Konsulko Group

Industrial I/O and You: Nonsense Hacks!

Matt Ranostay
Konsulko Group
<matt.ranostay@konsulko.com>

Brief Introduction

- Been a contributor to the Industrial I/O system for about two years
  - Any weird sensors you see in the IIO system are likely mine
  - Chemical sensors e.g. pH, gas sensors, etc
  - Lightning sensors
- Feel free to ask questions during the talk!

Introduction to Industrial I/O Subsystem

- Kernel subsystem that allows ease of implementing of drivers for sensors and some even miscellaneous devices (e.g. potentiometers, DACs)
- Solves use cases that hwmon subsystem can’t handle
  - High speed sensors
  - Triggered sampling
  - Data buffering
- Provides a stable ABI for various userspace HALs
  - libiio is the ‘official’ preferred one
- Simple API that allows easy driver development for sensors
  - Much boilerplate code exists for most device types
  - Often requires only slight modification to existing IIO drivers

Introduction to Industrial I/O Subsystem

- Typically sensors but there are niche drivers that aren’t in that category
    - Potentiometers
    - DACs
    - Clock Generators
- Industrial I/O is also a fitting place for some drivers that would probably end up in drivers/misc otherwise
- Due to the dedicated API it is a much better solution than using the input subsystem
    - Android devices have been slowly switching drivers over from input + misc subsystems to IIO

Industrial I/O Sensors Types

- Accelerometers + Gyroscopes + Magnetometers + IMUs
- Temperature + Barometric Pressure + Humidity
- ADCs
- Ambient Light Sensors (Colour Detection + Lux)
- Gesture Sensors
- Chemical Sensors
- Health Sensors
- Sensor Hubs

Industrial I/O Components Overview

- IIO channels
- Triggers (SW + HW based)
- Data Buffers
- Single-shot Data Access
- IIO Events
- IIO Channel Consumers

IIO Driver Development (Steps)

- Are you completely sure this shouldn’t be a hwmon driver?
- What is the interface to the sensor chip? I2C, SPI, gpio bit-banging.
  - Should regmap be used here?
  - Any interrupt lines?
- HW or SW triggers need to be used?
  - Hrtimer periodic interrupts is the most useful software trigger
  - GPIO interrupt trigger is also useful for ad-hoc events
  - HW triggers are typical from interrupt lines, or from another iio driver’s trigger
- What is the device type?
  - ADCs should export their channels so other consumer drivers can use it (e.g. hwmon-iio)
  - Single shot readings or will use triggered or kfifo buffers
  - Will this require new channel types (e.g. IIO_PH) or modifiers

Industrial I/O - When to Use

- hwmon subsystem doesn’t fit into sensor usage
    - Typically low speed local temperature + humidity sensor that fits lmsensors use aren’t accepted for IIO
- Sensor data needs to be deterministic, and samples not handled in time need to be dropped
    - Needs a per sample timestamp which is important for fusion code integration
    - Userspace HAL misses samples the KFIFO needs to keep attempting to store data
- Devices (typically ADCs) that will have other consumers within the kernel
    - Examples are battery chargers, fuel gauges, and even thermal sensors that use an IIO ADC driver’s channels

IIO Channels (struct iio_chan_spec)

- Devices can have multiple channels defined for data reporting
  - Temperature (IIO_TEMP), Humidity (IIIO_RELATIVEHUMIDITY), and etc.
  - Modifiers for channel type to give more information to userspace.
    - Example: IO_CONCENTRATION has IIO_MOD_CO2, and IIO_MOD_VOC
- Bit mask for enumeration of features per channel, channel type, and etc
  - IIO_CHAN_INFO_RAW
  - IIO_CHAN_INFO_OFFSET
  - IIO_CHAN_INFO_SCALE
  - IIO_CHAN_INFO_PROCESSED
- Direction of channel as input or output. Most are inputs, but heater control for humidity sensors is an example of an output

IIO Channel (struct iio_chan_spec) Example

```
const struct iio_chan_spec max31855_channels[] = {
	{		/* thermocouple temperature */
		.type = IIO_TEMP,
		.address = 2,
		.info_mask_separate =
			BIT(IIO_CHAN_INFO_RAW) | BIT(IIO_CHAN_INFO_SCALE),
		.scan_index = 0,
		.scan_type = {
			.sign = 's',
			.realbits = 14,
			.storagebits = 16,
			.shift = 2,
			.endianness = IIO_BE,
		},
	},
```

**** Continued on next page ****

IIO Channel (struct iio_chan_spec) Example

```
	{		/* cold junction temperature */
		.type = IIO_TEMP,
		.address = 0,
		.channel2 = IIO_MOD_TEMP_AMBIENT,
		.modified = 1,
		.info_mask_separate =
			BIT(IIO_CHAN_INFO_RAW) | BIT(IIO_CHAN_INFO_SCALE),
		.scan_index = 1,
		.scan_type = {
			.sign = 's',
			.realbits = 12,
			.storagebits = 16,
			.shift = 4,
			.endianness = IIO_BE,
		},
	},
	IIO_CHAN_SOFT_TIMESTAMP(2),
};
```

Software + Hardware Triggers

- Hardware based triggers
  - GPIO-based interrupts are typical
- Software based triggers
  - Sysfs - simple way to trigger a data poll from userspace
  - Hrtimer - much more useful polling using the high resolution timer
- Triggers can be mapped to multiple devices
- Triggers can also be chained to other IIO drivers to signal status
  - Example case is an IIO consumer driver requesting a sample from another IIO ADC driver
- Triggers can be signaled within the kernel as well using the iio_trigger_poll() call
- IIO driver can be a provider and/or consumer of a trigger(s)

Buffers Overview

- Data is usually not processed so channels need to have some configuration to describe a few key points
  - Storage size
  - Real size, and right shifting of data
  - Endianness
- Buffered data is typical a raw SPI/I2C block read passed to userspace, and the userspace HAL processing the data using the configuration noted above

Buffers Explained

- KFIFO backed
  - Allows data to be gracefully dropped if userspace HAL can’t keep up
- Triggerable buffers from software + hardware events
- Configurable sample size to allocate for buffer
  - echo 16 > /sys/bus/iio/devices/iio:device0/buffer/length
- Per device buffers (e.g. local interrupts, FIFO)
  - Usually used for device with a HW FIFOs that can’t be synced with other devices
- Callback buffers to allow IIO drivers to communicate data to other kernel subsystems

Buffers Explained (Handler Example)

```
static irqreturn_t lidar_trigger_handler(int irq, void *private)
{
    struct iio_poll_func *pf = private;
    struct iio_dev *indio_dev = pf->indio_dev;
    struct lidar_data *data = iio_priv(indio_dev);
    int ret;

    ret = lidar_get_measurement(data, data->buffer);
    if (!ret) {
        iio_push_to_buffers_with_timestamp(indio_dev, data->buffer,
                                           iio_get_time_ns(indio_dev));
    } else if (ret != -EINVAL) {
        dev_err(&data->client->dev, "cannot read LIDAR measurement");
    }

    iio_trigger_notify_done(indio_dev->trig);

    return IRQ_HANDLED;
}
```

Single-shot Data Access

- Sensors that have no fast access methods, or when simple access that is all is needed
  - Most sensors that can handle high speed buffered access can report data in this slower method. e.g. thermocouples, ambient light, ADCs, and chemical sensors.
- Some data cannot be single shot data, and this is usually if it is deterministic or useless without calculating the deltas between samples
  - Oximeter data is basically a glorified ADC but have single shot samples that make no sense alone
- Secondary channels that have no real application in data processing but can’t hurt to have

IIO Data Capture Example (Single-shot Reading)

```
$ cd /sys/bus/iio/devices/iio:device0
$ cat in_temp_raw
98
$ cat in_temp_ambient_raw
416
$ cat in_temp_scale
250
$ cat in_temp_ambient_scale
62.500000
```

IIO Data Capture Example (Single-shot Reading)

- Processing the results into a milliCesius reading (weird units due to being backwards compatible with hwmon)
  - Thermocouple Temp -> 98 * 250 = **24500** = 24.5 C
  - Ambient Temp (cold junction) -> 416 * 62.5 = **26000** = 26.0 C

IIO Events (struct iio_event_spec)

- Events are used to signal back to userspace that some actions has happened
- Access is via an IOCTL interface on the character device (i.e /dev/iio:device0)
    - Userspace application sets up monitoring via IIO_GET_EVENT_FD_IOCTL, and polls for events
- Different than buffers because they don’t signal any values or raw data reads
- Typically threshold events (i.e IIO_EV_TYPE_THRESH) are signaled along with some event data
    - Channel type and modifier
    - Direction information
        - IIO_EV_DIR_EITHER
        - IIO_EV_DIR_RISING
        - IIO_EV_DIR_FALLING

IIO Events (struct iio_event_spec) Example

```
static const struct iio_event_spec apds9960_pxs_event_spec[] = {
	{
		.type = IIO_EV_TYPE_THRESH,
		.dir = IIO_EV_DIR_RISING,
		.mask_separate = BIT(IIO_EV_INFO_VALUE) |
			BIT(IIO_EV_INFO_ENABLE),
	},
	{
		.type = IIO_EV_TYPE_THRESH,
		.dir = IIO_EV_DIR_FALLING,
		.mask_separate = BIT(IIO_EV_INFO_VALUE) |
			BIT(IIO_EV_INFO_ENABLE),
	},
};
```

IIO Consumer Channels

- Allows the channels of a provider to be consumed by another driver in the kernel
  - API calls to access are iio_channel_read() and iio_channel_read_processed()
  - Support for multiple consumers avoids one off drivers to access ADC data
  - Useful for exposing ADC channels to drivers that are monitoring battery charging, fuel gauges, and even some touchscreens
    - hwmon also has an IIO channel consumer driver
- Scaling will be handled transparently with iio_channel_read_processed() even if the provider doesn’t have an IIO_CHAN_INFO_PROCESSED in the .info_mask_*

IIO Data Processing

- Data should be outputted to userspace unprocessed (IIO_CHAN_TYPE_RAW) when possible to keep the time spent in kernel space to a minimum.
  - Also floating point calculations within the kernel are discouraged heavily, and would be rejected upstream
- Times where processing (IIO_CHAN_TYPE_PROCESSED) is required is when the scaling and offset value varies due to another variable, or when results are non-linear
  - Example would be the temperature correction of a some high end humidity and pressure sensors. e.g. BMP280/BME280

IIO Data Processing

- SI Units are typically used (sorry no Imperial Units)
- ABI for both driver and HAL development is documented in Documentation/ABI/testing/sysfs-bus-iio
- Check ABI documentation first when writing your IIO channels for the correct scaling value you’ll need to use
  - IIO_TEMP -> milliCelsius
  - IIO_VOLTAGE -> millivolts
  - IIO_RESISTANCE -> ohms
  - IIO_DISTANCE -> meters

IIO Utils - Overview

- lsiio - enumerate IIO triggers, devices, and accessible channels
- iio_event_monitor - monitor on IIO device’s ioctl interface for IIO events
- iio_generic_buffer - monitors, processes, and print data received from a IIO device’s buffer

IIO Utils - Enumeration

```
$ lsiio -v
Device 000: maxim_thermocouple
  in_temp_raw
  in_temp_ambient_raw
Device 001: ams-iaq-core
  in_concentration_co2_input
  in_concentration_voc_input
  In_resistance_input
Trigger 000: trigger0
```

IIO Buffered Data Capture

```
$ mkdir /sys/kernel/config/iio/triggers/hrtimer/trigger0
$ echo 50 > /sys/bus/iio/devices/trigger0/sampling_frequency
$ cd /sys/bus/iio/devices/iio:device0
$ echo trigger0 > trigger/current_trigger
$ echo 1 > scan_elements/in_temp_en
$ echo 1 > scan_elements/in_temp_ambient_en
$ echo 1 > scan_elements/in_timestamp_en
$ echo 1 > buffer/enable
```

IIO Buffered Data Capture

```
$ cat /dev/iio:device0 | xxd -
0000000: 0188 1a30 0000 0000 8312 68a8 c24f 5a14  ...0......h..OZ.
0000010: 0188 1a30 0000 0000 192d 98a9 c24f 5a14  ...0.....-...OZ.
0000020: 0188 1a30 0000 0000 4b28 1c2f c34f 5a14  ...0....K(./.OZ.
0000030: 0188 1a30 0000 0000 3f50 4d30 c34f 5a14  ...0....?PM0.OZ.
....
```

IIO Buffered Data Capture

Processing the IIO_TEMP channel data out of one sample from buffer:

```
0000000: 0188 1a30 0000 0000 8312 68a8 c24f 5a14  ...0......h..OZ.
```

```
$ cat scan_elements/in_temp_index
0
$ cat scan_elements/in_temp_type
be:s14/16>>2
```

Temp Data Processing -> 0x188 >> 2 = 98 * 250 = **24500** = 24.5 Celsius

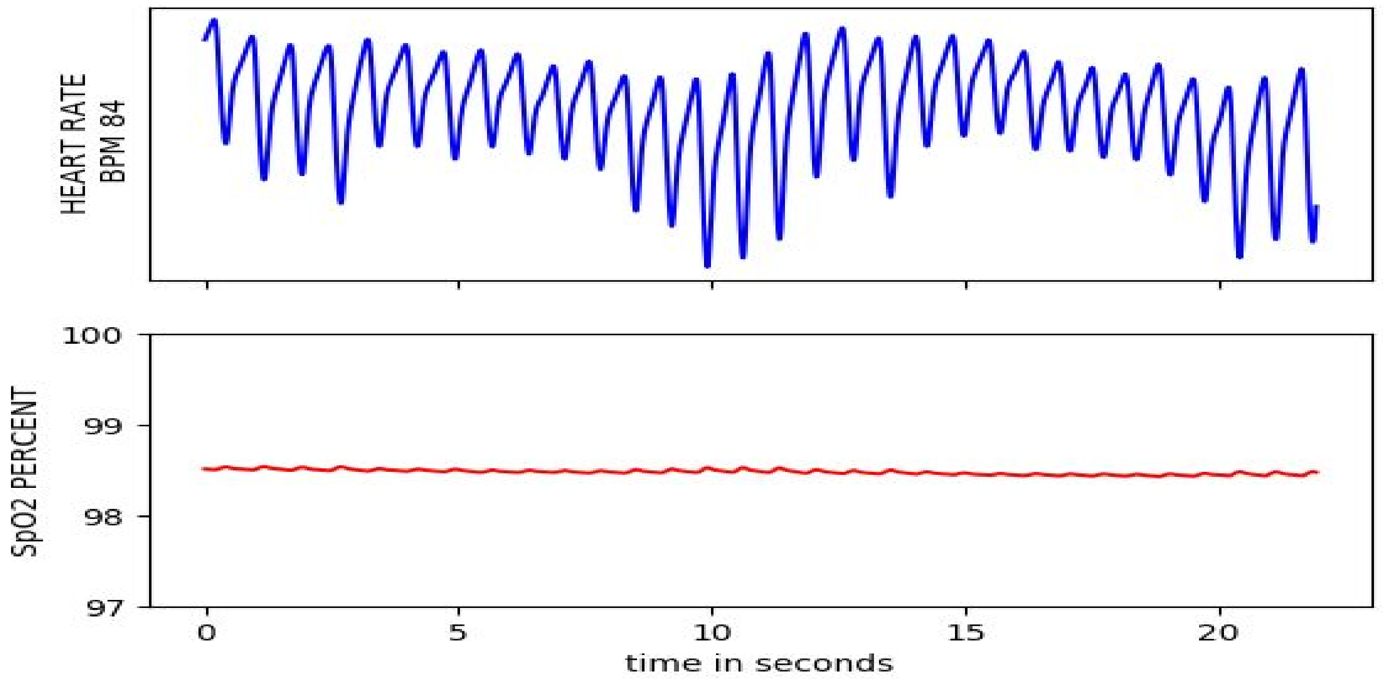
IIO Utils - iio_generic_buffer

```
$ iio_generic_buffer --device-num 1 --trigger-name trigger0 -c 10
iio device number being used is 1
iio trigger number being used is 0
/sys/bus/iio/devices/iio:device1 trigger0
24250.000000 26750.000000 1471056389182591331
24250.000000 26750.000000 1471056389192580164
24250.000000 26750.000000 1471056389202586414
24250.000000 26750.000000 1471056389212630539
24500.000000 26750.000000 1471056389222614789
24500.000000 26750.000000 1471056389232603956
24500.000000 26750.000000 1471056389242606748
24500.000000 26750.000000 1471056389252611164
24500.000000 26750.000000 1471056389262606164
24500.000000 26750.000000 1471056389272607164
```

Demo

- Beaglebone AM335x board
- MAX30102 Oximeter
- libiio userspace application
    - Processes oximeter data with filters and averaging
    - Streaming graphs with Gnuplot

Oximeter Data Processed



Conclusion

- Industrial I/O is good solution for fast updating sensors, and ferrying data to userspace HALs
- Really no limit to what Industrial I/O can support sensor wise, and there isn't anything too niche
- Industry is learning this is a stable ABI to use and is much more suited for sensors than input or misc subsystems
- Patches welcome! ***linux-iio@vger.kernel.org***

QUESTIONS? Hopefully a lot!

References

- Industrial I/O ABI documentation
  - https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/ABI/testing/sysfs-bus-iio
- MAX30102 Datasheet
  - https://datasheets.maximintegrated.com/en/ds/MAX30102.pdf
- Oximeter demo application
  - https://gist.github.com/mranostay/bd926be0753197b78c842f430a754f33