Introduction and review of Linux's Industrial Input/Output (IIO) subsystem

Robin Getz
Open Source Developer
23 Jun 2025

Robin Getz

Robin Getz brings over 25 years of experience in embedded systems, hardware design, and software architecture. With a background that spans digital and analog design, kernel drivers, system libraries, and user interfaces, he has a strong track record of building end-to-end solutions across a range of industries.

His work focuses on team leadership, system-level thinking, and translating complex technical requirements into high-quality, user-focused products. Robin is especially passionate about data acquisition systems, sensor interfaces, modeling tools like MATLAB and SIMULINK, and embedded SoCs (Xilinx Zynq, Intel SoC, FreeRTOS, Linux, and others).

He enjoys diving into new projects—bringing both hands-on experience and strategic insight to help teams deliver innovative, reliable systems.

Fun Fact: Robin's first Linux install was Slackware… using 29 floppy disks. He's been hands-on with open systems ever since.

https://www.linkedin.com/in/rgetz/
https://x.com/robinlgetz
rgetz503@gmail.com

What does the Linux® kernel do?

- The Linux kernel is the main component of a Linux operating system (OS) and is the core interface between a computer's hardware and its processes. It communicates between the 2, managing resources as efficiently as possible.
- The kernel has 4 jobs:
  - Memory management: Keep track of how much memory is used to store what, and where (including DMA), and what memory belongs to what process
  - Process management: Determine which processes/users can use the central processing unit (CPU) and peripherals when, and for how long (scheduling)
  - Device drivers: Act as mediator/interpreter between the hardware and processes
  - System calls and security: Receive requests for service from the processes
- It's the kernel's job to manage hardware.
  (turn on, off, and allocate to users)

Device Drivers belong in the kernel (not in userspace)

- It's the kernel's job to mediator/interpreter between the hardware and processes, and ensure only the "right" user gets access to the hardware.

Kernel-Space Device Drivers:

1. **Execution Context**:
   - Runs in kernel mode, which has full access to the system's hardware and resources.
2. **Performance**:
   - Typically faster because they operate directly within the kernel, avoiding context switches between user mode and kernel mode.
3. **Access to Resources**:
   - Can access hardware directly and manage interrupts, memory, and other low-level operations.
4. **Complexity**:
   - More complex to develop and debug due to the need for careful management of kernel resources and potential for system crashes. Kernel Developers do this all day, and don't see this as "complexity".

User-Space Device Drivers

1. **Execution Context**:
   - Runs in user mode, which has restricted access to hardware and system resources for safety and stability.
2. **Performance**:
   - Will incur a performance overhead due to context switches between user space and kernel space, especially during I/O operations.
3. **Access to Resources**:
   - Cannot directly access hardware; instead, it communicates with the kernel via system calls, which can limit functionality and performance.
4. **Complexity**:
   - Generally easier to develop and debug since user-space applications can be run and tested without risking system stability. Non-kernel Developers should stay in userspace.

Device Drivers belong in the kernel (not in userspace)

- It's the kernel's job to mediator/interpreter between the hardware and processes, and ensure only the "right" user gets access to the hardware.

Kernel-space drivers are crucial for performance-critical and low-level hardware interactions, while user-space drivers offer a safer and more flexible environment for development, with some performance loss.

The choice between the two often depends on the specific requirements of the device and the desired trade-offs between developer skillset, performance, and complexity.

memory, and other low-level operations.

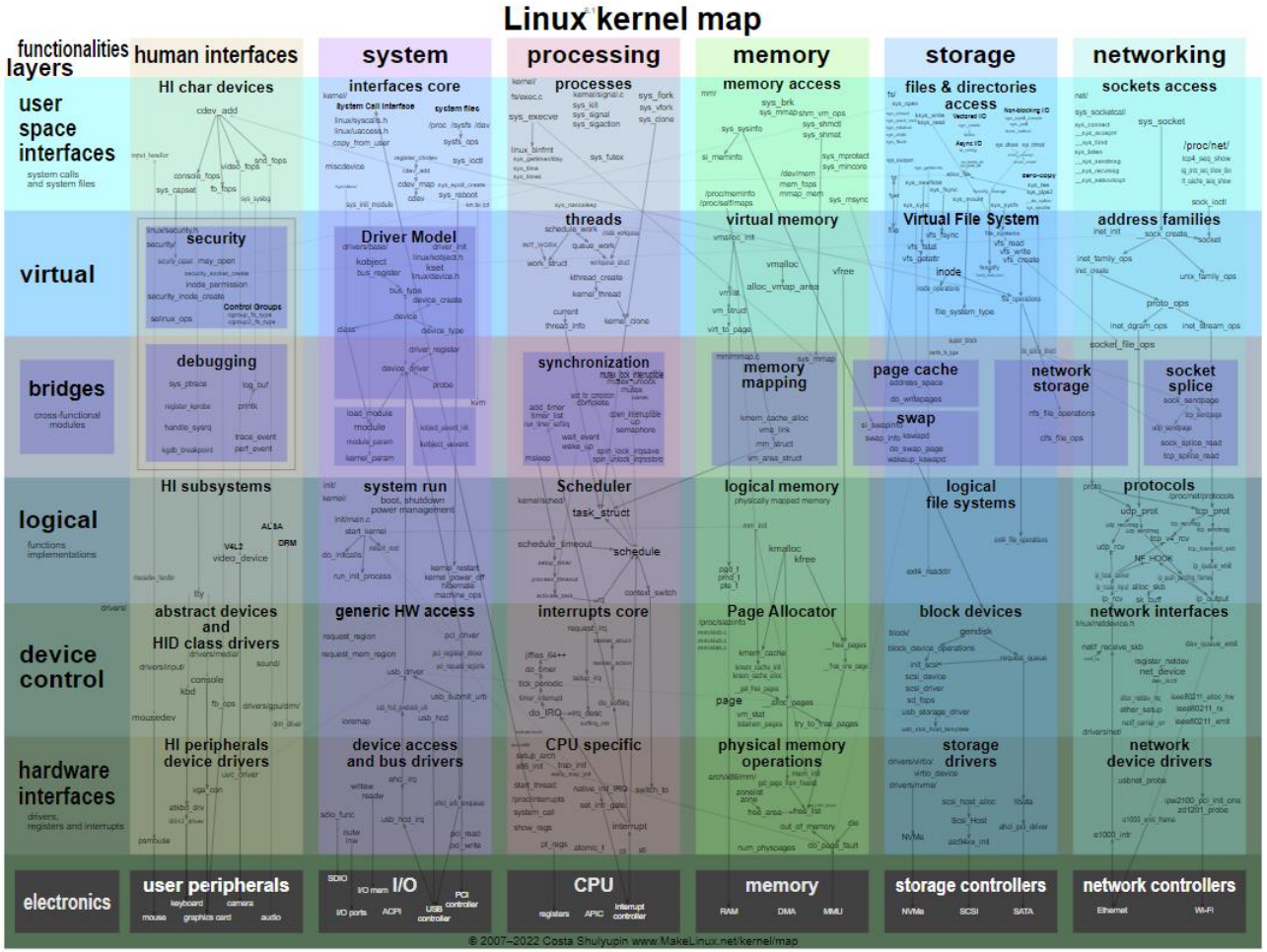
4. **Complexity**:
   - More complex to develop and debug due to the need for careful management of kernel resources and potential for system crashes. Kernel Developers do this all day, and don't see this as "complexity".

communicates with the kernel via system calls, which can limit functionality and performance.

4. **Complexity**:
   - Generally easier to develop and debug since user-space applications can be run and tested without risking system stability. Non-kernel Developers should stay in userspace.

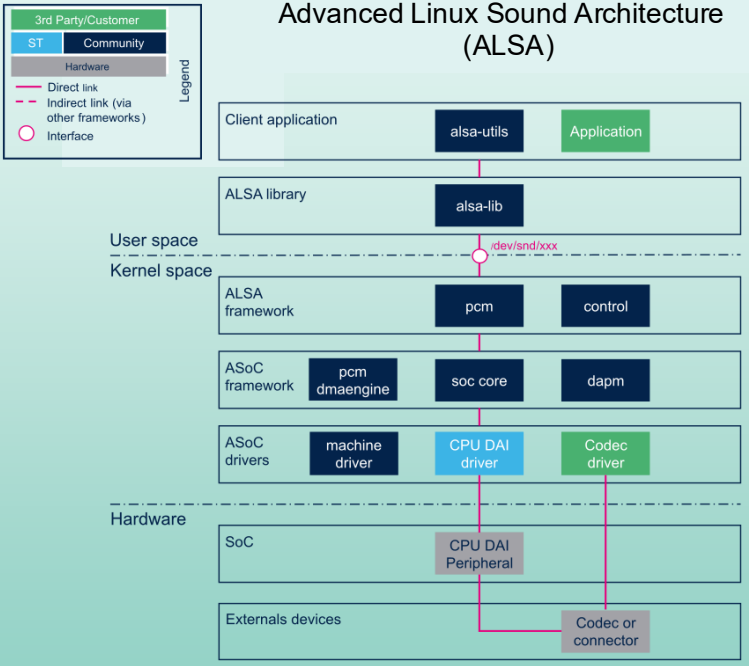
Linux Kernel

- Split into subsystems
- Drivers is a major subsystem
- Drivers are grouped into similar types of devices
  - Audio (ALSA)
  - Video
  - Human Interface (mouse, keyboard, etc)
  - IIO (ADC/DACs)
  - Power / frequency
  - File Systems
- Buses are the way things communicate
  - SPI, I²C
  - USB
  - PCIe
  - Bluetooth

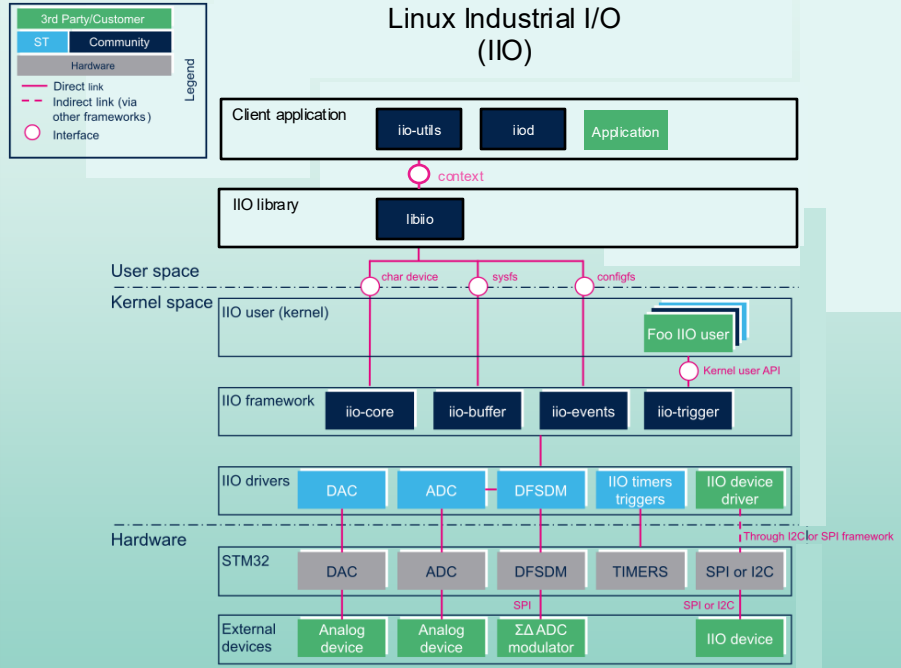


https://makelinux.github.io/kernel/map/intro

Different subsystems have different architectures, use different libraries, but have “common” kernel / user space separation



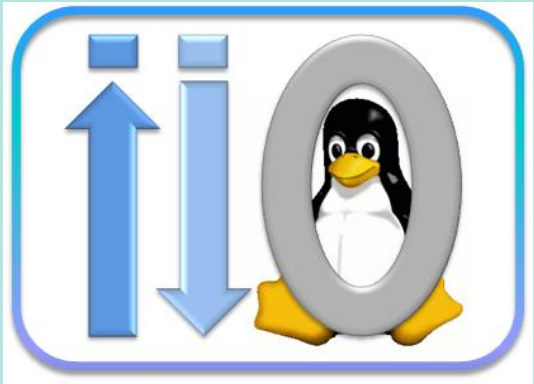
https://wiki.st.com/stm32mpu/wiki/ALSA_overview



https://wiki.st.com/stm32mpu/wiki/IIO_overview

Linux IIO

- The Industrial I/O subsystem is intended to provide support for devices that in some sense are analog to digital or digital to analog converters (ADCs, DACs).
  - Devices that fall into this category are:
  - ADCs
  - Accelerometers
  - Gyros
  - IMUs
  - Capacitance to Digital Converters (CDCs)
  - Pressure Sensors
  - Color, Light and Proximity Sensors
  - Temperature Sensors
  - Magnetometers
  - DACs
  - DDS (Direct Digital Synthesis)
  - PLLs (Phase Locked Loops)
  - Variable/Programmable Gain Amplifiers (VGA, PGA)

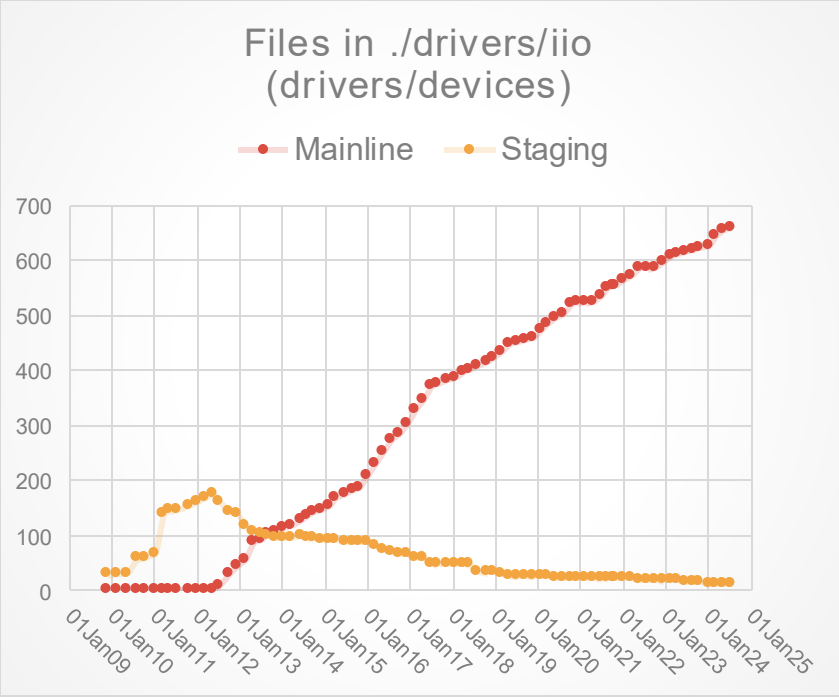


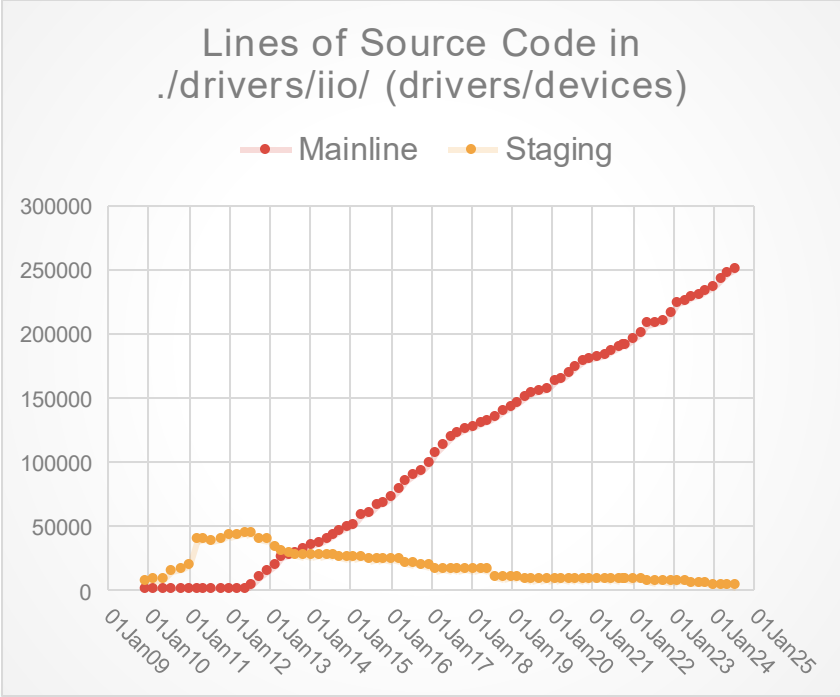
- Added to the Linux Kernel (staging) 15 Sept 2009 (2.6.32) by Jonathan Cameron, research associate, University of Cambridge (now Huawei Research in UK)
- Moved out of staging 25 April 2012 (3.4) by Jonathan Cameron - 2 years, 7 months, 10 days to “bake”.
- High Speed “DMABUF” interface for Rx/Tx added by Paul Cercueil accepted upstream 30 June 2024, in kernel 6.11.

Increasing popularity in the Linux kernel

Since 15 Sept 2009 (2.6.32) through 15 July 2024 (v6.10)

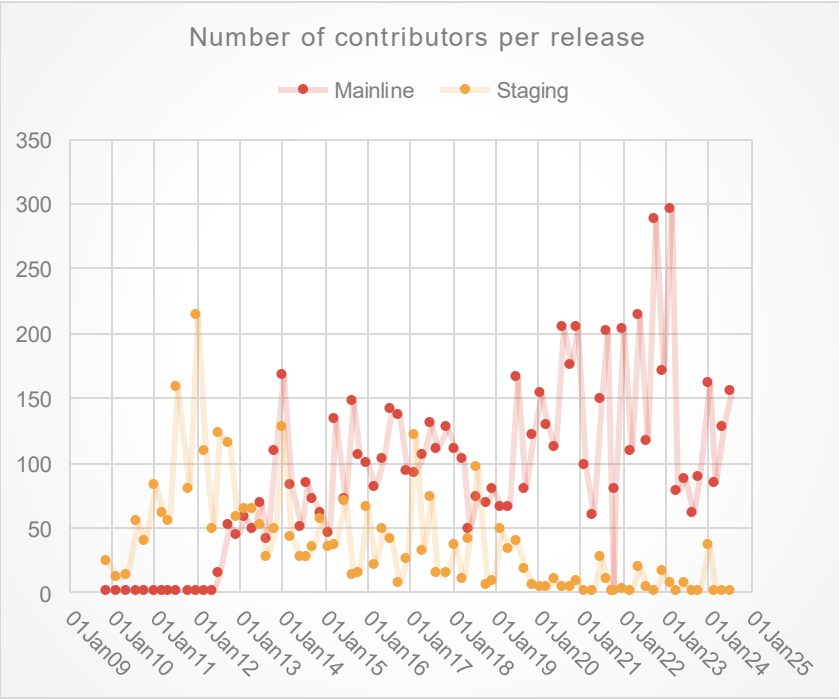
Currently 254,991 lines of code

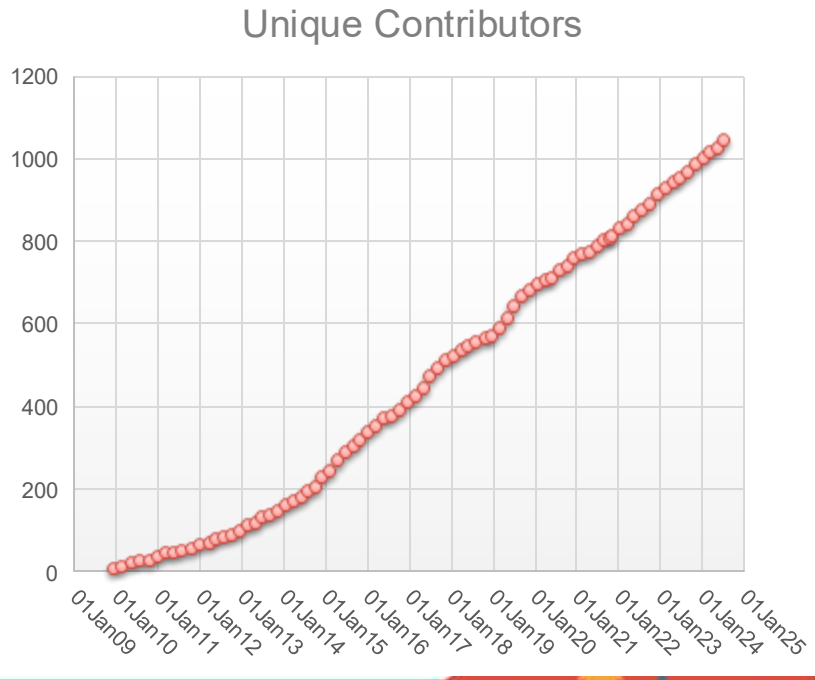




Increasing popularity in the Linux kernel

Since 15Sept2009 (2.6.32) through 15July24 (v6.10)





Who?

- Combination of:
  - Semiconductor (sensor) vendors,
  - Third parties (Tools and Consultants) and
  - End users

analog.com
arm.com
arm.linux.org.uk
atmel.com
beagleboard.org
bp.renesas.com
broadcom.com
freescale.com
hisilicon.com
idt.com
intel.com
invensense.com
linaro.org
linux-foundation.org
maximintegrated.com
mediatek.com
microchip.com
nvidia.com
nxp.com
opensource.cirrus.com
opensource.wolfsonmicro.com
rock-chips.com
st.com
ti.com
xilinx.com

denx.de
iis.fraunhofer.de
mathworks.com
redhat.com
suse.com
windriver.com

chromium.org (Google)
cisco.com
google.com
huawei.com
ni.com
osg.samsung.com
rockwellcollins.com
rohde-schwarz.com
samsung.com
seeed.cc
siemens.com
sonymobile.com
zte.com.cn

What types of devices are supported for IIO devices

- The Industrial I/O subsystem is intended to provide support for devices that in some sense are analog to digital or digital to analog converters (ADCs, DACs).
- Staging and development:
  - https://git.kernel.org/pub/scm/linux/kernel/git/jic23/iio.git/
- Other vendor (sensor) source trees, where development happens:
  - STM: https://github.com/STMicroelectronics/linux/tree/v6.1-stm32mp/drivers/iio
  - AMD: https://github.com/Xilinx/linux-xlnx/tree/master/drivers/iio
  - ADI: https://github.com/analogdevicesinc/linux/tree/main/drivers/iio
  - Microchip : https://github.com/linux4microchip/linux/tree/linux4microchip-2024.04/drivers/iio
  - NXP : https://github.com/nxp-imx/linux-imx/tree/lf-6.6.y/drivers/iio

```
Mainline drivers, peer reviewed, accepted:
source "drivers/iio/accel/Kconfig"
source "drivers/iio/adc/Kconfig"
source "drivers/iio/addac/Kconfig"
source "drivers/iio/afe/Kconfig"
source "drivers/iio/amplifiers/Kconfig"
source "drivers/iio/cdc/Kconfig"
source "drivers/iio/chemical/Kconfig"
source "drivers/iio/common/Kconfig"
source "drivers/iio/dac/Kconfig"
source "drivers/iio/dummy/Kconfig"
source "drivers/iio/filter/Kconfig"
source "drivers/iio/frequency/Kconfig"
source "drivers/iio/gyro/Kconfig"
source "drivers/iio/health/Kconfig"
source "drivers/iio/humidity/Kconfig"
source "drivers/iio/imu/Kconfig"
source "drivers/iio/light/Kconfig"
source "drivers/iio/magnetometer/Kconfig"
source "drivers/iio/multiplexer/Kconfig"
source "drivers/iio/orientation/Kconfig"
source "drivers/iio/test/Kconfig"
source "drivers/iio/trigger/Kconfig“
source "drivers/iio/position/Kconfig"
source "drivers/iio/potentiometer/Kconfig"
source "drivers/iio/potentiostat/Kconfig"
source "drivers/iio/pressure/Kconfig"
source "drivers/iio/proximity/Kconfig"
source "drivers/iio/resolver/Kconfig“
source "drivers/iio/temperature/Kconfig"
```

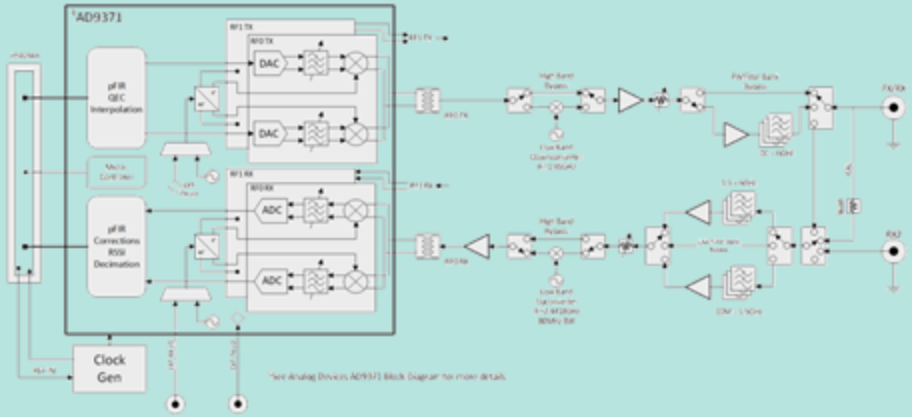
High Speed Interface (upstreamed)

https://www.kernel.org/doc/html/next/iio/iio_dmabuf_api.html

- The Industrial I/O subsystem supports access to buffers through a file-based interface, with read() and write() access calls through the IIO device's dev node.
- It additionally supports a DMABUF based interface, where the userspace can attach DMABUF objects (externally created) to an IIO buffer, and subsequently use them for data transfers.
  - A userspace application can then use this interface to share DMABUF objects between several interfaces, allowing it to transfer data in a zero-copy fashion, for instance between IIO and the USB stack.
  - The userspace application can also memory-map the DMABUF objects, and access the sample data directly. The advantage of doing this vs. the read() interface is that it avoids an extra copy of the data between the kernel and userspace. This is particularly useful for high-speed devices which produce several megabytes or even gigabytes of data per second.

Application level control, not just ADC/DACs

- Not just about “ADC/DAC”
- System level control
  - When tuning changes, external filters/switches/DSA/attenutator/gain/etc change
  - Clocking changes
  - External LO changes
  - DUC/DDC changes (data rate changes)
- Easier/Faster/hard real time when everything is in same subsystem
- Different vendors (RF / µWave / Baseband) working on same subsystem
- Extensible between evaluation, prototype and production

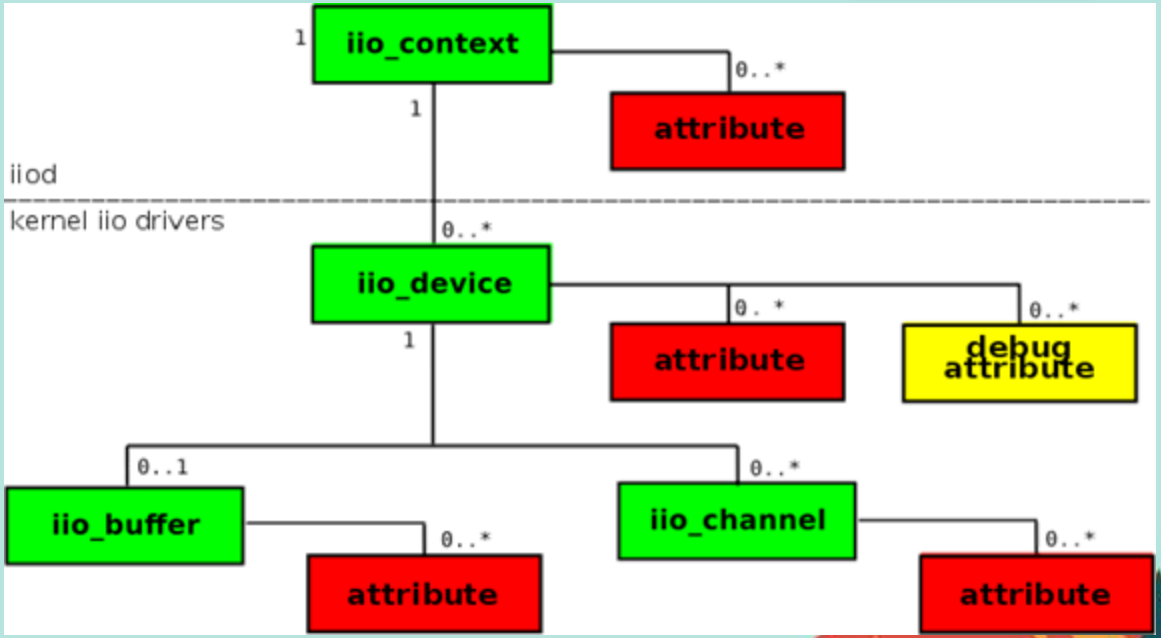


External devices
SPI, or GPIO controlled
All have separate IIO device drivers

“chains” of devices
https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?h=v6.12-rc3&id=1a97905d3e48ebe79a06d16143fbfa427c56ce5f

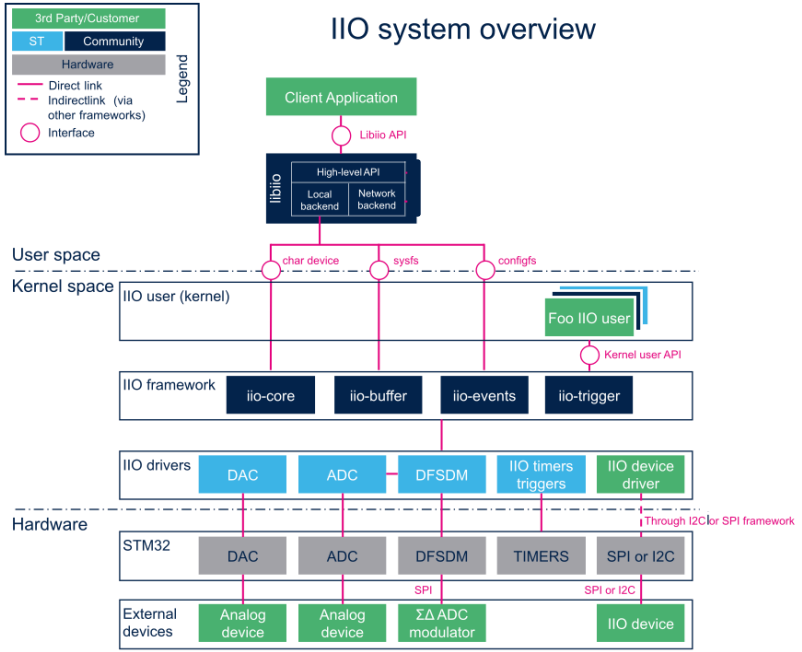
IIO Concepts

- Context:
  - Groups of devices
  - Management Info (attributes)
- Device:
  - Normally the “chip”
  - Not always the chip
    - Single chips can be split into multiple devices
  - Control (attributes)
  - Data (buffers and channels)

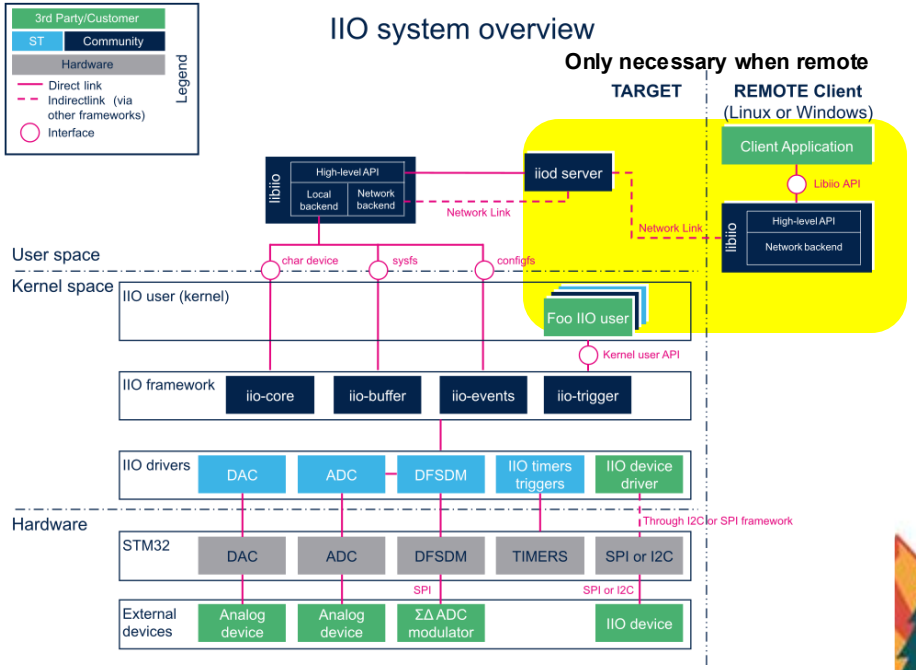


Two Main use cases: (1) deployed and (2) development

- "Local" (embedded/deployed)
  - Final Product



- "Remote" (from host)
  - during development stages (stream to host)

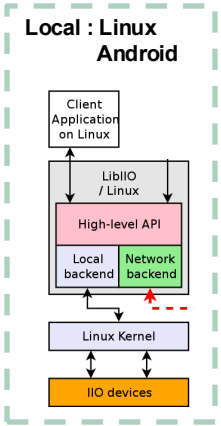


THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
NORTH AMERICA

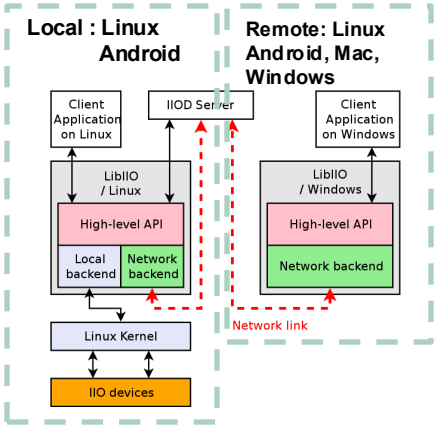
Moving from Remote to Local

opening a context

- Local:
  - `C:  iio_create_context_from_uri("local:");`
  - `Py:  iio.Context(_context="local:")`
  - `Rust:  iio::Context::from_uri("local:")`
  - `Julia: Context("local:")`



- Remote:
  - network host "brain.local"
  - `C:  iio_create_context_from_uri("ip:brain.local");`
  - `Py:  iio.Context(_context="ip:brain.local")`
  - `C#:  new Context("ip:brain.local");`
  - `Rust:  iio::Context::from_uri("ip:brain.local")`
  - `Julia: Context("ip:brain.local")`



- network ipv4 host
  `iio_create_context_from_uri("ip:192.168.1.119")`
- network ipv6 host
  `iio_create_context_from_uri("ip:2601:190:400:da:47b3:55ab:3914:bff1")`
- network ipv6 link-local host
  `iio_create_context_from_uri("ip:fe80::f14d:3728:501e:1f94eth0")`
- Network with mdns
  `iio_create_context_from_uri("raspberrypi.local")`
- usb
  `iio_create_context_from_uri("usb:3.32.5")`
- serial
  `iio_create_context_from_uri("serial:/dev/ttyUSB0,115200,8n1")`

Moving from Remote to Local

change an attribute – true RPC

A Remote Procedure Call (RPC) is a protocol or technology that allows a program to execute as if it were a local, without the programmer explicitly coding the details for remote interaction.

This abstraction simplifies the process of building distributed systems by hiding the complexities of network communication.

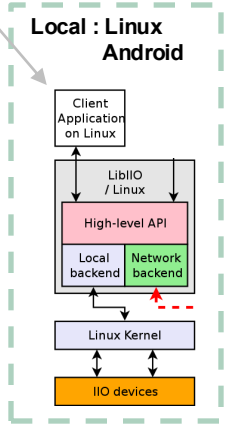
- Local:

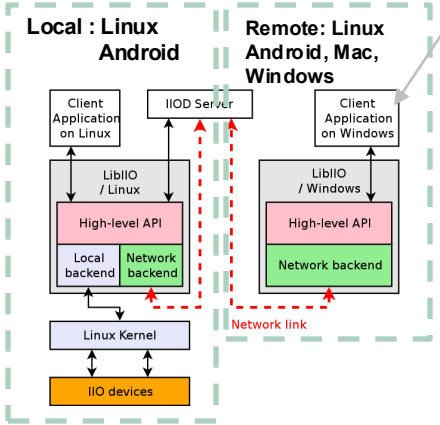
```
ctx = iio_create_context_from_uri("local:");
device = iio_context_find_device(ctx, "ad9361-phy");
channel = iio_device_find_channel(device, "altvoltage0", true);
iio_channel_attr_write_longlong(channel, "frequency", 2.4e6);
```

Same

- Remote:

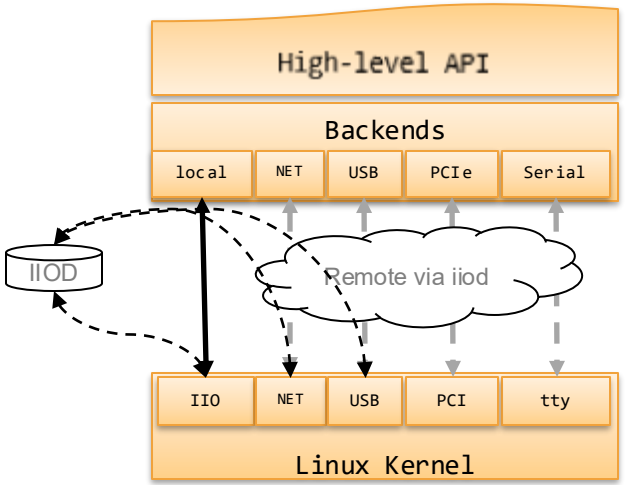
```
ctx = iio_create_context_from_uri("ip:brain.local");
device = iio_context_find_device(ctx, "ad9361-phy");
channel = iio_device_find_channel(device, "altvoltage0", true);
iio_channel_attr_write_longlong(channel, "frequency", 2.4e6);
```

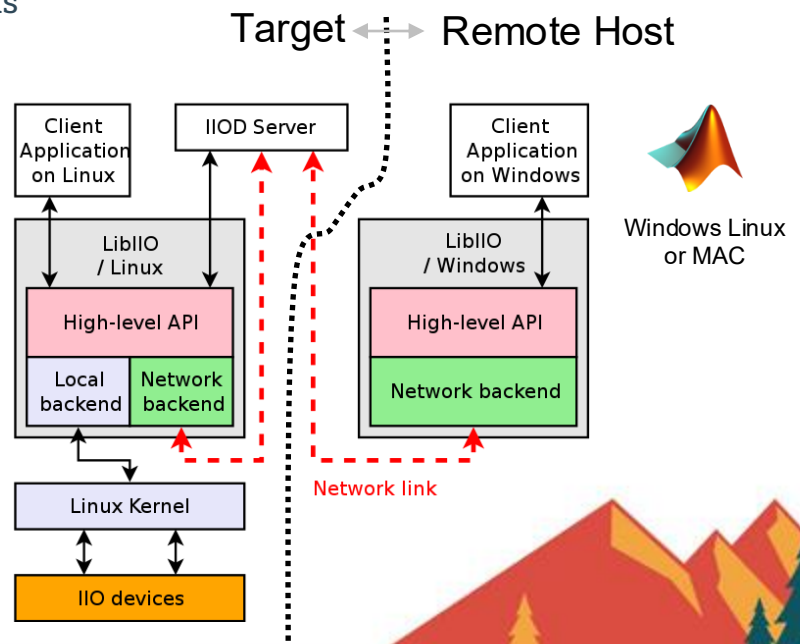




Libiio : Transport Layers (Backends)

- Support for backends via RPC mechanism
  - Backend takes care of low-level communication details
  - Provide the same API for applications
  - Transparent from the applications point of view

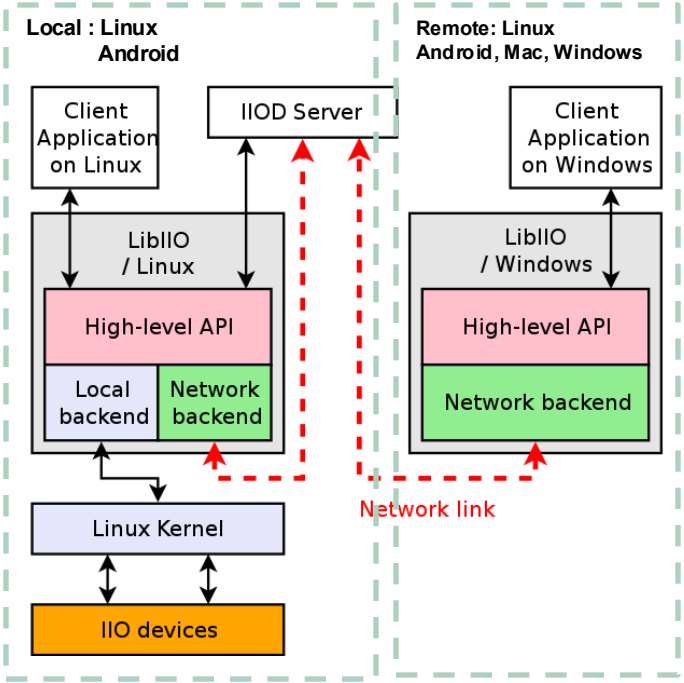




From : https://wiki.analog.com/resources/tools-software/linux-software/libiio

iiod

- The IIO Daemon (IIOD) server creates a libiio context that uses the “local” backend and then share it on the network (wired or wireless), USB or serial (serial, Bluetooth, JTAG, etc) to any client application using the corresponding backend of libiio.
- Host: Linux (local devices)
- Client: Linux, Windows, MAC, iOS, Android Over:
  - USB (Linux, Windows, Mac),
  - Ethernet (all),
  - Serial (all)



libiio improvements (0.26 -> 1.0)

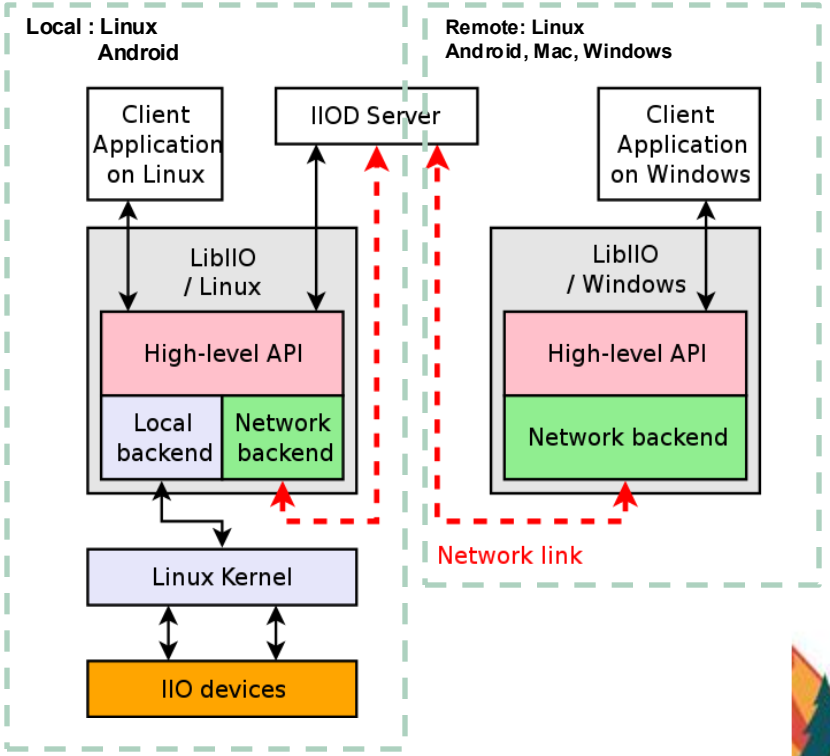
- **Robust Error Handling**
  - Pointer-returning functions now encode errors using special pointers rather than NULL and errno.
  - Use `iio_err`(ptr) to check for errors and `iio_strerror`() (or debug macros) to obtain human-readable message
- Better C++ support
- next-gen buffer model allowing direct enqueue/dequeue control and support for buffer metadata.
- Retains 0.x compatibility layer

- **Cleaner and safer** use of contexts and errors
- **Greater flexibility** for logging and streaming behaviors
- **Improved C++ experience** for modern application development
- **Foundation** for advanced buffer architectures and zero-copy support
- **Easier migration**, fewer breaking changes for existing users

Platforms / Language bindings

- Cross Platform (local on Linux/Android only)
  - Linux (amd64 arm64 armel armhf i386 mips64el mipsel ppc64el s390x)
    - Ubuntu packages
    - RedHat packages
    - Arch Linux
    - FreeBSD
    - OpenSUSE
    - BuildRoot
    - Debian Packages
    - Fedora
    - Raspbian
    - Gentoo
    - MacPorts
    - OpenEmbedded
  - Android
  - Mac (intel and apple)
  - Windows (amd64)

- Native C, C++, python (including pip & PyPi), C# interfaces
- External bindings for RUST (via crates), Julia, MATLAB, and node.js (via npm)
  - Meet algorithm/application customers where they already are.



Pythonic examples

Objects which can be created based on base classes

Configuring hardware properties and reading back settings

```
# Import the library
import adi

# Create a device interface
sdr = adi.ad9361()
# Configure properties
sdr.rx_rf_bandwidth = 4000000
sdr.rx_lo = 2000000000
sdr.tx_lo = 2000000000
sdr.tx_cyclic_buffer = True
sdr.tx_hardwaregain = -30
sdr.gain_control_mode = "slow_attack"

# Read back properties from hardware
print(sdr.rx_hardwaregain)
```

Send data to a device and receiving data from a device

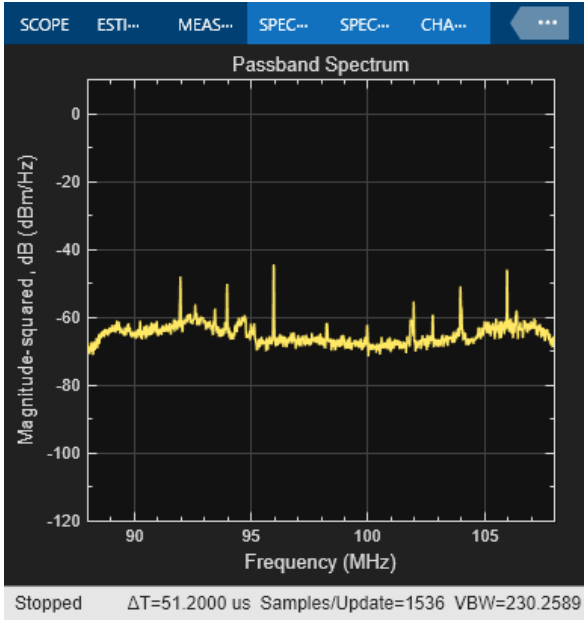
```
import adi
import numpy as np

sdr = adi.ad9361()
data = np.arange(1, 10, 3)

# Send
sdr.tx(data)

# Receive
data_rx = sdr.rx()
```

Pythonic refers to code that follows the conventions and idioms of the Python language, emphasizing readability, simplicity, and efficiency.

```
import sys

import adi
import matplotlib.pyplot as plt
import numpy as np
from scipy import signal

# 20-Bit, 1.8 MSPS/1 MSPS/500 kSPS, Easy Drive, Differential SAR ADCs
device_name = "ad4020"
vref = 5.0  # Manually entered, consult eval board manual

# Optionally pass URI as command line argument,
# else use default context manager search
my_uri = sys.argv[1] if len(sys.argv) >= 2 else "ip:analog.local"
print("uri: " + str(my_uri))

my_adc = adi.ad4020(uri=my_uri)
my_adc.rx_buffer_size = 4096

print("Sample Rate: ", my_adc.sampling_frequency)

data = my_adc.rx()

x = np.arange(0, len(data))
voltage = data * 2.0 * vref / (2 ** 20)
dc = np.average(voltage)  # Extract DC component
ac = voltage - dc  # Extract AC component

plt.figure(1)
plt.clf()
plt.title("AD4020 Time Domain Data")
plt.plot(x, voltage)
plt.xlabel("Data Point")
plt.ylabel("Voltage (V)")
plt.show()

f, Pxx_spec = signal.periodogram(
    ac, my_adc.sampling_frequency, window="flattop", scaling="spectrum"
)
Pxx_abs = np.sqrt(Pxx_spec)

plt.figure(2)
plt.clf()
plt.title("AD4020 Spectrum (Volts absolute)")
plt.semilogy(f, Pxx_abs)
plt.ylim([1e-6, 4])
plt.xlabel("frequency [Hz]")
plt.ylabel("Voltage (V)")
plt.draw()
plt.pause(0.05)

del my_adc
```

MATLAB-esque Examples

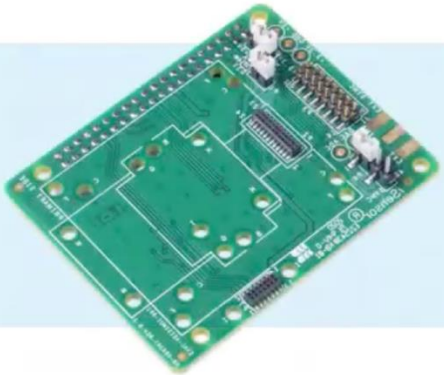
```
PlutoSDR = sdrrx('Pluto');

% Main loop
while radioTime < userInput.Duration
% Receive baseband samples (Signal Source)

  rcv = PlutoSDR();
  step(hSpectrum, rcv);

  % Update radio time.
  radioTime = radioTime + SAParams.FrontEndFrameTime
end

% Release all System objects
release(sigSrc);
release(hSpectrum);
```



MATLAB-esque describe code that adheres to MATLAB's idioms and best practices. MATLAB-esque code typically leverages MATLAB's strengths, such as matrix operations, built-in functions, and vectorization, to write efficient and readable code.

Example

ANALOG DEVICES
ADIS16480
iSensor®
MEMS IMU



ADI-KUIPER-LINUX
LINUX DEVICE DRIVERS FOR ADI PERIPHERALS

Demo

Streaming Data into MATLAB using the Command Window

- Make sure that the host PC is connected to the same network as the RPi.

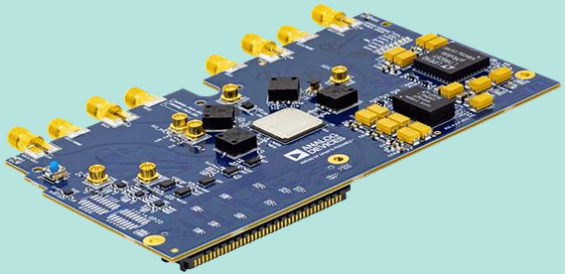
The Streaming Process

https://www.youtube.com/watch?v=G-k81W2yJRY

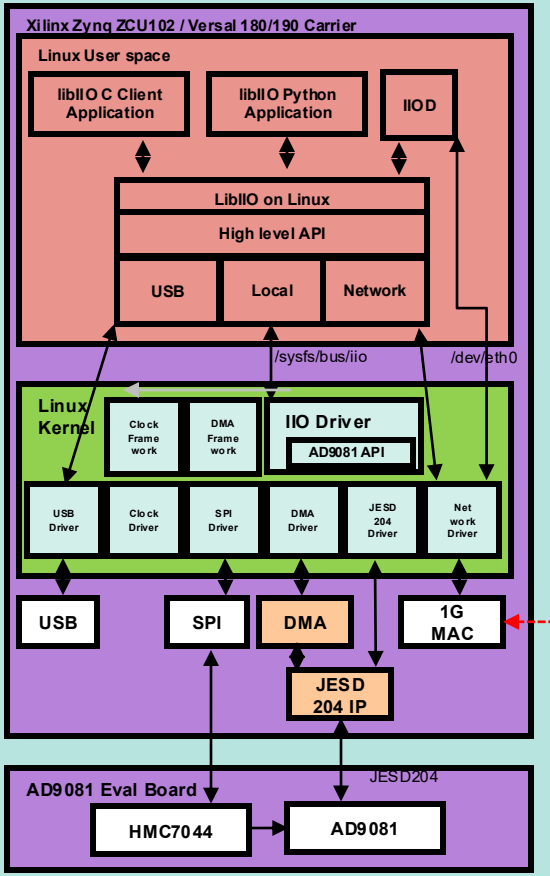
More details

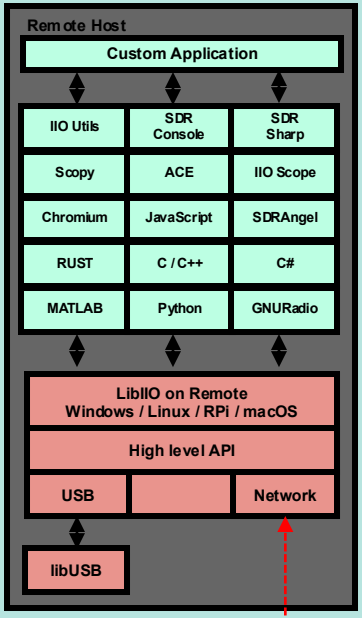
Complete Reference designs and example kernels

Many, many applications already natively support libiio on the host, which makes it easy to connect to remote hardware.



Quad, 16-Bit, 12GSPS RFDAC and Quad, 12-Bit, 4GSPS RFADC in single package





Standard/Discoverable IIO (command, control, and data) over TCP/IP

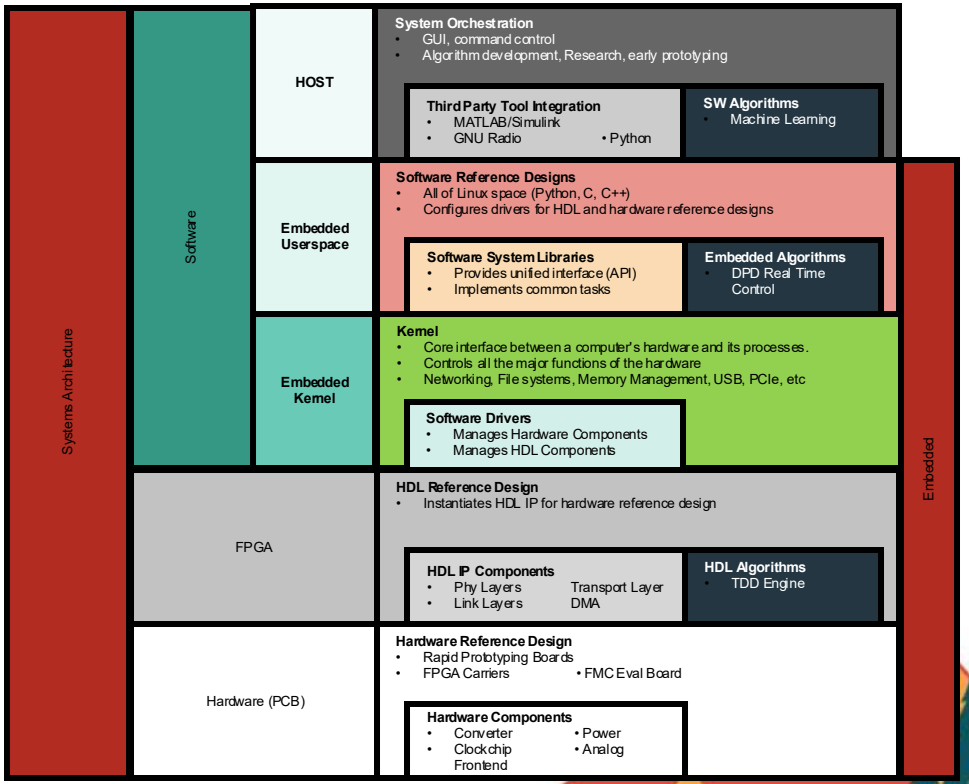
Remote functions over Serial, Bluetooth, USB (2 and 3) and PCIe

Analog Devices. ©2020 Analog Devices

Embedded OS Stack (to host)

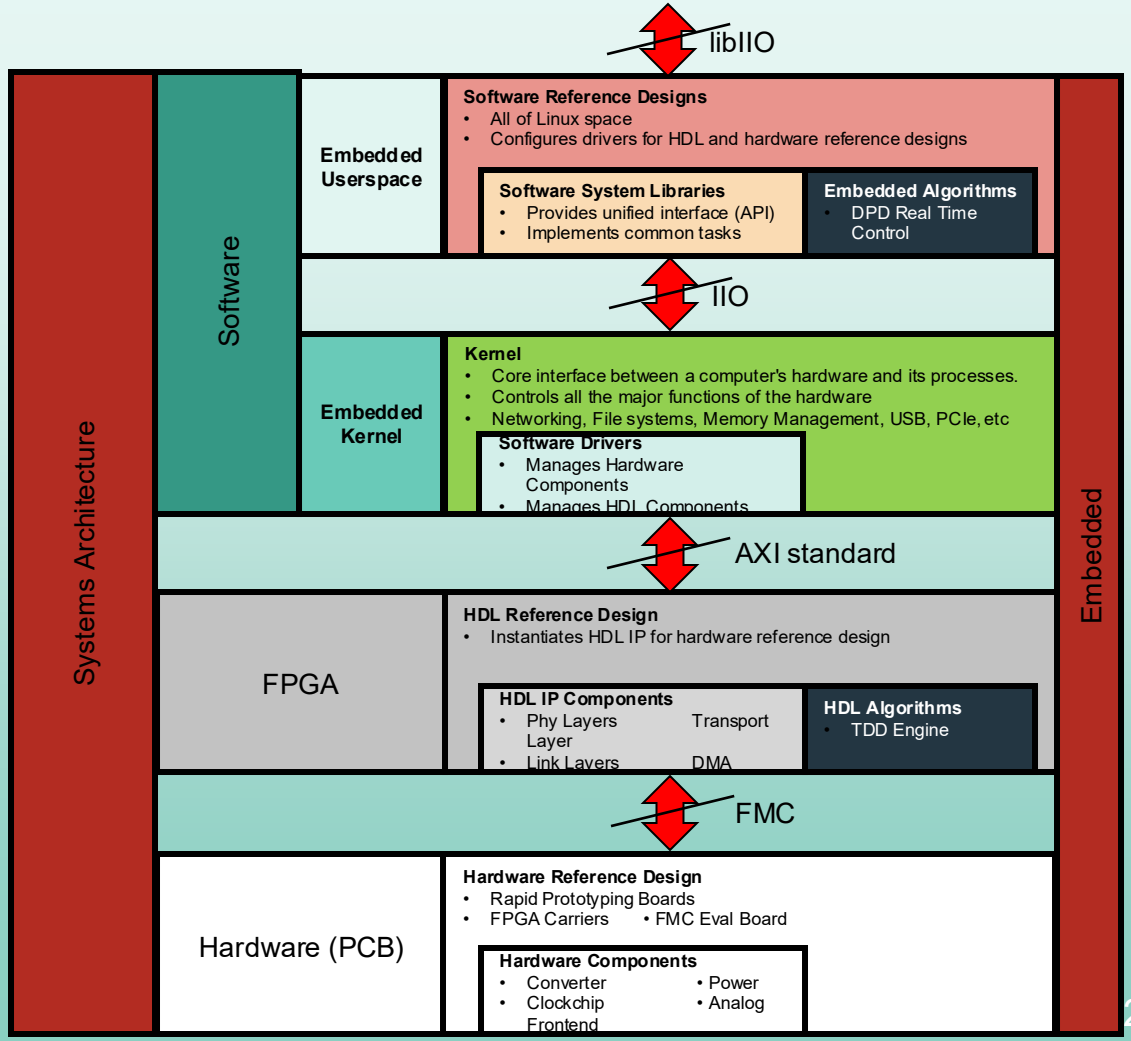
FPGA based system, with Linux

- Algorithm Engineers want to use the system, not understand how it works.
  - Capture Data
  - Send Data
- Want to command/control hardware from high level tools (ie. MATLAB) in a native manner
  - MATLAB System Objects
  - Python Objects
- Advanced hardware use cases need tight coupling between tools and hardware
  - ACG, TDD
- Test what you ship, ship what you test. Evaluation platform is shipping platform.



Interconnect

- Interconnect can be more important than the block itself
- How it “fits” into the larger system
  - Determines:
    - Yes I can use this.
    - No I can’t use this
- Bespoke vs industry standard
- Focus on the level you care about

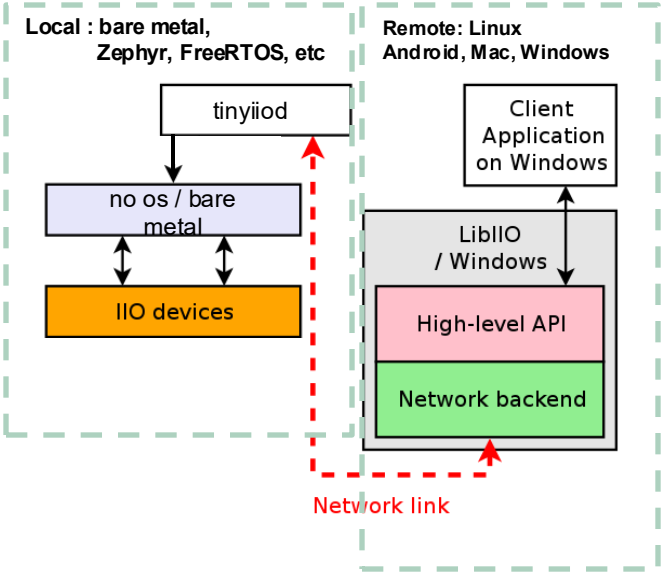


Isolate & Debug at different layers

- Cross platform command line utilities:
  - `iio_info` - list IIO devices and device attributes
  - `iio_attr` - list IIO devices, and read/write device attributes
  - `iio_readdev` - read buffers from an IIO device
  - `iio_writedev` - write buffers on an IIO device
  - `iio_reg` - do a low level read or write to SPI or I2C register

Not just for Linux anymore - Tinyiiod

- Tiny IIO Daemon runs on a no-operating-system (no-OS or MCU) host
- Offers a libiio compatible interface
- Written in C, dual licensed under GNU Lesser General Public License (LGPL), version 2.1 or MIT license.
- Supported connections:
  - Serial
  - Ethernet and WIFI



Libiio : License complications

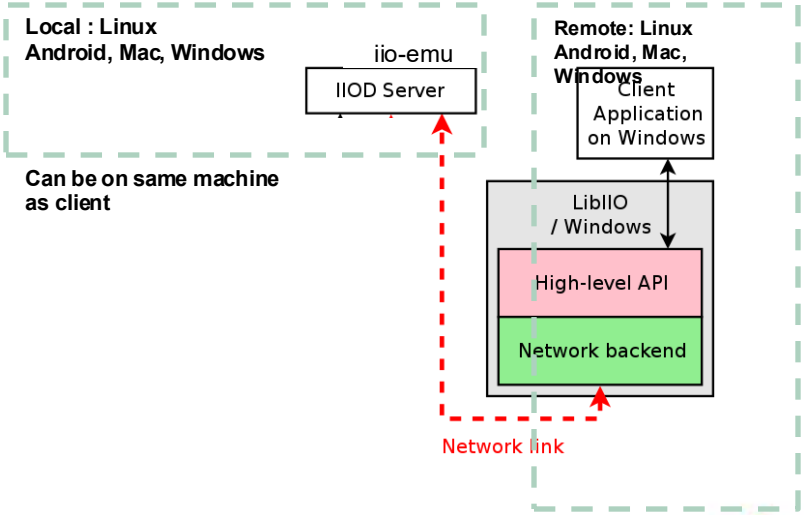
adds a spdx_check_target() CMake function to validate that all source files in a target (and its linked static/shared libraries) include a valid SPDX-License-Identifier tag and hides it behind a new "**-DLICENCE_CHECK**" cmake option.

```
=== License Check for : iio ==========
-- MIT: attr.c, backend.c, block.c, buffer.c, channel.c, context.c, device.c,
events.c, library.c, mask.c, scan.c, sort.c, stream.c, task.c, utilities.c, xml.c
-- LGPL-2.1-or-later: usb.c, local.c, local-dmabuf.c, local-mmap.c, network.c,
network-unix.c, dns_sd.c, dns_sd_avahi.c, lock.c, iiod-client.c, iiod-responder.c
-- Missing SPDX: deps/libini/libini.c
-- External libraries: /usr/lib/x86_64-linux-gnu/libusb-1.0.so, /usr/lib/x86_64-
linux-gnu/librt.so, /usr/lib/x86_64-linux-gnu/libatomic.so.1, /usr/lib/x86_64-
linux-gnu/libzstd.so, /usr/lib/x86_64-linux-gnu/libavahi-common.so,
/usr/lib/x86_64-linux-gnu/libavahi-client.so, /usr/lib/x86_64-linux-gnu/libxml2.so,
/usr/lib/x86_64-linux-gnu/libpthread.so
```

iio-emu – for when hardware doesn't exist

(or is in limited supply) great for CI systems. (work in progress)

- **`iio-emu`** emulates (or clones) a device
- Hardware or contexts can be emulated for testing without physical devices.
  - this emulation does not validate attribute rates, states of drivers, or equivalent data sources. This feature should be used to develop or test an application rather than hardware drivers.
- Emulating hardware is done through a XML file generated from an existing hardware set up and pointing the tool **`xml_gen`** to the target context

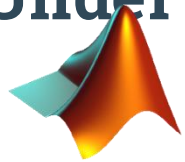


Enabling Zero-Copy Data Transfers in libiio via devmem

- Achieving **zero-copy** in libiio is challenging because traditional network paths involve **multiple copies**:
  - Device → Kernel buffer
  - Kernel → User buffer
  - User buffer → Socket buffer
  - Socket buffer → Network stack → Remote
- To reduce latency and CPU load, **zero-copy** aims to remove or coalesce these steps - ideally, exposing device memory directly to user or network-space without copies.

- The devmem virtual network interface in the Linux kernel:
  - Exposes a memory region (MMIO or physical) as a **network interface**.
  - Frames sent to/received from this interface **are backed by raw memory**, not traditional socket buffers.
  - Mimics a high-speed DMA device — with extremely low overhead.
- This makes it ideal for **libiio** where the system is architected to allow shared memory between:
  - The **ADC's DMA destination buffer**, and
  - The **memory region mapped to the devmem network device**.

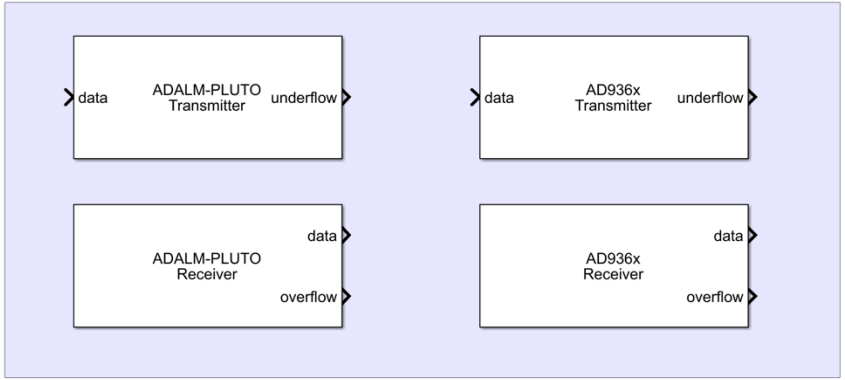
MATLAB and Simulink Blocks and System Objects Using IIO Under the Hood



MATLAB

Simulink

```
>> rx = sdrrx('AD936x');
>> rx = sdrrx('Pluto');
rx =

  comm.SDRRxPluto with properties:

   Main
                DeviceName: 'Pluto'
                   RadioID: 'usb:0'
           CenterFrequency: 2.4000e+09
                GainSource: 'AGC Slow Attack'
            ChannelMapping: 1
        BasebandSampleRate: 1000000
            OutputDataType: 'int16'
           SamplesPerFrame: 20000
           EnableBurstMode: false
     ShowAdvancedProperties: false

   Show all properties
```



35

Interesting ...

- IIO/libiio supports I/O + running locally (targeting) with little/no changes.
  - True RPC interface
- Sensor / ADC/DAC support from multiple vendors.
- Spans high speed (20+ GSPS) ADC/DAC and precision (32-bit 1 Hz)
- Spans payload and control
- Ethernet (1/10/1000 Mbps + 10G + 100G)
  - RNDIS/CDC-NCM/CDC-EMC,
  - Serial (UART, JTAG, Bluetooth, USB),
  - USB supported
- Network discovery with ZeroConf / DNS-SD (across Windows / Linux / Mac)

Popular IIO clients
(besides MATLAB/Simulink)

- GNU Radio
  - https://www.gnuradio.org/news/2022-01-17-gnuradio-v3.10.0.0-release/ (gr-iio)
- SDRangel
  - https://www.sdrangel.org/ (supports PlutoSDR, which is also IIO, would now be straightforward to support other devices).
- Custom applications in C#, Python, or C
  - C/C++ : https://libiio.github.io/libiio/master/libiio/index.html
  - C# : https://libiio.github.io/libiio/master/csharp/index.html
  - Python: https://libiio.github.io/libiio/master/python/index.html
    - ADI has created a more "pythonic" / Object-oriented interface based on the above for specific devices:
    - https://analogdevicesinc.github.io/pyadi-iio/
- Command line tools (read/write/discover)
  - https://manpages.debian.org/bullseye/libiio-utils/index.html (also available for Windows/Mac)
- IIO Scope (runs on ARM/HDMI/DisplayPort, so can be completely host-free)
  - https://www.youtube.com/watch?v=LfCX8K0MNTg
- Scopy (ADI open source tool)
  - https://youtu.be/LCf-_iREESQ?t=259
- And many others...

Popular IIO devices

- Xilinx and Intel Development Kits
- Raspberry Pi
- ST Development Kits
- ADI Development Kits
- NXP Development Kits
- ADI Education kits : Pluto SDR & ADALM2000 instrument
  - high volume, so libiio is pre-installed in all linux distributions

Summary

- IIO Subsystem
  - Unified Linux framework for ADCs, DACs, IMUs, and more
  - Modular and scalable for embedded and industrial systems
- libiio for Remote Access
  - Simplifies host-side development
  - Enables remote data acquisition and processing over network
- Host-Based Algorithm Development
  - Leverages powerful user-space tools (e.g., Python, MATLAB, C++)
  - Reduces target-side complexity and speeds up iteration
- Performance & Integration Best Practices
  - Buffer tuning, trigger setup, and driver configuration
  - Strategies for low-latency, high-throughput data paths
- Real-World Applications
  - Raspberry Pi examples for rapid prototyping
  - Techniques apply to edge devices, test systems, and distributed architectures

Use IIO and libiio to create flexible, efficient, and scalable sensor data systems - from prototype to production.

THE LINUX FOUNDATION

OPEN SOURCE SUMMIT

NORTH AMERICA

Thanks