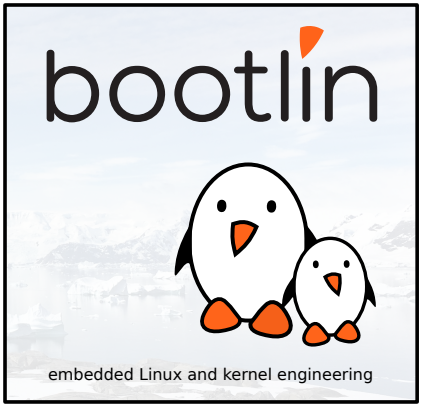
IIO, a new kernel subsystem

Maxime Ripard
maxime.ripard@bootlin.com



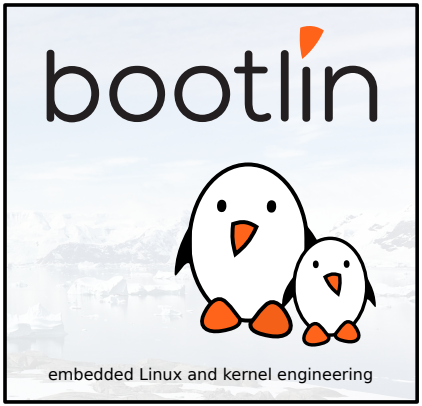
Outline

Maxime Ripard

- Embedded Linux engineer and trainer at Bootlin since 2011
  - Embedded Linux development: kernel and driver development, system integration, boot time and power consumption optimization, etc.
  - Embedded Linux, Embedded Android and driver development training, with materials freely available under a Creative Commons license.
  - `https://bootlin.com`
- Contributor to Buildroot, a simple open-source embedded build system

What is IIO ?

Maxime Ripard
maxime.ripard@bootlin.com



Definition

IIO

- A subsystem for Analog to Digital Converters (ADCs) and related hardwares (accelerometers, light sensors, gyroscopes), but also DACs
- Can be used on ADCs ranging from a SoC ADC to 100M samples/sec industrial ADCs
- Until recently, mostly focused on user-space abstraction with no in-kernel API for other drivers

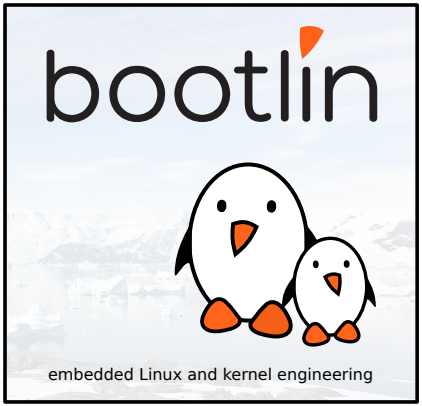
Current state in the kernel

Current state in the kernel

- Developed since 2009 by Jonathan Cameron
- Being developed in the `staging/` directory until it comes to an high quality code and a mature API
- It is now moving out of staging, one step at a time: first, basic features, then the support for advanced IIO features.
- Already has a lot of different hardware supports and drivers for them, mostly from Analog Devices Inc, but also drivers for Texas Instruments, Atmel, etc.

Getting started

Maxime Ripard
maxime.ripard@bootlin.com



IIO device

iio_dev structure

- Main structure of all IIO drivers
- Holds informations about the device and the driver, such as :
  - How much channels are available on the device ?
  - What modes can the device operate in ?
  - What hooks are available for this driver ?

Allocate a new device

```
struct iio_dev *idev = iio_allocate_device)
                                sizeof(struct at91_adc_state));
```

- Allocates a struct iio_dev, along with the private data of your driver
- Does all the basic initialisation

Capture modes

```
idev->modes = INDIO_DIRECT_MODE;
```

Defines the mode of operations available for this device, to choose between :

INDIO_DIRECT_MODE the device can operate using software triggers

INDIO_BUFFER_TRIGGERED the device can use hardware triggers

INDIO_BUFFER_HARDWARE the device has a hardware buffer

INDIO_ALL_BUFFER_MODES union of the two above

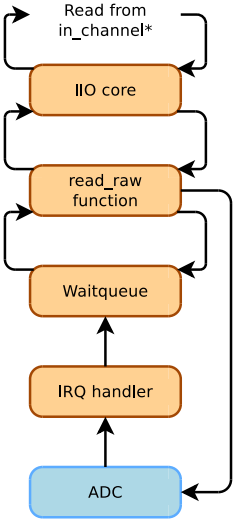
IIO infos

```
static const struct iio_info at91_adc_info = {
        .driver_module = THIS_MODULE,
        .read_raw = &at91_adc_read_raw,
};
```

```
idev->info = &at91_adc_info;
```

- Used to declare the hooks the core can use for this device
- Lot of hooks available corresponding to interactions the user can make through sysfs.
- `read_raw` for example is called to request a value from the driver. A bitmask allows us to know more precisely which type of value is requested, and for which channel if needed. It can be for example either the scale used to convert value returned to volts or the value in itself.

Basic design of the driver



IIO channels

iio_channels declaration

```
struct iio_chan_spec *chan = kmalloc(sizeof(struct iio_chan_spec),
                                     GFP_KERNEL);
chan->type = IIO_VOLTAGE;
chan->indexed = 1;
chan->channel = 1;
chan->scan_type.sign = 'u';
chan->scan_type.realbits = 10;
chan->scan_type.storagebits = 16;
chan->info_mask = IIO_CHAN_INFO_SCALE_SHARED_BIT;

idev->channels = chan;
idev->num_channels = 1;
```

Register the device

```
iio_device_register(idev);
```

- this is sufficient to have a basic IIO device driver

Userspace API

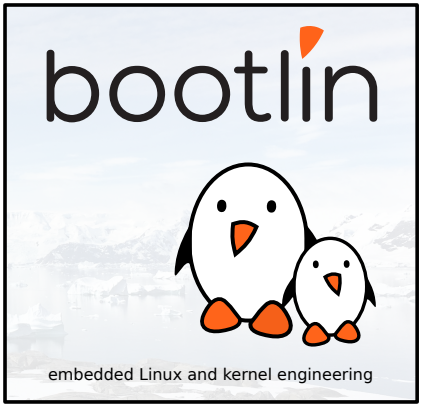
If we look at `/sys/bus/iio/devices/iio:deviceX`, we should have:

```
$ ls /sys/bus/iio/devices/iio:deviceX
in_voltage0_raw
in_voltage_scale
name
$
```

- reading in_voltage0_raw calls the read_raw hook, with the mask set to 0, and the chan argument set with the iio_chan_spec structure corresponding to the the channel 0
- reading in_voltage_scale calls the read_raw hook, with the mask set to IIO_CHAN_INFO_SCALE

Going further : Hardware triggers and buffers

Maxime Ripard
maxime.ripard@bootlin.com

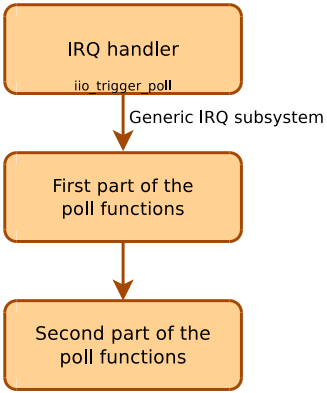


Hardware Triggers

Hardware Triggers

IIO exposes API so that we can :

- declare any given number of triggers
- choose which channels we want enabled for conversions



Kernel API

```
static const struct iio_trigger_ops at91_adc_trigger_ops = {
        .owner = THIS_MODULE,
        .set_trigger_state = &at91_adc_configure_trigger,
};
```

```
struct iio_trigger *trig = iio_allocate_trigger("foo");
trig->ops = &at91_adc_trigger_ops;
iio_trigger_register(trig)
```

- Once again, we have hooks to declare
- These hooks are this time triggers-specific, so that we can have a function called when the trigger state changes.

Poll Function

- function triggered at each conversion
- called through the functions `iio_trigger_poll` or `iio_trigger_poll_chained`
- basically, its job is to feed retrieve data from the device and feed them into the buffer
- IIO uses the IRQ model, so the poll function has the same prototype than any other IRQ handler.

```
idev->pollfunc = iio_alloc_pollfunc(&iio_pollfunc_store_time,
                                    &at91_adc_trigger_handler,
                                    IRQF_ONESHOT,
                                    idev,
                                    "foo");
```

Buffers

Buffers

- doesn't make much sense to have triggered captures without a buffer
- Buffers and triggers are closely tied together in IIO
- 2 types of buffers in IIO
  - One relies on kfifo
  - The other one is a ring buffer

Allocate a buffer

- in the latest code, if you want to use the IIO ring buffer, boiler plate code has been added so it's pretty straightforward.

```
ret = iio_sw_rb_simple_setup(idev,
                             &iio_pollfunc_store_time,
                             &at91_adc_trigger_handler);
```

- Allocate the buffer, allocates the poll function, declares the device as supporting triggered capture, register the buffer against the core, etc

Userspace API 1/3: Set up the capture

If we look at `/sys/bus/iio/devices/`, we should now have in addition to `iio:deviceX`:

```
$ ls /sys/bus/iio/devices/
iio:device0
trigger0
$ ls /sys/bus/iio/devices/iio:device0
buffer
scan_elements
trigger
$ ls /sys/bus/iio/devices/iio:device0/buffer
enabled length
$ ls /sys/bus/iio/devices/iio:device0/scan_elements
in_voltage0_en
in_voltage0_index
in_voltage0_type
$ ls /sys/bus/iio/devices/trigger0
name
$
```

Userspace API 2/3: Start and stop a capture

```
$ echo 1 > \
    /sys/bus/iio/devices/iio:device0/scan_elements/in_voltage0_en
$ echo "foo" > \
    /sys/bus/iio/devices/iio:device0/trigger/current_trigger
$ echo 100 > /sys/bus/iio/devices/iio:device0/buffer/length
$ echo 1 > /sys/bus/iio/devices/iio:device0/buffer/enable
$ echo 0 > /sys/bus/iio/devices/iio:device0/buffer/enable
$ echo "" > \
    /sys/bus/iio/devices/iio:device0/trigger/current_trigger
```

Userspace API 3/3: Get the capture results

- IIO also exposes a character device to get the converted values: `/dev/iio:deviceX`
- You just have to read in it to get data
- Data are organized by chunks
  - Example: You have 4 channels, plus a timestamp one. All are enabled except channel 2.

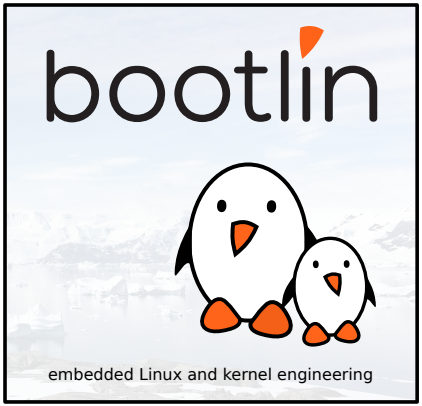
	Channels			Timestamp
Index	0	1	3	
Size in bits	16	16	16	64

- There is a program that provides a basic implementation and a good way to test your driver in the IIO documentation directory: `drivers/staging/iio/Documentation/generic_buffer.c`

```
# ./generic-buffer -n at91_adc -t at91_adc-dev0-external
```

Useful Resources

Maxime Ripard
maxime.ripard@bootlin.com

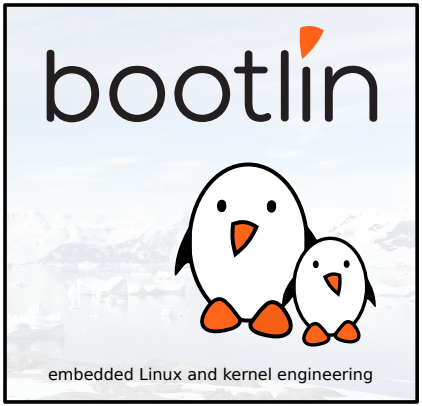


Useful Resources

- drivers/staging/iio/Documentation
- drivers/staging/iio/iio_simple_dummy.c
- http://www.ohwr.org/projects/zio/wiki/Iio
- http://www.at91.com/linux4sam/bin/view/Linux4SAM/IioAdcDriver
- linux-iio@vger.kernel.org

Conclusion

Maxime Ripard
maxime.ripard@bootlin.com



Conclusion

- IIO is a nice subsystem to add ADCs and the like support
- Still under heavy development, but also really opens to changes and feedback
- Yet reliable enough to be used in production