1) Short introduction to IIO
2) Motivation for libiio
3) Using libiio
4) Infrastructure built with and around libiio
5) Future outlook
Short Introduction to IIO
Introduction to IIO

- Industrial Input/Output framework
  - Not really just for Industrial IO
  - All non-HID IO
    - ADC, DAC, light, accelerometer, gyro, magnetometer, humidity, temperature, rotation, angular momentum, lifestyle sensors ...
- Developed by Jonathan Cameron
- In the kernel since v2.6.32 (2009)
- Moved out of staging/ in v3.5 (2012)
- ~220 IIO device drivers (v4.6)
  - Many drivers support multiple devices
IIO Structure

- Device represents logical functional unit
  - Typically a piece of physical hardware
- Attributes
  - Describe hardware capabilities
  - Allow to change hardware configuration
IIO Structure

• Channels represent data channels
  – Channels have a type and direction
  – E.g. ADC has voltage channels
  – Channels can have attributes

• Buffers are used for continuous data capture
static const struct iio_chan_spec adc_channels[] = {
    {
        .type = IIO_VOLTAGE,
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW),
        .info_mask_shared_by_type = BIT(IIO_CHAN_INFO_SCALE),
        .indexed = 1,
        .channel = 0,
        .scan_index = 0,
    },
    ...,
    {
        .type = IIO_TEMP,
        .info_mask_separate = BIT(IIO_CHAN_INFO_RAW) | BIT(IIO_CHAN_INFO_AVERAGE_RAW) | BIT(IIO_CHAN_INFO_SCALE),
        .indexed = 1,
        .channel = 0,
        .scan_index = 8,
    }
};
static const struct iio_info adc_info = {
    .read_raw = &adc_read_raw,
    .write_raw = &adc_write_raw,
    .driver_module = THIS_MODULE,
};

struct iio_dev *indio_dev;

indio_dev = iio_device_alloc(0);
indio_dev->name = "adc123";
indio_dev->channels = adc_channels;
indio_dev->num_channels = ARRAY_SIZE(adc_channels);
indio_dev->info = &adc_info;
indio_dev->dev.parent = dev;
indio_dev->modes = INDIO_DIRECT_MODE | INDIO_BUFFER_SOFTWARE;

iio_device_register(indio_dev);
IIO Userspace ABI

• Devices, channels and attributes are represented as sysfs directories and files
  – sysfs is a virtual filesystem where read/write operations are mapped to kernel callbacks
• Buffers are represented as character devices
  – Use read()/write() to access data
```bash
# cd /sys/bus/iio/devices
# ls
iio:device0  iio:device1  iio:device2
#
# cd iio:device0
# ls
buffer/
in_voltage0_raw  in_voltage5_raw
power
in_voltage1_raw  in_voltage6_raw
scan_elements/
in_temp0_mean_raw  in_voltage2_raw  in_voltage7_raw
subsystem
in_temp0_raw       in_voltage3_raw  in_voltage_scale
uevent
in_temp0_scale     in_voltage4_raw  name
# cat in_temp0_raw
2013
# cat in_temp0_scale
12.5
```
# cd /sys/bus/iio/devices/iio:device1/scan_elements
# ls
in_voltage0_en in_voltage0_index in_voltage0_type ...
# cat in_voltage0_type
be:u12/16>>0
# cd /sys/bus/iio/devices/iio:device1/buffer
# ls
enable length
# cd /sys/bus/iio/devices/iio:device1/
# echo 1 > scan_elements/in_voltage0_en
# echo 1024 > buffer/length
# echo 1 > buffer/enable
# cat /dev/iio:device0 | ... 
# echo 0 > buffer/length
IIO Userspace ABI – Writing Applications

- Looks all nice and good...
- … until you try to use it in an application
  - Involves a lot of string parsing and formatting
  - Structured data from the kernel driver has been flattened
- String parsing is not easy and error prone (especially in a language like C)
- Applications require lot of boilerplate code
libiio Design Goals
libiio Design Goals

• Hide low level details of communicating with the kernel driver
  – Take care of all boilerplate code
• Provide proper data structures and functions
  – Reconstruct kernel driver data structures
• Support for (remote) backends
  – Allow applications to access the devices when running on a remote machine (e.g. laptop connected to embedded board)
libiio Design Goals

• Support for (remote) backends
  – Allow applications to access the devices when running on a remote machine (e.g. laptop connected to embedded board)
  – Have a system daemon that serializes and multiplexes access to the same device for multiple application
About libiio
About libiio

- Development started: Beginning of 2014
- First stable release: August of 2014
- Two stable releases per year
- Maintainer and lead developer: Paul Cercueil
- Written in the C programming language
- Stable ABI guarantee
- Under active development
  - Patches welcome
Using libiio
libiio itself has zero global state

All state is contained in a context
  - Multiple contexts can be instantiated

Context can be local or remote
  - `iio_create_local_context(void)`
  - `iio_create_network_context(const char *host)`
  - `iio_create_default_context(void)`
    - Looks up the target context from the IIOD_REMOTE environment variable
  - `iio_context_destroy()` to free context state
Devices

- `struct iio_device` maps to a device registered by the kernel
- `iio_context_get_devices_count(struct iio_context *)`  
  `iio_context_get_device(struct iio_context *, unsigned int index)`  
  - Enumerate all available devices of a context
- `iio_context_find_device(struct iio_context *, const char *name)`  
  - Lookup device by ID (iio:deviceX) or name
Channels

- `struct iio_channel` maps to a channel of a device
- `iio_device_get_channels_count(struct iio_device *)
  iio_device_get_channel(struct iio_device *, unsigned int index)`
  - Enumerate all available channels of a device
- `iio_device_find_channel(struct iio_device *, const char *name, bool output)`
  - Lookup channel of a device by ID (e.g. voltage0) or name
  - Input and output channels can have overlapping IDs
Attributes

- `const char *` used to represent attribute names
- `iio_device_get_attrs_count(struct iio_device *)`
- `iio_device_get_attr(struct iio_device *, unsigned int index)`
- `iio_channel_get_attrs_count(struct iio_channel *)`
- `iio_channel_get_attr(struct iio_channel *, unsigned int index)`
  - Enumerate available attributes
Attributes

- `iio_device_find_attr(const char *name)`
- `iio_channel_find_attr(const char *name)`
  - Lookup attribute by name
  - Can be used to check if attribute exist
  - Returned string is valid as long as context is valid
Attributes

- `iio_{device,channel}_attr_read(struct iio_{device,channel} *, const char *attr, char *dst, size_t len)`

- `iio_{device,channel}_attr_read_bool(struct iio_{device,channel} *, const char *attr, bool *val)`

- `iio_{device,channel}_attr_read_double(struct iio_{device,channel} *, const char *attr, double *val)`

- `iio_{device,channel}_attr_read_longlong(struct iio_{device,channel} *, const char *attr, long long *val)`

- Get the value of a attribute
- String value converted to the target data type
Attributes

- `iio_{device,channel}_attr_write(struct iio_{device,channel} *, const char *attr, const char *src)`
  - Set the value of a attribute
- `iio_{device,channel}_attr_write_bool(struct iio_{device,channel} *, const char *attr, bool val)`
- `iio_{device,channel}_attr_write_double(struct iio_{device,channel} *, const char *attr, double val)`
- `iio_{device,channel}_attr_write_longlong(struct iio_{device,channel} *, const char *attr, long long val)`
  - Source data type converted to string value
Buffers

- `iio_channel_enable(struct iio_channel *)`
  - Enable/Disable channel for buffered capture
- `iio_channel_disable(struct iio_channel *)`
- `struct iio_buffer *` represents an active buffer
- `iio_device_create_buffer(struct iio_device *, size_t size, bool cyclic)`
  - Configures **and** enables buffer
- `iio_buffer_destroy(struct iio_buffer *)`
  - Disables buffer and frees data structure
Buffers

- `iio_buffer_refill(struct iio_buffer *)`
  - Fetches samples from the kernel buffer
- `iio_buffer_start(struct iio_buffer *)`
  - Returns the address of the userspace buffer
  - Might change after `iio_buffer_refill()`
- `iio_buffer_step(struct iio_buffer *)`
  - Spacing between sample sets in the buffer
- `iio_buffer_first(struct iio_buffer *, struct iio_channel *)`
  - Returns the address of the first sample for a channel
Example

```c
struct iio_context *ctx;
struct iio_device *dev;
struct iio_channel *ch;

/* Error handling is missing */
ctx = iio_create_default_context();
dev = iio_context_get_device(ctx, 0);
ch = iio_device_get_channel(dev, 0);

iio_device_attr_write_longlong(dev, "sample_rate", 1000);
iio_channel_attr_write_double(ch, "scale", 0.525);
```
Example – Data Capture

```c
uint16_t *data;
struct iio_buffer *buf;

iio_channel_enable(chn);
buf = iio_device_create_buffer(dev, 1000, false);
iio_buffer_refill(buf);
for (data = iio_buffer_first(buf, ch);
     data < iio_buffer_end(buf);
     data += iio_buffer_step(buf))
    printf("%u\n", *data);

iio_buffer_destroy(buf);
iio_channel_disable(chn);
```
Bindings
• Bindings are available for multiple programming languages
  - Python, C#, Matlab, C++ (experimental)
• Cross-platform
  - Linux (native and remote backends)
  - Windows, MacOS X, BSDs (remote backends)
iiiod
• System service
• Multiplexing between multiple readers/writers
• Support for remote clients (via TCP/IP and USB)
• Applications do not need system level privileges
• Transparent from the applications point of view
• Allows client state tracking
iiiod and libiiio

Client Application on Linux
- LibIIIO / Linux
- Local backend
- Network backend
- High-level API
- Linux Kernel
- IIO devices

IIOD Server

Client Application on Windows
- LibIIIO / Windows
- Network backend
- High-level API

Network link
Tools
• List information about all available device
• Prints snapshot of all devices and all their channels and attributes

# iio_info
Library version: 0.6 (git tag: 284b224)
IIO context created with local backend.
Backend version: 0.6 (git tag: 284b224)
Backend description string: Linux analog 3.19.0-gf733099 #1 SMP PREEMPT Mon Nov 2 11:05:07
EET 2015 armv7I
IIO context has 5 devices:
  iio:device0: ad7291
    9 channels found:
      temp0: (input)
      3 channel-specific attributes found:
        attr 0: scale value: 250
        attr 1: mean_raw value: 110
        attr 2: raw value: 109
    voltage0: (input)
    2 channel-specific attributes found:
      attr 0: raw value: 2512
      attr 1: scale value: 0.610351562
iio_readdev

- Allows to capture continuous data from a device

```bash
# iio_readdev --buffer-size 100000 iio:device4 voltage0 | pv > /dev/null
584MB 0:00:10 [58.6MB/s]
```
iio_monitor

- Digital multimeter type application
- ncurses based interface
- Useful for having a look at “live” data
IIO Scope

- Capture and display data
  - Time domain
  - Frequency domain
  - Constellation plot
  - Markers
  - Math operations
- Device configuration
- Plug-in system allow to create device or complex specialized GUI
Future
Future Developments – Short Term

- **USB remote backend support**
  - Implemented as a gadget driver using function `fs`
  - Allows embedded data aggregation devices to directly connect to PC/laptop

- **Support for backend enumeration**
  - Applications can offer a list of available backends to user
  - Reference backends by URI (e.g. `usb://3-25`, `local://`)
    - New `iio_create_context_from_uri()` function
Future Developments – Long Term

- Hotplug support
  - So far most platforms with IIO devices have a static setup
  - DeviceTree overlays allow dynamic insertion/removal
  - IIO recently gained support for user insatiable triggers
- Make remote backend support independent of IIO
  - Allows support for other data sources (e.g. hwmon, storage)
  - Allows to implement application logic on the target side, but outside of kernel space
Q/A
Thanks
Further Information

- **Source**
  - https://github.com/analogdevicesinc/libiio

- **API reference**
  - http://analogdevicesinc.github.io/libiio/

- **Design document**
Bonus Slides
3rd Party Tools Integration
GNURadio

- Signal-processing development environment
- Many diverse pre-built processing blocks available
- Processing pipelines are assembled from blocks in flow graphs
- GnuRadio IIO Sink and Source blocks are available through the gr-iio package
sigrok (WIP)

- Portable, cross-platform, Open-Source signal analysis software (logic analyzers, scopes, multimeters, and more)
  - Protocol decoders I2C, SPI, UART ...
- Generic IIO sigrok driver allows to capture data from any device
- Specialized drivers allow to provide better configuration mapping between IIO and sigrok
Matlab/Simulink

• IIO System Object
  – Based on MATLAB System Objects
  – Available in both Matlab and Simulink
  – Data is streamed over a remote backend into the simulation
  – Control settings
• Enables hardware in the loop simulation