# Industrial I/O driver developer's guide

**Daniel Baluta <daniel.baluta@intel.com>**

# Industrial I/O driver developer's guide

by Daniel Baluta

Copyright © 2015 Intel Corporation

This documentation is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License version 2.

# Table of Contents

1. Introduction ..... 1
2. Industrial I/O core ..... 2
    Industrial I/O devices ..... 2
        IIO device sysfs interface ..... 8
        IIO device channels ..... 9
    Industrial I/O buffers ..... 13
        IIO buffer sysfs interface ..... 18
        IIO buffer setup ..... 18
    Industrial I/O triggers ..... 20
        IIO trigger sysfs interface ..... 23
        IIO trigger setup ..... 23
        IIO trigger ops ..... 24
    Industrial I/O triggered buffers ..... 25
        IIO triggered buffer setup ..... 25
3. Resources ..... 30

# Chapter 1. Introduction

The main purpose of the Industrial I/O subsystem (IIO) is to provide support for devices that in some sense perform either analog-to-digital conversion (ADC) or digital-to-analog conversion (DAC) or both. The aim is to fill the gap between the somewhat similar hwmon and input subsystems. Hwmon is directed at low sample rate sensors used to monitor and control the system itself, like fan speed control or temperature measurement. Input is, as its name suggests, focused on human interaction input devices (keyboard, mouse, touchscreen). In some cases there is considerable overlap between these and IIO.

Devices that fall into this category include:

- analog to digital converters (ADCs)
- accelerometers
- capacitance to digital converters (CDCs)
- digital to analog converters (DACs)
- gyroscopes
- inertial measurement units (IMUs)
- color and light sensors
- magnetometers
- pressure sensors
- proximity sensors
- temperature sensors

Usually these sensors are connected via SPI or I2C. A common use case of the sensors devices is to have combined functionality (e.g. light plus proximity sensor).

# Chapter 2. Industrial I/O core

The Industrial I/O core offers:

- a unified framework for writing drivers for many different types of embedded sensors.
- a standard interface to user space applications manipulating sensors.

The implementation can be found under `drivers/iio/industrialio-*`

# Industrial I/O devices

# Name

struct iio_dev — industrial I/O device

## Synopsis

```
struct iio_dev {
  int id;
  int modes;
  int currentmode;
  struct device dev;
  struct iio_event_interface * event_interface;
  struct iio_buffer * buffer;
  struct list_head buffer_list;
  int scan_bytes;
  struct mutex mlock;
  const unsigned long * available_scan_masks;
  unsigned masklength;
  const unsigned long * active_scan_mask;
  bool scan_timestamp;
  unsigned scan_index_timestamp;
  struct iio_trigger * trig;
  struct iio_poll_func * pollfunc;
  struct iio_poll_func * pollfunc_event;
  struct iio_chan_spec const * channels;
  int num_channels;
  struct list_head channel_attr_list;
  struct attribute_group chan_attr_group;
  const char * name;
  const struct iio_info * info;
  struct mutex info_exist_lock;
  const struct iio_buffer_setup_ops * setup_ops;
  struct cdev chrdev;
#define IIO_MAX_GROUPS 6
  const struct attribute_group * groups[IIO_MAX_GROUPS + 1];
  int groupcounter;
  unsigned long flags;
#if defined(CONFIG_DEBUG_FS)
  struct dentry * debugfs_dentry;
  unsigned cached_reg_addr;
#endif
};
```

## Members

| | |
|---|---|
| id | [INTERN] used to identify device internally |
| modes | [DRIVER] operating modes supported by device |
| currentmode | [DRIVER] current operating mode |
| dev | [DRIVER] device structure, should be assigned a parent and owner |
| event_interface | [INTERN] event chrdevs associated with interrupt lines |

| | |
|---|---|
| buffer | [DRIVER] any buffer present |
| buffer_list | [INTERN] list of all buffers currently attached |
| scan_bytes | [INTERN] num bytes captured to be fed to buffer demux |
| mlock | [INTERN] lock used to prevent simultaneous device state changes |
| available_scan_masks | [DRIVER] optional array of allowed bitmasks |
| masklength | [INTERN] the length of the mask established from channels |
| active_scan_mask | [INTERN] union of all scan masks requested by buffers |
| scan_timestamp | [INTERN] set if any buffers have requested timestamp |
| scan_index_timestamp | [INTERN] cache of the index to the timestamp |
| trig | [INTERN] current device trigger (buffer modes) |
| pollfunc | [DRIVER] function run on trigger being received |
| pollfunc_event | [DRIVER] function run on events trigger being received |
| channels | [DRIVER] channel specification structure table |
| num_channels | [DRIVER] number of channels specified in *channels*. |
| channel_attr_list | [INTERN] keep track of automatically created channel attributes |
| chan_attr_group | [INTERN] group for all attrs in base directory |
| name | [DRIVER] name of the device. |
| info | [DRIVER] callbacks and constant info from driver |
| info_exist_lock | [INTERN] lock to prevent use during removal |
| setup_ops | [DRIVER] callbacks to call before and after buffer enable/disable |
| chrdev | [INTERN] associated character device |
| groups[IIO_MAX_GROUPS + 1] | [INTERN] attribute groups |
| groupcounter | [INTERN] index of next attribute group |
| flags | [INTERN] file ops related flags including busy flag. |
| debugfs_dentry | [INTERN] device specific debugfs dentry. |
| cached_reg_addr | [INTERN] cached register address for debugfs reads. |

# Name

iio_device_alloc — allocate an iio_dev from a driver

## Synopsis

```
struct iio_dev * iio_device_alloc (int sizeof_priv);
```

## Arguments

*sizeof_priv* Space to allocate for private structure.

# Name

iio_device_free — free an iio_dev from a driver

## Synopsis

```
void iio_device_free (struct iio_dev * dev);
```

## Arguments

*dev* the iio_dev associated with the device

# Name

iio_device_register — register a device with the IIO subsystem

## Synopsis

```
int iio_device_register (struct iio_dev * indio_dev);
```

## Arguments

*indio_dev* Device structure filled by the device driver

# Name

iio_device_unregister — unregister a device from the IIO subsystem

## Synopsis

```
void iio_device_unregister (struct iio_dev * indio_dev);
```

## Arguments

*indio_dev* Device structure representing the device.

An IIO device usually corresponds to a single hardware sensor and it provides all the information needed by a driver handling a device. Let's first have a look at the functionality embedded in an IIO device then we will show how a device driver makes use of an IIO device.

There are two ways for a user space application to interact with an IIO driver.

- `/sys/bus/iio/iio:deviceX/`, this represents a hardware sensor and groups together the data channels of the same chip.
- `/dev/iio:deviceX`, character device node interface used for buffered data transfer and for events information retrieval.

A typical IIO driver will register itself as an I2C or SPI driver and will create two routines, `probe` and `remove` . At `probe`:

- call `iio_device_alloc`, which allocates memory for an IIO device.
- initialize IIO device fields with driver specific information (e.g. device name, device channels).
- call `iio_device_register`, this registers the device with the IIO core. After this call the device is ready to accept requests from user space applications.

At `remove`, we free the resources allocated in `probe` in reverse order:

- `iio_device_unregister`, unregister the device from the IIO core.
- `iio_device_free`, free the memory allocated for the IIO device.

# IIO device sysfs interface

Attributes are sysfs files used to expose chip info and also allowing applications to set various configuration parameters. For device with index X, attributes can be found under `/sys/bus/iio/iio:deviceX/` directory. Common attributes are:

- `name`, description of the physical chip.
- `dev`, shows the major:minor pair associated with `/dev/iio:deviceX` node.
- `sampling_frequency_available`, available discrete set of sampling frequency values for device.

Available standard attributes for IIO devices are described in the `Documentation/ABI/testing/sysfs-bus-iio` file in the Linux kernel sources.

# IIO device channels

# Name

struct iio_chan_spec — specification of a single channel

## Synopsis

```
struct iio_chan_spec {
  enum iio_chan_type type;
  int channel;
  int channel2;
  unsigned long address;
  int scan_index;
  struct scan_type;
  long info_mask_separate;
  long info_mask_shared_by_type;
  long info_mask_shared_by_dir;
  long info_mask_shared_by_all;
  const struct iio_event_spec * event_spec;
  unsigned int num_event_specs;
  const struct iio_chan_spec_ext_info * ext_info;
  const char * extend_name;
  const char * datasheet_name;
  unsigned modified:1;
  unsigned indexed:1;
  unsigned output:1;
  unsigned differential:1;
};
```

## Members

| | |
|---|---|
| type | What type of measurement is the channel making. |
| channel | What number do we wish to assign the channel. |
| channel2 | If there is a second number for a differential channel then this is it. If modified is set then the value here specifies the modifier. |
| address | Driver specific identifier. |
| scan_index | Monotonic index to give ordering in scans when read from a buffer. |
| scan_type | Sign: 's' or 'u' to specify signed or unsigned |
| info_mask_separate | What information is to be exported that is specific to this channel. |
| info_mask_shared_by_type | What information is to be exported that is shared by all channels of the same type. |
| info_mask_shared_by_dir | What information is to be exported that is shared by all channels of the same direction. |
| info_mask_shared_by_all | What information is to be exported that is shared by all channels. |
| event_spec | Array of events which should be registered for this channel. |
| num_event_specs | Size of the event_spec array. |

| | |
|---|---|
| ext_info | Array of extended info attributes for this channel. The array is NULL terminated, the last element should have its name field set to NULL. |
| extend_name | Allows labeling of channel attributes with an informative name. Note this has no effect codes etc, unlike modifiers. |
| datasheet_name | A name used in in-kernel mapping of channels. It should correspond to the first name that the channel is referred to by in the datasheet (e.g. IND), or the nearest possible compound name (e.g. IND-INC). |
| modified | Does a modifier apply to this channel. What these are depends on the channel type. Modifier is set in channel2. Examples are IIO_MOD_X for axial sensors about the 'x' axis. |
| indexed | Specify the channel has a numerical index. If not, the channel index number will be suppressed for sysfs attributes but not for event codes. |
| output | Channel is output. |
| differential | Channel is differential. |

## realbits

Number of valid bits of data

## storage_bits

Realbits + padding

## shift

Shift right by this before masking out realbits.

## endianness

little or big endian

## repeat

Number of times real/storage bits repeats. When the repeat element is more than 1, then the type element in sysfs will show a repeat value. Otherwise, the number of repetitions is omitted.

An IIO device channel is a representation of a data channel. An IIO device can have one or multiple channels. For example:

- a thermometer sensor has one channel representing the temperature measurement.
- a light sensor with two channels indicating the measurements in the visible and infrared spectrum.
- an accelerometer can have up to 3 channels representing acceleration on X, Y and Z axes.

An IIO channel is described by the struct iio_chan_spec . A thermometer driver for the temperature sensor in the example above would have to describe its channel as follows:

```
static const struct iio_chan_spec temp_channel[] = {
    {
        .type = IIO_TEMP,
        .info_mask_separate = BIT(IIO_CHAN_INFO_PROCESSED),
    },
};
```

Channel sysfs attributes exposed to userspace are specified in the form of *bitmasks*. Depending on their shared info, attributes can be set in one of the following masks:

- *info_mask_separate*, attributes will be specific to this channel
- *info_mask_shared_by_type*, attributes are shared by all channels of the same type
- *info_mask_shared_by_dir*, attributes are shared by all channels of the same direction
- *info_mask_shared_by_all*, attributes are shared by all channels

When there are multiple data channels per channel type we have two ways to distinguish between them:

- set *.modified* field of iio_chan_spec to 1. Modifiers are specified using *.channel2* field of the same iio_chan_spec structure and are used to indicate a physically unique characteristic of the channel such as its direction or spectral response. For example, a light sensor can have two channels, one for infrared light and one for both infrared and visible light.
- set *.indexed* field of iio_chan_spec to 1. In this case the channel is simply another instance with an index specified by the *.channel* field.

Here is how we can make use of the channel's modifiers:

```
static const struct iio_chan_spec light_channels[] = {
    {
        .type = IIO_INTENSITY,
        .modified = 1,
        .channel2 = IIO_MOD_LIGHT_IR,
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
        .info_mask_shared = BIT(IIO_CHAN_INFO_SAMP_FREQ),
    },
    {
        .type = IIO_INTENSITY,
        .modified = 1,
        .channel2 = IIO_MOD_LIGHT_BOTH,
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
        .info_mask_shared = BIT(IIO_CHAN_INFO_SAMP_FREQ),
    },
    {
        .type = IIO_LIGHT,
        .info_mask_separate = BIT(IIO_CHAN_INFO_PROCESSED),
        .info_mask_shared = BIT(IIO_CHAN_INFO_SAMP_FREQ),
    },

}
```

This channel's definition will generate two separate sysfs files for raw data retrieval:

- `/sys/bus/iio/iio:deviceX/in_intensity_ir_raw`
- `/sys/bus/iio/iio:deviceX/in_intensity_both_raw`

one file for processed data:

- `/sys/bus/iio/iio:deviceX/in_illuminance_input`

and one shared sysfs file for sampling frequency:

- `/sys/bus/iio/iio:deviceX/sampling_frequency.`

Here is how we can make use of the channel's indexing:

```
static const struct iio_chan_spec light_channels[] = {
    {
        .type = IIO_VOLTAGE,
        .indexed = 1,
        .channel = 0,
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
    },
    {
        .type = IIO_VOLTAGE,
        .indexed = 1,
        .channel = 1,
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
    },
}
```

This will generate two separate attributes files for raw data retrieval:

- `/sys/bus/iio/devices/iio:deviceX/in_voltage0_raw`, representing voltage measurement for channel 0.
- `/sys/bus/iio/devices/iio:deviceX/in_voltage1_raw`, representing voltage measurement for channel 1.

# Industrial I/O buffers

# Name

struct iio_buffer — general buffer structure

## Synopsis

```
struct iio_buffer {
  int length;
  int bytes_per_datum;
  struct attribute_group * scan_el_attrs;
  long * scan_mask;
  bool scan_timestamp;
  const struct iio_buffer_access_funcs * access;
  struct list_head scan_el_dev_attr_list;
  struct attribute_group scan_el_group;
  wait_queue_head_t pollq;
  bool stufftoread;
  struct list_head demux_list;
  void * demux_bounce;
  struct list_head buffer_list;
  struct kref ref;
  unsigned int watermark;
};
```

## Members

| | |
|---|---|
| length | [DEVICE] number of datums in buffer |
| bytes_per_datum | [DEVICE] size of individual datum including timestamp |
| scan_el_attrs | [DRIVER] control of scan elements if that scan mode control method is used |
| scan_mask | [INTERN] bitmask used in masking scan mode elements |
| scan_timestamp | [INTERN] does the scan mode include a timestamp |
| access | [DRIVER] buffer access functions associated with the implementation. |
| scan_el_dev_attr_list | [INTERN] list of scan element related attributes. |
| scan_el_group | [DRIVER] attribute group for those attributes not created from the iio_chan_info array. |
| pollq | [INTERN] wait queue to allow for polling on the buffer. |
| stufftoread | [INTERN] flag to indicate new data. |
| demux_list | [INTERN] list of operations required to demux the scan. |
| demux_bounce | [INTERN] buffer for doing gather from incoming scan. |
| buffer_list | [INTERN] entry in the devices list of current buffers. |
| ref | [INTERN] reference count of the buffer. |

| | |
|---|---|
| watermark | [INTERN] number of datums to wait for poll/read. |

# Name

iio_validate_scan_mask_onehot — Validates that exactly one channel is selected

## Synopsis

```
bool iio_validate_scan_mask_onehot (struct iio_dev * indio_dev, const
unsigned long * mask);
```

## Arguments

*indio_dev* the iio device

*mask* scan mask to be checked

## Description

Return true if exactly one bit is set in the scan mask, false otherwise. It can be used for devices where only one channel can be active for sampling at a time.

# Name

iio_buffer_get — Grab a reference to the buffer

## Synopsis

```
struct iio_buffer * iio_buffer_get (struct iio_buffer * buffer);
```

## Arguments

*buffer* The buffer to grab a reference for, may be NULL

## Description

Returns the pointer to the buffer that was passed into the function.

# Name

iio_buffer_put — Release the reference to the buffer

## Synopsis

```
void iio_buffer_put (struct iio_buffer * buffer);
```

## Arguments

*buffer* The buffer to release the reference for, may be NULL

The Industrial I/O core offers a way for continuous data capture based on a trigger source. Multiple data channels can be read at once from `/dev/iio:deviceX` character device node, thus reducing the CPU load.

# IIO buffer sysfs interface

An IIO buffer has an associated attributes directory under `/sys/bus/iio/iio:deviceX/buffer/`. Here are the existing attributes:

- *length*, the total number of data samples (capacity) that can be stored by the buffer.
- *enable*, activate buffer capture.

# IIO buffer setup

The meta information associated with a channel reading placed in a buffer is called a *scan element* . The important bits configuring scan elements are exposed to userspace applications via the `/sys/bus/iio/iio:deviceX/scan_elements/` directory. This file contains attributes of the following form:

- *enable*, used for enabling a channel. If and only if its attribute is non zero, then a triggered capture will contain data samples for this channel.
- *type*, description of the scan element data storage within the buffer and hence the form in which it is read from user space. Format is *[be|le]:[s|u]bits/storagebitsXrepeat[>>shift]* .
  - *be* or *le*, specifies big or little endian.
  - *s* or *u*, specifies if signed (2's complement) or unsigned.
  - *bits*, is the number of valid data bits.
  - *storagebits*, is the number of bits (after padding) that it occupies in the buffer.
  - *shift*, if specified, is the shift that needs to be applied prior to masking out unused bits.
  - *repeat*, specifies the number of bits/storagebits repetitions. When the repeat element is 0 or 1, then the repeat value is omitted.

For example, a driver for a 3-axis accelerometer with 12 bit resolution where data is stored in two 8-bits registers as follows:

```
7   6   5   4   3   2   1   0
```

```
+---+---+---+---+---+---+---+---+
|D3 |D2 |D1 |D0 | X | X | X | X | (LOW byte, address 0x06)
+---+---+---+---+---+---+---+---+

  7   6   5   4   3   2   1   0
+---+---+---+---+---+---+---+---+
|D11|D10|D9 |D8 |D7 |D6 |D5 |D4 | (HIGH byte, address 0x07)
+---+---+---+---+---+---+---+---+
```

will have the following scan element type for each axis:

```
$ cat /sys/bus/iio/devices/iio:device0/scan_elements/in_accel_y_type
le:s12/16>>4
```

A user space application will interpret data samples read from the buffer as two byte little endian signed data, that needs a 4 bits right shift before masking out the 12 valid bits of data.

For implementing buffer support a driver should initialize the following fields in iio_chan_spec definition:

```
struct iio_chan_spec {
    /* other members */
    int scan_index
    struct {
        char sign;
        u8 realbits;
        u8 storagebits;
        u8 shift;
        u8 repeat;
        enum iio_endian endianness;
    } scan_type;
};
```

The driver implementing the accelerometer described above will have the following channel definition:

```
struct struct iio_chan_spec accel_channels[] = {
    {
      .type = IIO_ACCEL,
      .modified = 1,
      .channel2 = IIO_MOD_X,
      /* other stuff here */
      .scan_index = 0,
      .scan_type = {
        .sign = 's',
        .realbits = 12,
        .storgebits = 16,
        .shift = 4,
        .endianness = IIO_LE,
      },
  }
```

```
        /* similar for Y (with channel2 = IIO_MOD_Y, scan_index = 1)
         * and Z (with channel2 = IIO_MOD_Z, scan_index = 2) axis
         */
}
```

Here *scan_index* defines the order in which the enabled channels are placed inside the buffer. Channels with a lower scan_index will be placed before channels with a higher index. Each channel needs to have a unique scan_index.

Setting scan_index to -1 can be used to indicate that the specific channel does not support buffered capture. In this case no entries will be created for the channel in the scan_elements directory.

# Industrial I/O triggers

# Name

struct iio_trigger — industrial I/O trigger device

## Synopsis

```
struct iio_trigger {
  const struct iio_trigger_ops * ops;
  int id;
  const char * name;
  struct device dev;
  struct list_head list;
  struct list_head alloc_list;
  atomic_t use_count;
  struct irq_chip subirq_chip;
  int subirq_base;
  struct iio_subirq subirqs[CONFIG_IIO_CONSUMERS_PER_TRIGGER];
  unsigned long pool[BITS_TO_LONGS(CONFIG_IIO_CONSUMERS_PER_TRIGGER)];
  struct mutex pool_lock;
};
```

## Members

| | |
|---|---|
| ops | [DRIVER] operations structure |
| id | [INTERN] unique id number |
| name | [DRIVER] unique name |
| dev | [DRIVER] associated device (if relevant) |
| list | [INTERN] used in maintenance of global trigger list |
| alloc_list | [DRIVER] used for driver specific trigger list |
| use_count | use count for the trigger |
| subirq_chip | [INTERN] associate 'virtual' irq chip. |
| subirq_base | [INTERN] base number for irqs provided by trigger. |
| subirqs[CONFIG_IIO_CONSUMERS_PER_TRIGGER] | [INTERN] information about the 'child' irqs. |
| pool[BITS_TO_LONGS(CONFIG_IIO_CONSUMERS_PER_TRIGGER)] | [INTERN] bitmap of irqs currently in use. |
| pool_lock | [INTERN] protection of the irq pool. |

# Name

devm_iio_trigger_alloc — Resource-managed `iio_trigger_alloc`

## Synopsis

```
struct iio_trigger * devm_iio_trigger_alloc (struct device * dev, const
char * fmt, ...);
```

## Arguments

*`dev`* Device to allocate iio_trigger for

*`fmt`* trigger name format. If it includes format specifiers, the additional arguments following format are formatted and inserted in the resulting string replacing their respective specifiers.

*`...`* variable arguments

## Description

Managed iio_trigger_alloc. iio_trigger allocated with this function is automatically freed on driver detach.

If an iio_trigger allocated with this function needs to be freed separately, `devm_iio_trigger_free` must be used.

## RETURNS

Pointer to allocated iio_trigger on success, NULL on failure.

# Name

devm_iio_trigger_free — Resource-managed `iio_trigger_free`

## Synopsis

```
void devm_iio_trigger_free (struct device * dev, struct iio_trigger *
iio_trig);
```

## Arguments

*`dev`* Device this iio_dev belongs to

*`iio_trig`* the iio_trigger associated with the device

## Description

Free iio_trigger allocated with `devm_iio_trigger_alloc`.

In many situations it is useful for a driver to be able to capture data based on some external event (trigger) as opposed to periodically polling for data. An IIO trigger can be provided by a device driver that also has an IIO device based on hardware generated events (e.g. data ready or threshold exceeded) or provided by a separate driver from an independent interrupt source (e.g. GPIO line connected to some external system, timer interrupt or user space writing a specific file in sysfs). A trigger may initiate data capture for a number of sensors and also it may be completely unrelated to the sensor itself.

# IIO trigger sysfs interface

There are two locations in sysfs related to triggers:

- `/sys/bus/iio/devices/triggerY`, this file is created once an IIO trigger is registered with the IIO core and corresponds to trigger with index Y. Because triggers can be very different depending on type there are few standard attributes that we can describe here:
  - *name*, trigger name that can be later used for association with a device.
  - *sampling_frequency*, some timer based triggers use this attribute to specify the frequency for trigger calls.
- `/sys/bus/iio/devices/iio:deviceX/trigger/`, this directory is created once the device supports a triggered buffer. We can associate a trigger with our device by writing the trigger's name in the `current_trigger` file.

# IIO trigger setup

Let's see a simple example of how to setup a trigger to be used by a driver.

```
struct iio_trigger_ops trigger_ops = {
    .set_trigger_state = sample_trigger_state,
    .validate_device = sample_validate_device,
}

struct iio_trigger *trig;
```

```
/* first, allocate memory for our trigger */
trig = iio_trigger_alloc(dev, "trig-%s-%d", name, idx);

/* setup trigger operations field */
trig->ops = &trigger_ops;

/* now register the trigger with the IIO core */
iio_trigger_register(trig);
```

# IIO trigger ops

# Name

struct iio_trigger_ops — operations structure for an iio_trigger.

## Synopsis

```
struct iio_trigger_ops {
  struct module * owner;
  int (* set_trigger_state) (struct iio_trigger *trig, bool state);
  int (* try_reenable) (struct iio_trigger *trig);
  int (* validate_device) (struct iio_trigger *trig,struct iio_dev *indio_dev);
};
```

## Members

| | |
|---|---|
| owner | used to monitor usage count of the trigger. |
| set_trigger_state | switch on/off the trigger on demand |
| try_reenable | function to reenable the trigger when the use count is zero (may be NULL) |
| validate_device | function to validate the device when the current trigger gets changed. |

## Description

This is typically static const within a driver and shared by instances of a given device.

Notice that a trigger has a set of operations attached:

- `set_trigger_state`, switch the trigger on/off on demand.
- `validate_device`, function to validate the device when the current trigger gets changed.

# Industrial I/O triggered buffers

Now that we know what buffers and triggers are let's see how they work together.

# IIO triggered buffer setup

# Name

iio_triggered_buffer_setup — Setup triggered buffer and pollfunc

## Synopsis

```
int iio_triggered_buffer_setup (struct iio_dev * indio_dev, irqreturn_t
(*h) (int irq, void *p), irqreturn_t (*thread) (int irq, void *p), const
struct iio_buffer_setup_ops * setup_ops);
```

## Arguments

*indio_dev* IIO device structure

*h* Function which will be used as pollfunc top half

*thread* Function which will be used as pollfunc bottom half

*setup_ops* Buffer setup functions to use for this device. If NULL the default setup functions for triggered buffers will be used.

## Description

This function combines some common tasks which will normally be performed when setting up a triggered buffer. It will allocate the buffer and the pollfunc.

Before calling this function the indio_dev structure should already be completely initialized, but not yet registered. In practice this means that this function should be called right before `iio_device_register`.

To free the resources allocated by this function call `iio_triggered_buffer_cleanup`.

# Name

iio_triggered_buffer_cleanup — Free resources allocated by `iio_triggered_buffer_setup`

## Synopsis

```
void iio_triggered_buffer_cleanup (struct iio_dev * indio_dev);
```

## Arguments

*`indio_dev`* IIO device structure

# Name

struct iio_buffer_setup_ops — buffer setup related callbacks

## Synopsis

```
struct iio_buffer_setup_ops {
  int (* preenable) (struct iio_dev *);
  int (* postenable) (struct iio_dev *);
  int (* predisable) (struct iio_dev *);
  int (* postdisable) (struct iio_dev *);
  bool (* validate_scan_mask) (struct iio_dev *indio_dev,const unsigned long *scan
};
```

## Members

| | |
|---|---|
| preenable | [DRIVER] function to run prior to marking buffer enabled |
| postenable | [DRIVER] function to run after marking buffer enabled |
| predisable | [DRIVER] function to run prior to marking buffer disabled |
| postdisable | [DRIVER] function to run after marking buffer disabled |
| validate_scan_mask | [DRIVER] function callback to check whether a given scan mask is valid for the device. |

A typical triggered buffer setup looks like this:

```
const struct iio_buffer_setup_ops sensor_buffer_setup_ops = {
  .preenable    = sensor_buffer_preenable,
  .postenable   = sensor_buffer_postenable,
  .postdisable  = sensor_buffer_postdisable,
  .predisable   = sensor_buffer_predisable,
};

irqreturn_t sensor_iio_pollfunc(int irq, void *p)
{
    pf->timestamp = iio_get_time_ns();
    return IRQ_WAKE_THREAD;
}

irqreturn_t sensor_trigger_handler(int irq, void *p)
{
    u16 buf[8];
    int i = 0;

    /* read data for each active channel */
    for_each_set_bit(bit, active_scan_mask, masklength)
        buf[i++] = sensor_get_data(bit)

    iio_push_to_buffers_with_timestamp(indio_dev, buf, timestamp);

    iio_trigger_notify_done(trigger);
```

```
    return IRQ_HANDLED;
}

/* setup triggered buffer, usually in probe function */
iio_triggered_buffer_setup(indio_dev, sensor_iio_polfunc,
                           sensor_trigger_handler,
                           sensor_buffer_setup_ops);
```

The important things to notice here are:

- `iio_buffer_setup_ops`, the buffer setup functions to be called at predefined points in the buffer configuration sequence (e.g. before enable, after disable). If not specified, the IIO core uses the default iio_triggered_buffer_setup_ops.
- `sensor_iio_pollfunc`, the function that will be used as top half of poll function. It should do as little processing as possible, because it runs in interrupt context. The most common operation is recording of the current timestamp and for this reason one can use the IIO core defined `iio_pollfunc_store_time` function.
- `sensor_trigger_handler`, the function that will be used as bottom half of the poll function. This runs in the context of a kernel thread and all the processing takes place here. It usually reads data from the device and stores it in the internal buffer together with the timestamp recorded in the top half.

# Chapter 3. Resources

- `drivers/iio/`, contains the IIO core plus and directories for each sensor type (e.g. accel, magnetometer, etc.)
- `include/linux/iio/`, contains the header files, nice to read for the internal kernel interfaces.
- `include/uapi/linux/iio/`, contains files to be used by user space applications.
- `tools/iio/`, contains tools for rapidly testing buffers, events and device creation.
- `drivers/staging/iio/`, contains code for some drivers or experimental features that are not yet mature enough to be moved out.

Besides the code, there are some good online documentation sources:

- Industrial I/O mailing list [http://marc.info/?l=linux-iio]
- Analog Device IIO wiki page [http://wiki.analog.com/software/linux/docs/iio/iio]
- Using the Linux IIO framework for SDR, Lars-Peter Clausen's presentation at FOSDEM [https://fosdem.org/2015/schedule/event/iiosdr/]