LIBIIO

A library for interfacing with Linux IIO devices

(27 October 2020)

About the presenter

Name: Dan Nechita

Job: Software Development Engineer

Company: Analog Devices Inc.

Role: One of the maintainers of LibIIO code

Summary

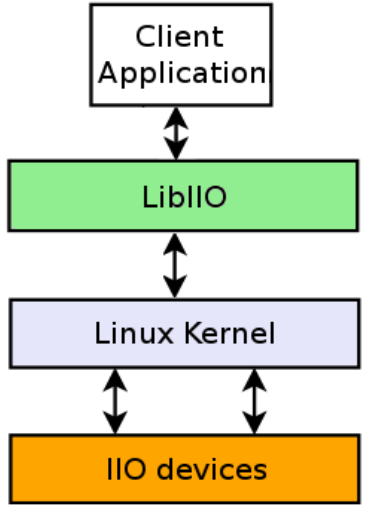
1. What is LibIIO? ⟸
2. A look at the library structure
3. The C language API
4. LibIIO bindings
5. Practices that aim for a robust library

1. What is LibIIO?

- LibIIO:
  - User-space library
  - Around for more than 6 years and consistently being improved
  - Written in C language
  - Cross platform: Linux, Windows, Mac OS
  - License: LGPL version 2.1+

1. What is LibIIO?

- The purpose:
  - To make it easier and faster to develop applications that need to interact with Linux Industrial I/O (IIO) devices



1. What is LibIIO? (Note on IIO)

- Quick note about Linux Industrial I/O (IIO)
  - It is part of the Linux Kernel and it is a subsystem that provides support for devices such as:
    - Analog-to-digital converters (ADCs)
    - Digital-to-analog converters (DACs)
    - Accelerometers
    - Inertial measurement units (IMUs), etc.
  - More at:
    - https://www.kernel.org/doc/html/v5.9/driver-api/iio/intro.html
    - https://wiki.analog.com/software/linux/docs/iio/iio

ANALOG DEVICES
AHEAD OF WHAT'S POSSIBLE™

THE LINUX FOUNDATION

1. What is LibIIO?

The API of Industrial I/O subsystem is exposed through sysfs at location:

- /sys/bus/iio/devices/*

1. What is LibIIO?

An example of IIO device:

```
1:  /sys/bus/iio/devices/iio:device0/name
2:  /sys/bus/iio/devices/iio:device0/out_voltage0_V1_raw
3:  /sys/bus/iio/devices/iio:device0/out_voltage0_V1_scale
4:  /sys/bus/iio/devices/iio:device0/out_voltage0_V1_powerdown
5:  /sys/bus/iio/devices/iio:device0/out_voltage0_V1_powerdown_mode
6:  /sys/bus/iio/devices/iio:device0/out_voltage1_V2_raw
7:  /sys/bus/iio/devices/iio:device0/out_voltage1_V2_scale
8:  /sys/bus/iio/devices/iio:device0/out_voltage1_V2_powerdown
9:  /sys/bus/iio/devices/iio:device0/out_voltage1_V2_powerdown_mode
10:  /sys/bus/iio/devices/iio:device0/out_voltage_powerdown_mode_available
11:  /sys/bus/iio/devices/iio:device0/sampling_rate
12:  /sys/bus/iio/devices/iio:device0/scan_elements/in_voltage0_en
13:  /sys/bus/iio/devices/iio:device0/scan_elements/in_voltage0_index
14:  /sys/bus/iio/devices/iio:device0/scan_elements/in_voltage0_type
15:  /sys/bus/iio/devices/iio:device0/scan_elements/in_voltage1_en
16:  /sys/bus/iio/devices/iio:device0/scan_elements/in_voltage1_index
17:  /sys/bus/iio/devices/iio:device0/scan_elements/in_voltage1_type
```

1. What is LibIIO?

Libiio will:

- identify the IIO devices that can be used
- identify the channels for the device
- identify the attributes specific to the channel
- identify the attributes specific to the device
- create a context where devices will be placed

1. What is LibIIO?

- LibIIO can run on:
  - An embedded system running Linux that includes IIO drivers for devices that are physically connected to the system, such as ADCs, DACs, etc. Also can run on an embedded system with a non-Linux framework. (Target)
  - A PC running a Linux distribution, Windows, Mac OS, OpenBSD/NetBSD that is connected to the embedded system through a network, USB or serial connection. (Remote)

Summary

1. What is LibIIO?
2. A look at the library structure
3. The C language API
4. LibIIO bindings
5. Practices that aim for a robust library

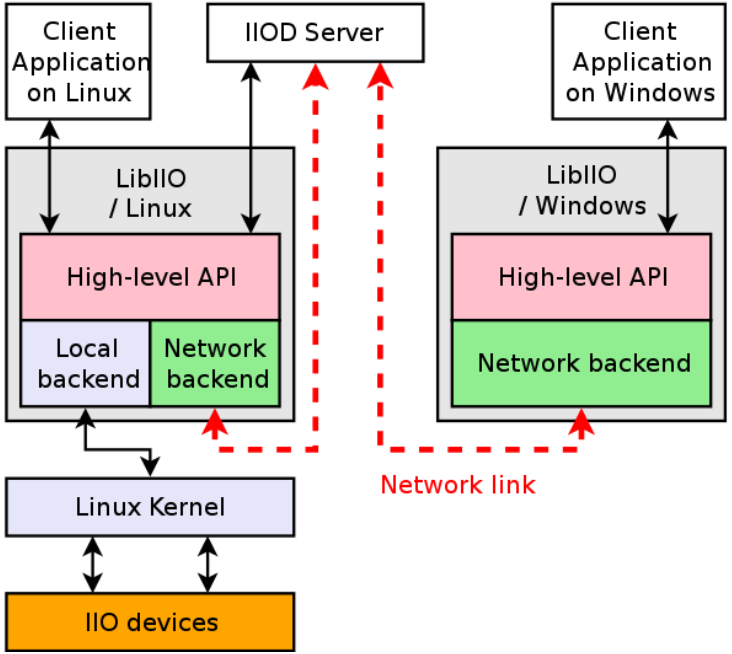
2. A look at the library structure

The library is composed by one high-level API and several backends:

1. Local – interfaces the Linux through sysfs virtual filesystem
2. Network – interfaces the iiod server through a network link
3. USB - interfaces the iiod server through a USB link
4. XML – interfaces a XML file
5. Serial – interfaces tiny-iiod throught a serial link

The iiod and tiny-iiod are part of the libIIO.

Software stack for a network connection



Summary

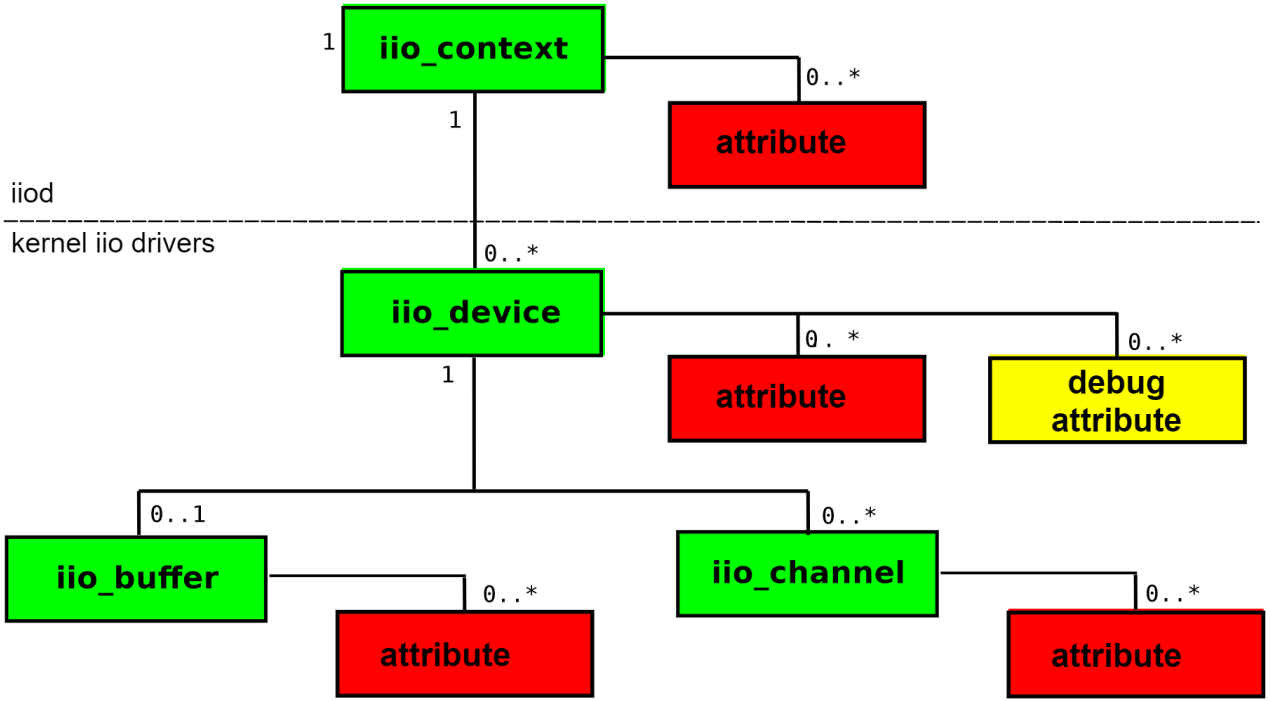
1. What is LibIIO?
2. A look at the library structure
3. The C language API 
4. Code examples
5. LibIIO bindings
6. Practices that aim for a robust library

3. The C language API

There are 4 data types that together make almost all of the API:

- iio_context (represents an instance of the library)
- iio_device
- iio_channel
- iio_buffer

The hierarchy of the 4 types:



3. The C language API

Scanning for IIO contexts:

iio_create_scan_context()
iio_scan_context_get_info_list()
iio_context_info_get_description()
iio_context_info_get_uri()
iio_create_scan_block()
iio_scan_block_scan()
iio_scan_block_get_info()

iio_scan_context_destroy()
iio_context_info_list_free()
iio_scan_block_destroy()

3. The C language API

Example of searching for contexts:

```
struct iio_scan_context *scan_ctx;
scan_ctx = iio_create_scan_context(NULL, 0);

struct iio_context_info **info;
iio_scan_context_get_info_list(scan_ctx, &info);

const char *description = iio_context_info_get_description(info[0]);
const char *uri = iio_context_info_get_uri(info[0]);
…
iio_context_info_list_free(info);
iio_scan_context_destroy(scan_ctx);
```

3. The C language API

Creating IIO contexts:

```
iio_create_default_context()      /* local or IIOD_REMOTE env_var */
iio_create_local_context()        /* local */
iio_create_xml_context()          /* from XML file */
iio_create_xml_context_mem()      /* from XML data stored in memory */
iio_create_network_context()      /* network: IPv4 or IPv6 */
iio_create_context_from_uri()     /* takes Uniform Resource Identifier */
iio_context_clone()               /* duplicate context */

iio_context_destroy()
```

3. The C language API

Example of creating different types of contexts:

```
struct iio_context * local_ctx;
local_ctx = iio_create_local_context();

struct iio_context * network_ctx;
network_ctx = iio_create_network_context("ip:192.168.100.15");

struct iio_context * usb_ctx;
usb_ctx = iio_create_context_from_uri("usb:3.80.5"); /*usb:[device:port:instance] */

struct iio_context * serial_ctx;
serial_ctx = iio_create_context_from_uri("serial:/dev/ttyUSB0,115200,8n1");
/* serial:[port],[baud],[config] */
```

ANALOG DEVICES
AHEAD OF WHAT'S POSSIBLE™

THE LINUX FOUNDATION

3. The C language API

Navigating through the context:

- **Device objects**
  - iio_context_get_devices_count()
  - iio_context_get_device()
  - iio_context_find_device()

- **Channel objects**
  - iio_device_get_channels_count()
  - iio_device_get_channel()
  - iio_device_find_channel()

3. The C language API

Example of going through all devices and through all channels of each device

```
struct iio_context * local_ctx;
local_ctx = iio_create_local_context();

int i;
 for (i = 0; i < iio_context_get_devices_count(local_ctx); ++i) {
        struct iio_device *dev = iio_context_get_device(local_ctx, i);
        int j;
        for (j = 0; j < iio_device_get_channels_count(dev); ++j) {
                struct iio_channel *channel = iio_device_get_channel(dev, j);
        }
}
```

ANALOG DEVICES
AHEAD OF WHAT'S POSSIBLE™

THE LINUX FOUNDATION

3. The C language API

The attributes (or parameters) can be identified by name, they can represent a value or an action and belong to one of the following types:

- **iio_context**
  - **Get attributes count:** iio_context_get_attrs_count()
  - **Get attribute at index:** iio_context_get_attr()
- **iio_device**
  - **Get attributes count:** iio_device_get_attrs_count()
  - **Get attribute at index:** iio_device_get_attr()
- **iio_channel**
  - **Get attributes count:** iio_channel_get_attrs_count()
  - **Get attribute at index:** iio_channel_get_attr()
- **iio_buffer**
  - **Get attributes count:** iio_device_get_buffer_attrs_count()
  - **Get attribute at index:** iio_device_get_buffer_attr()

3. The C language API

Reading from or writing to attributes:

- Read device-specific attributes:
  - iio_device_attr_read()
  - iio_device_attr_read_all()
  - iio_device_attr_read_bool()
  - iio_device_attr_read_longlong()
  - iio_device_attr_read_double()
- Read channel-specific attributes:
  - iio_channel_attr_read()
  - iio_channel_attr_read_all()
  - iio_channel_attr_read_bool()
  - iio_channel_attr_read_longlong()
  - iio_channel_attr_read_double()
- Write device-specific attributes:
  - iio_device_attr_write()
  - iio_device_attr_write_all()
  - iio_device_attr_write_bool()
  - iio_device_attr_write_longlong()
  - iio_device_attr_write_double()
- Write channel-specific attributes:
  - iio_channel_attr_write()
  - iio_channel_attr_write_all()
  - iio_channel_attr_write_bool()
  - iio_channel_attr_write_longlong()
  - iio_channel_attr_write_double()

3. The C language API

Capturing samples from or sending samples to device:

These two actions are done through the iio_buffer object and its functions.

Steps:

- **Enable channels**

  iio_channel_enable(), iio_channel_disable(), iio_channel_is_enabled()

  The enable/disable will actually happen when the iio_buffer is created.

  Not all channels can be enabled, only those of type 'scan_element'. This can be checked with iio_channel_is_scan_element(). A 'scan_element' is a channel capable of streaming data into/from a buffer.

- **Create buffer**

  iio_device_create_buffer(), iio_buffer_destroy()

- **Refill a buffer (for an input device)**

  To update the buffer with new samples: iio_buffer_refill()

3. The C language API

Capturing samples from or sending samples to device:

- **Push to a buffer (for an output device)**
  - To send new samples to the buffer: iio_buffer_push()

  If the iio_buffer object has been created with the "cyclic" parameter set, and the kernel driver supports cyclic buffers, the submitted buffer will be repeated until the iio_buffer is destroyed, and no subsequent call to iio_buffer_push() will be allowed.

- **Push a subset of samples to a buffer (for an output device)**
  - To send fewer samples than the size of the buffer: iio_buffer_push_partial()

Accessing samples from the iio_buffer:

- **Iterating over the buffer with a callback**

  Libiio provides a way to iterate over the buffer by registering a callback function, with the iio_buffer_foreach_sample() function.

  The callback function will be called for each "sample slot" of the buffer, which will contain a valid sample if the buffer has been refilled, or correspond to an area where a sample should be stored if using an output device.

3. The C language API

Accessing samples from the iio_buffer:

Example:

```
ssize_t sample_cb(const struct iio_channel *chn, void *src, size_t bytes, void *d)
{
    /* Use "src" to read or write a sample for this channel */
}
int main(void)
{
    ...
    Iio_buffer_for_each_sample(buffer, sample_cb, NULL);
    ...
}
```

Note that the callback will be called in the order that the samples appear in the buffer, and only for samples that correspond to channels that were enabled.

3. The C language API

Accessing samples from the iio_buffer:

- **Iterating on the samples with a for loop**

This method allows you to iterate over the samples slots that correspond to one channel. As such, it is interesting if you want to process the data channel by channel.

It basically consists in a for loop that uses the functions iio_buffer_first(), iio_buffer_step() and iio_buffer_end():

```
for (void *ptr = iio_buffer_first(buffer, channel); ptr < iio_buffer_end(buffer); ptr += iio_buffer_step(buffer)) {
/* Use "ptr" to read or write a sample for this channel */
}
```

Summary

1. What is LibIIO?
2. A look at the library structure
3. The C language API
4. LibIIO bindings 
5. Practices that aim for a robust library

4. LibIIO bindings

- Can be used directly in C++
- Python
- C#

Managed separately outside of LibIIO repository:

- Rust:

https://github.com/fpagliughi/rust-industrial-io

- Node.js:

https://github.com/drom/node-iio

- GNU Radio:

https://github.com/analogdevicesinc/gr-iio

4. LibIIO bindings

Python bindings:

- The python bindings consist of a .py file (https://github.com/analogdevicesinc/libiio/blob/master/bindings/python/iio.py)
- The ‘ctypes’ modules has been used to write the bindings
- Since v0.21 the python bindings have been available through pypi, and therefore can be installed with pip:

  pip install pylibiio

- Doc: https://analogdevicesinc.github.io/libiio/master/python/index.html

4. LibIIO bindings

C# bindings:

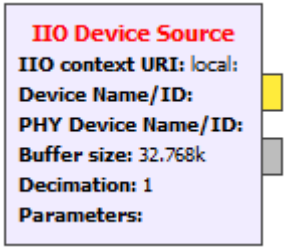
They cover the full panel of features that libiio provides.

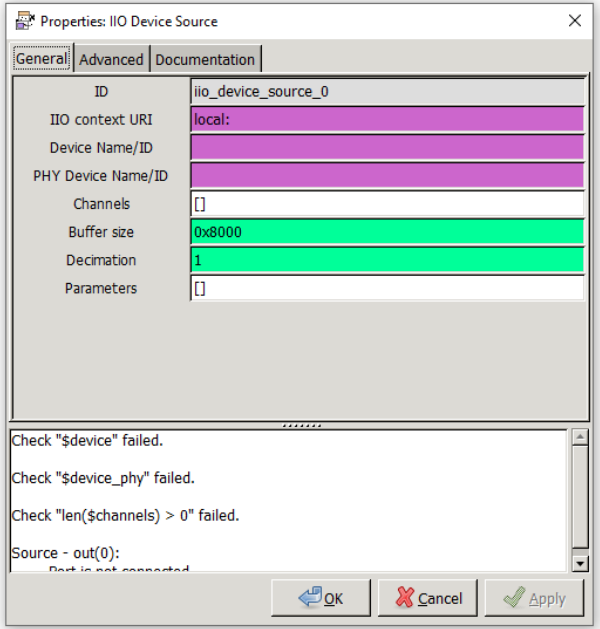
- The C# bindings are spread across multiple files:
  - ScanContext.cs, Context.cs, Devices.cs, Channel.cs, IOBuffer.cs, Attr.cs, Trigger.cs, IoLib.cs
  - Each of these files provide a couple of methods that directly call their C counterpart
- Doc: https://analogdevicesinc.github.io/libiio/master/csharp/index.html

4. LibIIO bindings

GNU Radio integration:

LibIIO can be integrated with GNU Radio through the IIO blocks: **iio_device_source** and **iio_device_sink**.
These blocks are available through the **gr-iio** module.



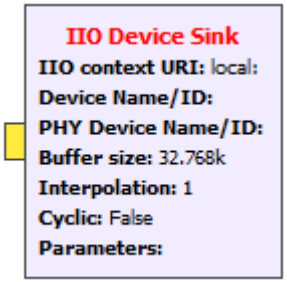


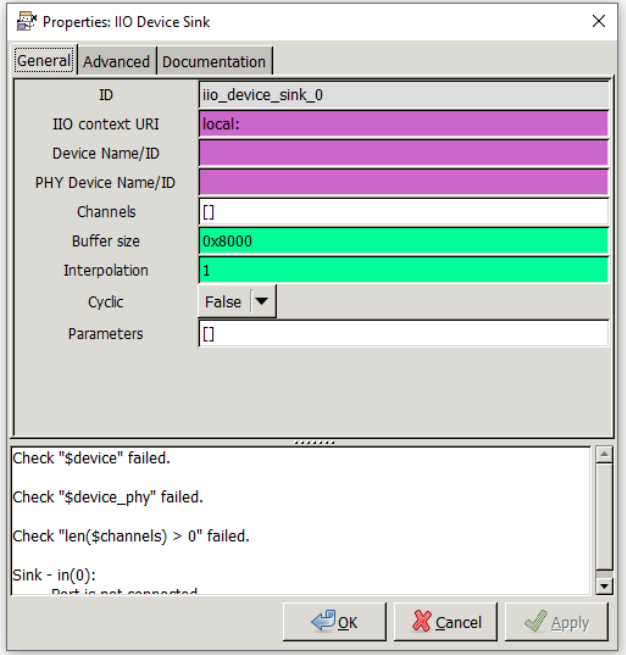
The iio_device_source provides configuration fields for:

- Choosing the context (IIO context URI)
- Enabling the channels (Channels – e.g [“voltage0”])
- Setting the buffer size (Buffer size)
- Decimation
- Changing values of device attributes (Parameters)

4. LibIIO bindings

GNU Radio integration:





The iio_device_sink provides configuration fields for:

- Choosing the context (IIO context URI)
- Enabling the channels (Channels – e.g [“voltage0”])
- Setting the buffer size (Buffer size)
- Interpolation
- Setting the Cyclic flag
- Changing values of device attributes (Parameters)

e.g. ([“in_voltage0_samplerate=24000”,])

Summary

1. What is LibIIO?
2. A look at the library structure
3. The C language API
4. LibIIO bindings
5. Practices that aim for a robust library 

5. Practices that aim for a robust library

Practice 1:

Development using a Pull-Request system where each PR is subject to review. No PR can be merged without at least one approval. Pushing commits directly to master is disabled.

Practice 2:

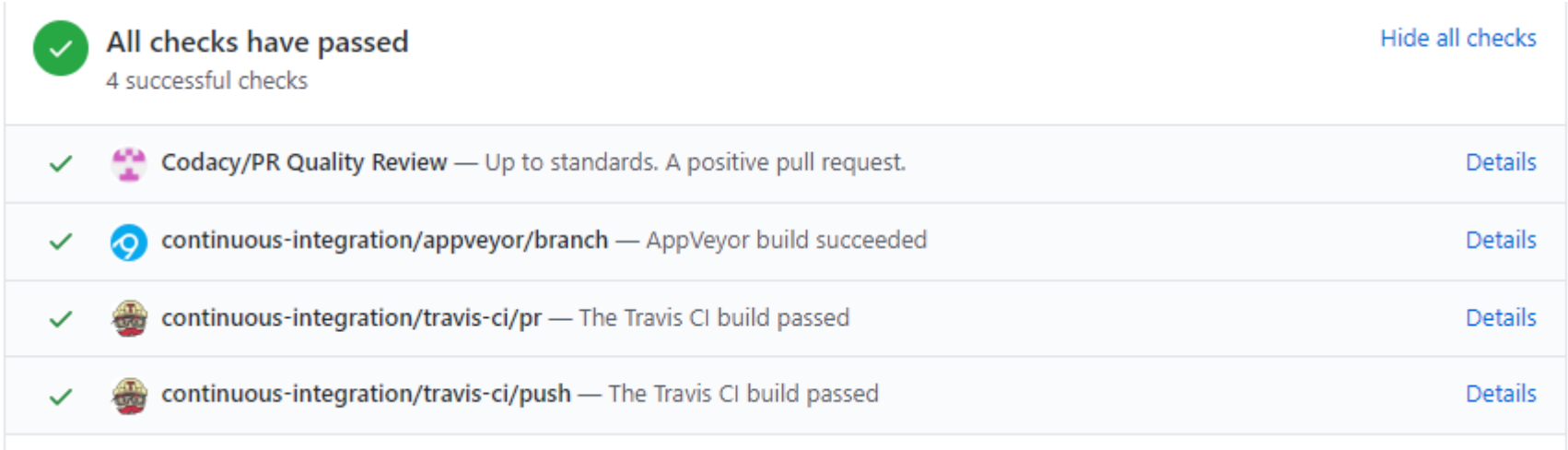
Enable as many warnings as possible*: -Wall*, *-Wextra*, *-Wpendatic*, *-std=C99*

The warnings are treated as errors. This enforces to not have warnings pile up. The –*Werror* flag will treat warnings as errors but only when LibIIO is built by CI.

5. Practices that aim for a robust library

Practice 3:

Use Continuous Integration (CI) when a PR is submitted (also for regular branches)



5. Practices that aim for a robust library

Appveyor checks Windows builds.

Travis checks MacOS builds, Ubuntu(Jessie, Stretch) builds, CentOS (6, 7, 8) builds.

Practice 4:

Make use of static analyzers to look for code issues:

- Coverity (as one of the Travis jobs)
- Codacy (integrated with Github)

Other aspects of the library

- The LibIIO ABI tries to be both backwards and forwards compatible
- Uses CMake to facilitate the building of Libiio
- One header (iio.h) and one shared library
- Doxygen generated API documentation
- Latest release is: v0.21 (26 releases so far)

LibIIO dependencies

- Core dependencies: *libxml2*, *bison*, *flex*
- Backends dependencies:
  - Local: *libaio*
  - USB: *libusb*
  - Network: *libavahi*
  - Serial: *libserialport*
- Documentation dependencies: *doxygen*, *graphviz*

Further reading on libIIO

- Hosted on:
  https://github.com/analogdevicesinc/libiio
- Welcome page:
  https://analogdevicesinc.github.io/libiio/master/index.html
- API documentation:
  https://analogdevicesinc.github.io/libiio/master/libiio/index.html
- Wiki libIIO overview:
  https://wiki.analog.com/resources/tools-software/linux-software/libiio
- Wiki libIIO internals:
  https://wiki.analog.com/resources/tools-software/linux-software/libiio_internals

Thank you!

Embedded Linux Conference Europe