäle eines AD-Wandlers. Listing 4 zeigt repräsentativ ein solches Makro, das mehrere Kanäle definiert, die eine Spannung liefern.

Listing 3: Kanaldefinition per Datenstruktur

```
static const struct iio_chan_spec foo_channels[] = {
  {
    .type = IIO_VOLTAGE,
    .indexed  = 1,
    .channel  = 0,
    .address  = 0,
    .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
    .info_mask_shared_by_type = BIT(IIO_CHAN_INFO_
SCALE) | BIT(IIO_CHAN_INFO_OFFSET ),
  },
  {
    .type = IIO_VOLTAGE,
    .indexed  = 1,
    .channel  = 1,
    .address  = 1,
    .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
    .info_mask_shared_by_type =
BIT(IIO_CHAN_INFO_SCALE),
  },
  IIO_CHAN_SOFT_TIMESTAMP(2),
};
```

Kernel über die Funktion `iio_device_register()` angemeldet (Listing 5).

Den vollständigen Beispielcode `iio-foo.c` zusammen mit einem Makefile haben wir für Sie zum Download bereitgestellt. Haben Sie schon einmal Kernel-Code auf Ihrer Maschine generiert, dann genügt die Eingabe von `make` auf der Konsole, damit das Kernel-Build-System aus dem Quellcode `iio-foo.c` den Treiber `iio-foo.ko` erzeugt **3**. Ansonsten müssten Sie unter Ubuntu oder Pi OS vorher noch über `sudo apt install build-essential flex bison` notwendige Pakete nachinstallieren.

Vor dem Laden des per `make` generierten Treibers muss allerdings der Treibercode für das IIO-Subsystem im Kernel liegen. Daher steht vor dem `insmod iio-foo.ko` erst einmal ein `modprobe industrialio` an. Geht alles gut – erscheint also keine Fehlermeldung – ploppt unterhalb des Verzeichnisses `/sys/bus/iio/devices/` ein neuer Ordner auf. War bisher kein weiteres IIO-Device aktiviert, heißt er `iio:device0`.

```
quade@raspberrypi:~/116 $ cat Makefile
obj-m := iio-foo.o

KERNELDIR ?= /lib/modules/$(shell uname -r)/build

all default: modules
install: modules_install

modules modules_install help clean:
	$(MAKE) -C $(KERNELDIR) M=$(shell pwd) $@
quade@raspberrypi:~/116 $ make
make -C /lib/modules/5.10.17-v7l-iio+/build M=/home/quade/116 modules
make[1]: Entering directory '/usr/src/linux'
  CC [M]  /home/quade/116/iio-foo.o
  MODPOST /home/quade/116/Module.symvers
  CC [M]  /home/quade/116/iio-foo.mod.o
  LD [M]  /home/quade/116/iio-foo.ko
make[1]: Leaving directory '/usr/src/linux'
quade@raspberrypi:~/116 $ sudo su
root@raspberrypi:/home/quade/116# insmod iio-foo.ko
root@raspberrypi:/home/quade/116# ls /sys/bus/iio/devices/iio\:device0/
buffer                   in_voltage0_raw    in_voltage_scale  scan_elements  uevent
current_timestamp_clock  in_voltage1_input  name              subsystem
dev                      in_voltage_offset  power             trigger
```

3 **Direkt nach dem Generieren und Laden steht das IIO-Gerät zur Verfügung.**

Ein in dem Verzeichnis ausgeführtes `ls` zeigt die instanziierten Kanäle, insbesondere `in_voltage0_raw` und `in_voltage1_input`. Der Zugriff per `cat in_voltage0_raw` gibt die über einen simplen Zähler simulierte Spannung als Fließkommawert zurück. Der Zähler und damit die Spannung wird übrigens bei jedem Zugriff inkrementiert.

Der Zugriff auf `in_voltage1_input` liefert die bereits skalierte Spannung. Im Beispiel wird unser Fake-Wert zur Differenzierung skaliert und der Offset aufaddiert. Außerdem unterstützt der Treiber die Attribute `in_voltage_scale` und `in_voltage_offset`, die beide beim Lesen einen konstanten Wert liefern.

Mit Puffer

In der letzten Kern-Technik haben wir bereits gezeigt, dass und wie das IIO-

Listing 4: Kanalspezifikation per Makro

```
#define FAKE_VOLTAGE_CHANNEL(num) \
{                                 \
  .type = IIO_VOLTAGE,            \
  .indexed = 1,                   \
  .channel = (num),               \
  .address = (num),               \
  .scan_index = (num),            \
  .info_mask_separate =
BIT(IIO_CHAN_INFO_RAW),           \
  .info_mask_shared_by_type =
BIT(IIO_CHAN_INFO_SCALE) \
}
FAKE_VOLTAGE_CHANNEL(0),
FAKE_VOLTAGE_CHANNEL(1),
```

Listing 5: Probe-Funktion

```
static int foo_pdrv_probe (struct platform_device
*pdev)
{
  int ret;
  struct iio_dev *indio_dev;
  struct foo_private_data *data;

  indio_dev = devm_iio_device_alloc(&pdev->dev,
sizeof(*data));
  if (!indio_dev)
    return -ENOMEM;
  indio_dev->dev.parent = &pdev->dev;
  indio_dev->info = &foo_iio_info;
  indio_dev->name = KBUILD_MODNAME;
  indio_dev->modes = INDIO_DIRECT_MODE;
  indio_dev->channels = foo_channels;
  indio_dev->num_channels = ARRAY_SIZE(
foo_channels);

  ret = iio_device_register(indio_dev);
  if (ret < 0) {
    return -EINVAL;
  }
  platform_set_drvdata(pdev, indio_dev);
  return 0;
}
```

```
quade@raspberrypi:~/116 $ sudo su
root@raspberrypi:/home/quade/116# modprobe industrialio industrialio_triggered_buffer
root@raspberrypi:/home/quade/116# modprobe iio_trig_hrtimer
root@raspberrypi:/home/quade/116#
root@raspberrypi:/home/quade/116# insmod iio-foo.ko
root@raspberrypi:/home/quade/116# mkdir /sys/kernel/config/iio/triggers/hrtimer/foo
root@raspberrypi:/home/quade/116#
root@raspberrypi:/home/quade/116# echo 1 >/sys/bus/iio/devices/trigger0/sampling_frequency
root@raspberrypi:/home/quade/116#
root@raspberrypi:/home/quade/116# cd /sys/bus/iio/devices/iio\:device0
root@raspberrypi:/sys/bus/iio/devices/iio:device0#
root@raspberrypi:/sys/bus/iio/devices/iio:device0# echo "foo" > trigger/current_trigger
root@raspberrypi:/sys/bus/iio/devices/iio:device0# echo 10 > buffer/length
root@raspberrypi:/sys/bus/iio/devices/iio:device0# echo 6 > buffer/watermark
root@raspberrypi:/sys/bus/iio/devices/iio:device0#
root@raspberrypi:/sys/bus/iio/devices/iio:device0# for i in scan_elements/*_en
> do
> echo 1 > $i
> done
root@raspberrypi:/sys/bus/iio/devices/iio:device0# echo 1 > buffer/enable
root@raspberrypi:/sys/bus/iio/devices/iio:device0#
root@raspberrypi:/sys/bus/iio/devices/iio:device0# hd /dev/iio\:device0
00000000  01 00 02 00 00 00 00 00  ed 16 4c 54 62 20 79 16  |..........LTb y.|
00000010  01 00 02 00 00 00 00 00  11 e8 e6 8f 62 20 79 16  |............b y.|
00000020  01 00 02 00 00 00 00 00  8b 93 81 cb 62 20 79 16  |............b y.|
00000030  01 00 02 00 00 00 00 00  f6 54 1c 07 63 20 79 16  |.........T..c y.|
00000040  01 00 02 00 00 00 00 00  ae 26 b7 42 63 20 79 16  |.........&.Bc y.|
00000050  01 00 02 00 00 00 00 00  56 13 52 7e 63 20 79 16  |........V.R~c y.|
```

4 Es erfordert etwas Konfiguration, die Trigger zum Laufen zu bringen.

Subsystem Trigger und Puffer unterstützt, um unterschiedliche Messwerte zeitgleich als konsistente Datensätze zu erfassen. Das funktioniert allerdings nur dann, wenn der Treiber das auch unterstützt.

Schade, dass sich viele Programmierer den Code dazu sparen. Das mag daran liegen, dass man dafür nicht auf die existierenden Lese- und Schreibfunktionen (`read_raw()`, `write_raw()`) zurückgreifen kann, sondern eine eigene Zugriffsfunktion her muss (Listing 6). Sie nimmt für jeden überwachten Kanal den Zugriff vor und legt das Ergebnis im Rohformat an die passende Stelle in einen Datenpuffer.

Den liest der Anwender schließlich über eine klassische Gerätedatei aus (in Abgrenzung zu den Dateien im Sys-Filesystem). Dazu übergibt man dem IIO-Subsystem den Puffer inklusive Zeitstempel. Da parallel zum automatisierten Zugriff auch ein manueller Zugriff möglich ist, gibt es einen kritischen Abschnitt, den zumeist ein Mutex sichert.

Mutex und Puffer benötigen Speicher, den der Entwickler pfiffigerweise direkt zusammen mit der Datenstruktur `iio_dev` reserviert. Das spart ein unnötiges Malloc und später die zugehörige Freigabe. Die Größe des Buffers muss übrigens eine Zweierpotenz sein.

Darüber hinaus implementiert der Entwickler die Zugriffsfunktion, die das IIO-Subsystem aufruft. Dabei handelt es sich um eine Interrupt-Service-Routine, weshalb beim Zugriff ein Schlafenlegen tabu ist. Der Code holt in einer Schleife für jeden überwachten Kanal die Daten und legt sie im Puffer ab. Außerdem gilt es, einen Zeitstempel als 64-Bit-Wert zu lesen, was über die Funktion `iio_get_time_ns()` erfolgen kann. Die Daten werden dem IIO-Subsystem übergeben, das sie dann quittiert.

Der Entwickler darf nicht vergessen, auch die Kanalspezifikation noch für jeden Kanal zu erweitern (Listing 7). Der Zugriff über die Puffer findet grundsätzlich auf die Rohwerte statt. Daher wird die exakte Repräsentationsform des zugehörigen Werts angegeben, also ob er im Little- oder im Big-Endian-Format vorliegt, wie viele Bits er hat und ob er vor der Nutzung noch um ein paar Bits verschoben werden muss. Außerdem gilt es, beim Kanal über die Angabe einer Nummer zu spezifizieren, an welcher Position im Buffer der Rohwert abgelegt wird.

Ein Bitfeld, die Scan-Maske, spezifiziert die zu überwachenden Kanäle (Listing 8). Das Bit an der Position X steht für den Kanal X, ein Null-Eintrag signalisiert das Ende des Felds. Die Scan-Maske wird vor dem Anmelden des IIO-Geräts beim IIO-Subsystem an die Datenstruktur `struct iio_dev` geknüpft.

Das eigentliche Einklinken des Trigger-Mechanismus findet direkt vor dem Registrieren des Geräts beim IIO-Subsystem (`iio_device_register()`) durch den Aufruf der Funktion `iio_triggered_buffer_setup()` statt (Listing 9). Dieser Funktion übergibt der Entwickler neben

Listing 6: Mehrere Kanäle zeitgleich Lesen

```
static irqreturn_t foo_trigger_handler( int irq,
void *p )
{
  struct iio_poll_func *pf = p;
  struct iio_dev *indio_dev = pf->indio_dev;
  struct foo_private_data *data;
  int bit=0, i=0;
  s64 time_ns = iio_get_time_ns( indio_dev );

  data = iio_priv(indio_dev);
  // get data...
  mutex_lock( &data->lock );
  for_each_set_bit( bit, indio_dev->active_scan_mask,
    indio_dev->masklength ) {
    ((s16 *)data->buffer)[i] = i+1; // fake data
    i++; // fake data
  }
  mutex_unlock( &data->lock );

  iio_push_to_buffers_with_timestamp( indio_dev,
      data->buffer, time_ns );
  iio_trigger_notify_done( indio_dev->trig );
  return IRQ_HANDLED;
}
```

dem IIO-Gerät die Adresse der implementierten Trigger-Funktion sowie die gewünschte Funktion (hier `io_pollfunc_store_time`).

Jetzt fehlt bloß noch das Aufräumen, sprich: das Abmelden des Triggers durch den Aufruf von `iio_triggered_buffer_cleanup()`. Das wird zum einen im Fehlerfall aufgerufen (falls das Anmelden fehlschlägt), zum anderen vor dem Abmelden des IIO-Geräts beim IIO-Subsystem (Listing 10).

Damit der Trigger auslöst, muss man noch die Module `industrialio_triggered_buffer` und `iio_trig_hrtimer` laden. Dann wird der Trigger angelegt und dem IIO-Gerät zugeordnet, die Sampling-Frequenz eingestellt, das Buffering konfiguriert, die zu überwachenden Kanäle aktiviert und schließlich die Datenerfassung gestartet 4.

Weiter geht's

Greift die Applikation anschließend auf die Gerätedatei `/dev/iio:device0` zu, liest sie die beiden Kanäle inklusive des Zeitstempels aus. Letzterer liefert die Anzahl der Nanosekunden, die seit dem 1. Januar1970 vergangen sind.

Neben dem Trigger bietet das IIO-Subsystem noch einen Event-Mechanismus an, den wir hier nicht weiter thematisieren. Daneben haben die IIO-Macher einen High-Speed-Zugriff implementiert, der hardwaretechnisch auf die Datenerfassung per DMA setzt und softwaretechnisch ein möglichst weitgehendes Zero-Copy realisiert. Damit wäre treiberseitig alles an Bord, es fehlen nur noch echte Sensoren. (*jlu*)

Weitere Infos und interessante Links

www.lm-online.de/qr/44538

Dateien zum Artikel herunterladen unter

www.lm-online.de/dl/44538

Listing 7: Kanalspezifikation erweitern

```
.scan_index = 0,
.scan_type = {
  .sign = 's',
  .realbits = 12,
  .storagebits = 16,
  .shift = 4,
  .endianness = IIO_LE,
},
```

Listing 8: Scan-Maske

```
static const unsigned long foo_
scan_masks[] = {
  3, // scan support for channel
0 and 1 (b0011)
  0  // end
};
```

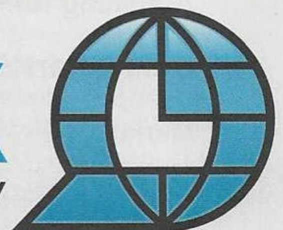
Listing 9: Trigger-Support initialisieren

```
[...]
indio_dev->available_scan_masks =
foo_scan_masks;
data = iio_priv(indio_dev);
mutex_init(&data->lock);

ret = iio_triggered_buffer_
setup(indio_dev,
  iio_pollfunc_store_time,
  foo_trigger_handler, NULL);
if (ret < 0)
  return -EINVAL;
ret = iio_device_register(indio_
dev);
[...]
```

Listing 10: Abmelden des IIO-Gerätes

```
static int foo_pdrv_remove(struct platform_device *pdev)
{
  struct iio_dev *indio_dev = platform_get_drvdata(pdev);
  iio_triggered_buffer_cleanup( indio_dev );
  iio_device_unregister( indio_dev );
  return 0;
}
```

Immer aktuell informiert mit dem **COMMUNITY NEWSLETTER!**

www.linux-community.de/newsletter