You can do everything with MultiUSRP, and for everything else, there is RFNoC!

Marian Koop – Principal Software Engineer

Neel Pandeya – Principal Wireless Solutions Architect

Jonathon Pendlum – Principal Technical Support Engineer

GRCon 26

Enter to Win a USRP B206mini-i!

Scan the QR code to enter for your chance to win.

NI Ettus USRP B206mini-i

No purchase necessary. One entry per person.
Winner will be selected at random and notified after the event.

EMERSON | NI

Agenda

- Intro to USRP & UHD
- Overview of MultiUSRP architecture
- Synchronization and Timed Commands
- Streaming
- Reference Power Level API
- Extension API
- Q&R

Intro to USRP & UHD

Incl. overview of common workflows

Universal Software Defined Peripheral (USRP™)

- USRP
  - Family of Software Defined Radios developed and marketed under the brands

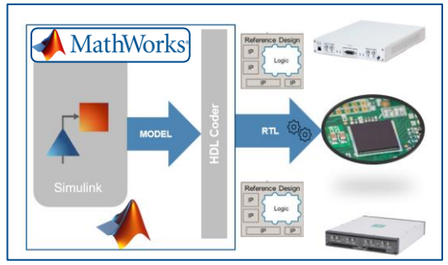
    Ettus Research and National Instruments (NI)

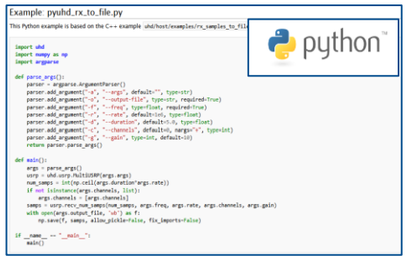
- About Ettus Research
  - Founded in 2004 by Matt Ettus
  - Acquired by National Instruments in 2010 and by Emerson in 2023.
  - Primary SDR R&D located in Austin, Texas, USA; Dresden, Germany

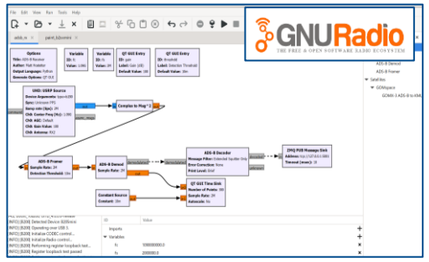
What is Software-Defined Radio (SDR)

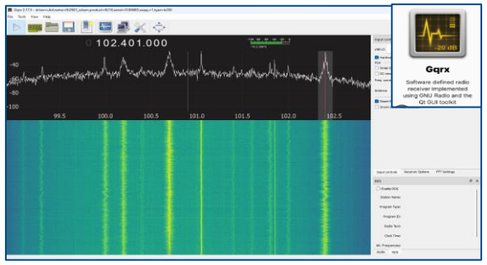
- A radio in which some or all of the physical-layer functions are implemented in software running on a microprocessor, and/or on an FPGA
- Algorithms from DSP and communications theory running as real-time software on a CPU and/or FPGA
- Software can be running on an embedded DSP chip (e.g., Analog Devices TigerSHARC, Texas Instruments C6400) or a general-purpose CPU (e.g., Intel x86, ARM Cortex-M)
- Joe Mitola first coined the term in 1991

Unified API Enabling Industry Standard Toolflows









UHD

RFNoC
Ettus
Research

EMERSON.

UHD in a nutshell

- 1 driver for all USRPs with 15+ years history
- Usermode library (DLL) to enable usage of all Ettus Research / NI / Emerson SDRs from user software
  - …includes utilities to configure and control devices
  - …includes FPGA bitfiles
  - …includes firmware code for various microcontrollers around our devices
- C++/C/Python APIs directly included
- Highly portable: Runs on Windows, Linux, MacOS, OpenBSD, x86, ARM, Risc-V, PowerPC,, laptops, desktops, servers, radio telescopes, spacecraft, …
- Natively ships with all relevant Linux distributions through our network of package maintainers
- Almost fully Open-Source (driver code, firmware, and HDL sources)
- Dual-license software: GPLv3 or custom commercial license upon request
- 2 major releases per year, Updates are regularly pushed to the master branch every few weeks.

UHD 4.11 Release (Sept 2026)

https://github.com/EttusResearch/uhd/releases/tag/v4.11.0.0

Add support for the new USRP B310, a PCIe/Thunderbolt-based device which supports 2x2 RF channels at up to 100 MHz bandwidth per channel.

Add support for X410, X420 and X440 variants with locked FPGA

Wideband DDC/DUC for X4xx devices

Enabled phase repeatability for X420 in support of applications like beam steering, angle of arrival detection etc. through FPGA design modifications

Modernize SW platform

- Update underlying OE version on embedded devices to Scarthgap (5.0 LTS) and Linux kernel to version 6.12.
- Migrated the MPM RPC transport layer from mprpc (msgpack-RPC) to gRPC/Protobuf, impacting all embedded devices running MPM.
- Replaced many boost dependencies with newer C++ equivalents to enable faster compilation, less dependencies, more useful error messages when developing UHD yourself

UHD Licensing

- UHD and RFNoC are free software and Open-Source projects
  - UHD issued under GPLv3 license
  - RFNoC issued under LGPL license
  - All source code for host driver, MPM, FPGA, microcontroller firmware is available on GitHub
- Available:
  - Partial BoM (key components)
  - 2D mechanical drawings (PDF file)
  - 3D CAD models (STP files)
  - Nominal validation data for some devices
  - Schematics available on request, requires NDA
- Restricted:
  - Full BoM
  - Gerber files for PCBs

UHD Licensing (continued)

- NI also offers an alternative license for UHD and RFNoC
  - Enable non-FOSS designs and proprietary product distribution
  - Available only from NI
- Does not apply to GNU Radio
  - Only issued under GPLv3
  - NI does not own the copyright

USRP Product Portfolio Overview








	Ettus USRP B2XX	Ettus USRP B310	Ettus USRP E320	Ettus USRP N310 / N32X	Ettus USRP X310	Ettus USRP X410	Ettus USRP X440	Ettus USRP X420
Frequency	70 MHz – 6 GHz	350 MHz – 7.2 GHz	70 MHz – 6 GHz	3 MHz-6 GHz (N32X) 10 MHz-6 GHz	DC* – 8.4 GHz	1MHz – 8 GHz	30MHz – 4 GHz IF	10 MHz – 20 GHz
Bandwidth	56 MHz	100 MHz	56 MHz	200 MHz (N32X) 100 MHz (N310)	*160 MHz	400 MHz	Up to 1.6 GHz	Up to 1 GHz
Channels	1 Tx, 1 Rx 2 Tx, 2 Rx	2 Tx, 2 Rx	2 Tx, 2 Rx	2 Tx, 2 Rx (N32X) 4 Tx, 4 Rx (N310)	2 Tx, 2 Rx *4 Rx (TwinRx)	4 Tx, 4 Rx	8 Tx, 8 Rx	2 Tx, 2 Rx
Architecture	Integrated	Integrated	Integrated	Integrated	*Configurable w/ Daughterboards	Integrated	Integrated	Integrated
Streaming	USB	PCIe / TB3	10 GbE	10 GbE	10 GbE or PCIe	100/10 GbE or PCIe	100/10 GbE	100/10 GbE
Sync	2x2 MIMO	Up to 8x8 with direct sync	2x2 MIMO	Up to 128x128 (N32X) LO sharing	*2x2 MIMO	4x4 MIMO	8x8 MIMO Sync across devices	128x128++ phase synchronization via LO sharing
SW Support	GNU Radio, C++, Python, MATLAB, LabVIEW	GNU Radio, C++, Python, RFNoC, LabVIEW	GNU Radio, C++, Python, MATLAB, RFNoC, LabVIEW	GNU Radio, C++, Python, MATLAB, RFNoC, LabVIEW	GNU Radio, C++, Python, MATLAB, RFNoC, LabVIEW, LabVIEW FPGA	GNU Radio, C++, Python, MATLAB, RFNoC, LabVIEW, LabVIEW FPGA	GNU Radio, C++, Python, RFNoC, LabVIEW	GNU Radio, C++, Python, RFNoC, LabVIEW
Key Features	Low SWAP-C, Highly portable	GPU coprocessing, deployments, Low SWAP-C	Low SWAP, Embedded Deployable, Standalone	Stand Alone, Wide bandwidth, Multi-Channel Sync Ready (N32X)	*Configurable RF Front End, Programable FPGA	RFSoC based FPGA, 5G Ready, Wide bandwidth, Multi-Channel	RFSoC based direct sampling, phase coherency, wide bandwidth	FR3, X, Ku Band Phase Sync Wide Bandwidth
Category	Entry Level (up to 100MHz per channel)			Mid Range (~100MHz to 400MHz per channel)		High-End (400+MHz per channel)		

The USRP Ecosystem

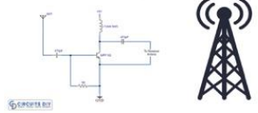
Frontends & HW Accessories

SIGNAL CRAFT TECHNOLOGIES

TMYTEK

...& more + DIY

Antenna Amplifier



Toolflow Support

C++ · Python · MATLAB · GNU Radio · RFNoC Ettus Research · LabVIEW

Comms Stack Integration

OPEN AIR INTERFACE

SRS SOFTWARE RADIO SYSTEMS

System Integration

allbesmart

pixus TECHNOLOGIES

AVDOR HELET

NOFFZ TECHNOLOGIES

EMERSON.

USRP and UHD Resources

Product Pages:

- https://www.ettus.com/products/
- https://www.ni.com/en-us/shop/category/software-defined-radios.html

UHD Binary Installer:

- https://files.ettus.com/binaries/uhd/latest_release/

Documentations:

- UHD User Manual: https://uhd.readthedocs.io/en/latest
- Gettings Started Tutorials: https://ni.com/usrp-getting-started
- Knowledge Base: https://kb.ettus.com

6th Edition NI Wireless Handbook

- https://www.ettus.com/6th-edition-wireless-research-handbook/
- Lots of customer success stories using NI SDR for research and prototyping
- Free to download
- Let us know if you‘d like to be featured in the next version!

Overview of MultiUSRP architecture, with core elements

What is MultiUSRP and how does it relate to USRP and UHD?

Common Acronyms used during this workshop:

- USRP – Universal Software Radio Peripheral
- UHD – USRP Hardware Driver
- MultiUSRP – Multi-Architecture USRP API
- RFNoC – RF Network on Chip
  - Digital Hardware Architecture used by 3rd and later generation USRP
  - UHD driver implementation for RFNoC based devices
  - API for dynamic FPGA configuration of RFNoC based devices
- CHDR – Hierarchical Datagram (pronounced like cheddar cheese)
  - Compressed Header of radio transfer protocol used by RFNoC based devices

MultiUSRP – Common Use Cases

As an RF engineer, I want to interface with a USRP device to capture and analyze IQ data from the RF spectrum on a computer, so that I can perform signal characterization and diagnostics.

As an RF engineer, I want to use a USRP device to transmit IQ signal data at a specified RF frequency from a computer, so that I can perform a variety of RF signal generation tasks.

MultiUSRP Hello World

Python Specific Abstraction Functions for recv and send

(for implementation details see <src>/host/python/uhd/usrp/multi_usrp.py)

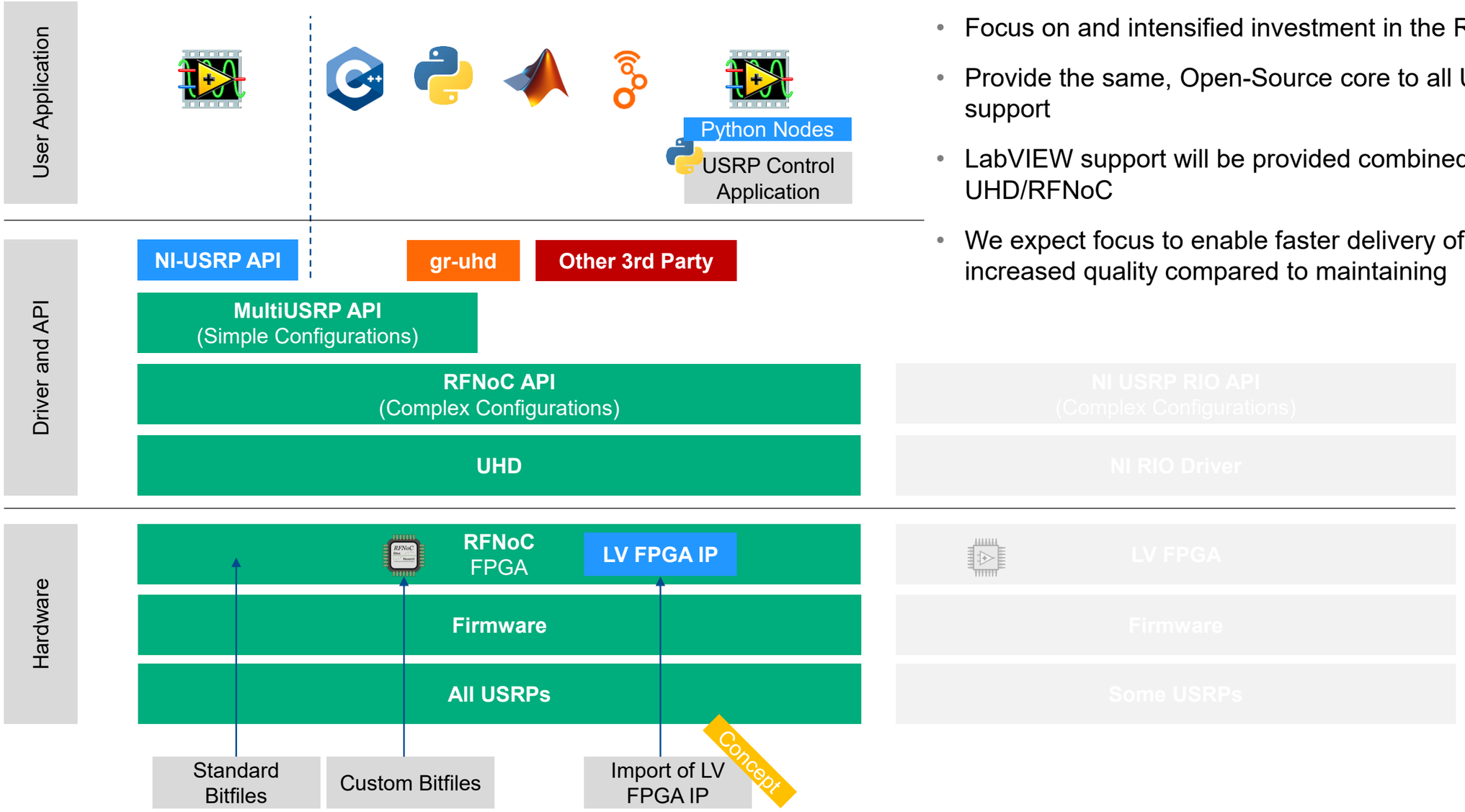
```
Import uhd
u = uhd.usrp.MultiUSRP(“addr=”)
samps = u.recv_num_samps(100, 800e3)
```

Common Core Functions

```
Import uhd
u = uhd.usrp.MultiUSRP(“addr=”)
u.set_rx_frequency(800e3)
streamer = u.get_rx_stream(uhd.usrp.StreamArgs("fc32", "sc16"))
recv_buffer = np.zeros((1, 100, dtype=np.complex64)
samps = streamer.recv(recv_buffer, uhd.types.RXMetadata())
```

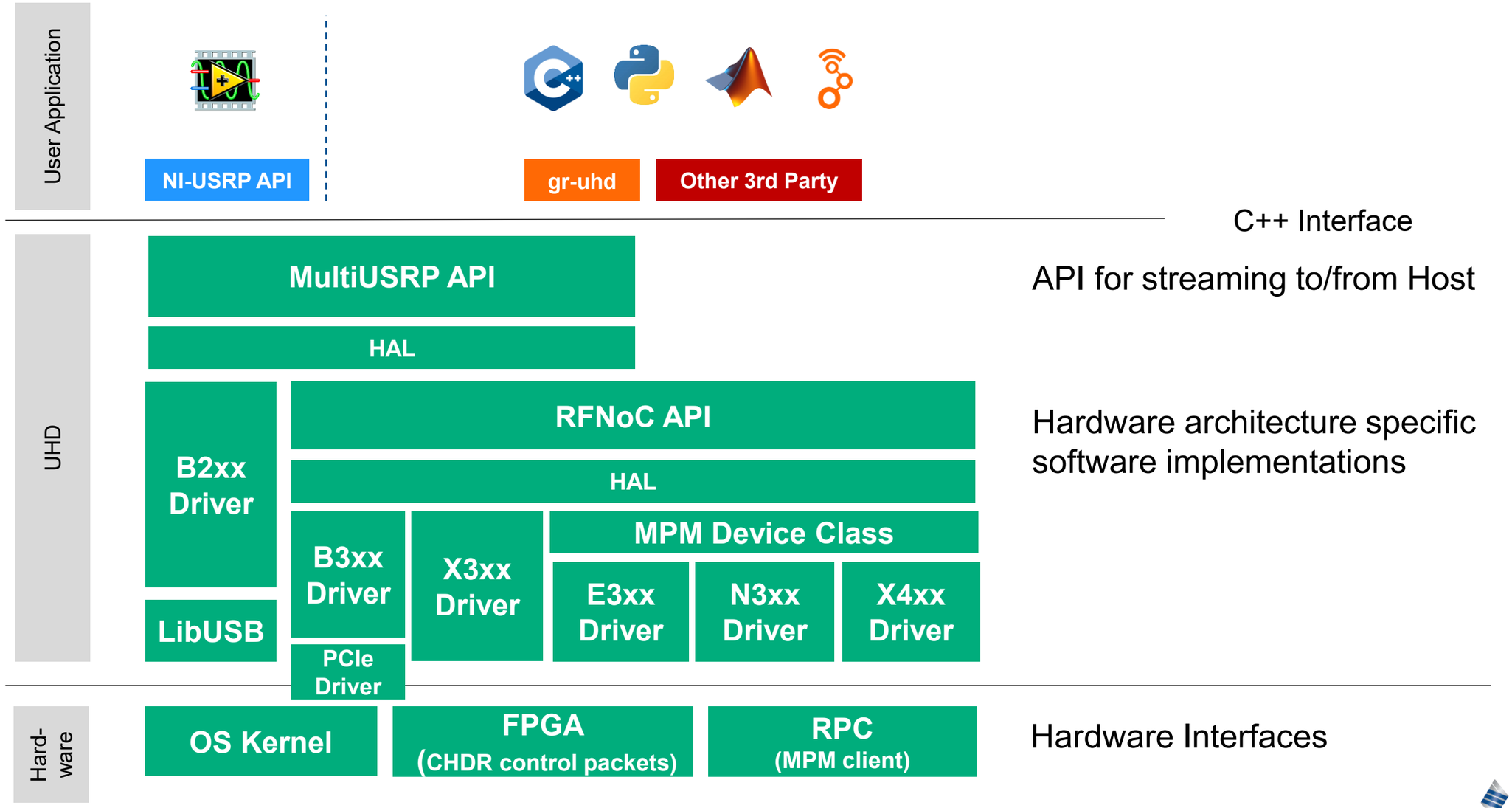
Open-Source
LabVIEW
3rd Party

USRP SW Overview (Future Devices)



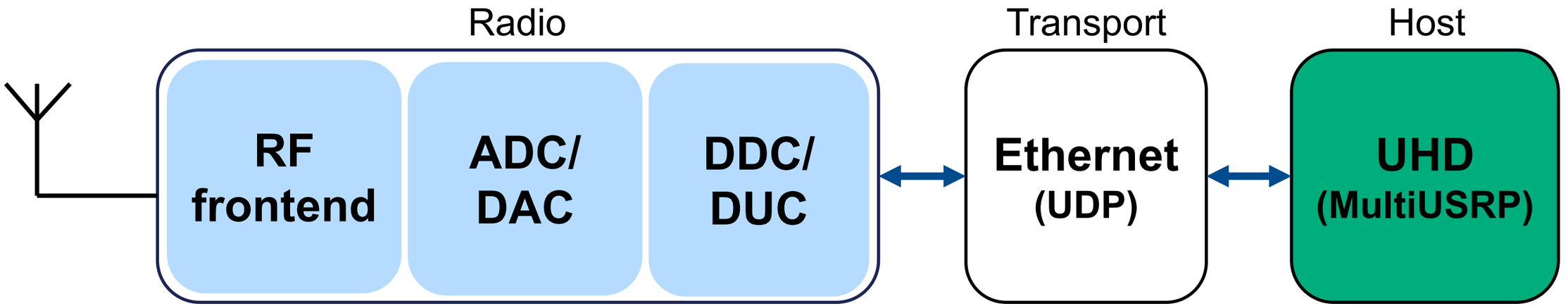
- Focus on and intensified investment in the RFNoC API
- Provide the same, Open-Source core to all USRP software support
- LabVIEW support will be provided combined with regular UHD/RFNoC
- We expect focus to enable faster delivery of features and increased quality compared to maintaining

UHD Driver Architecture



UHD – MultiUSRP API

A hardware-agnostic API for transmitting and receiving RF signals to and from USRP devices, using predefined FPGA personalities.



When MultiUSRP is not enough

- ReplayBlock*
- RX and TX Loopback in FPGA
- FPGA peer to peer applications
- Custom FPGA Design (a.k.a. use FFT block)
- Advanced LO control (like LO distribution)

* ReplayBlock is part of several predefined FPGA personalities, but as of UHD 4.11 still requires using the RFNoC API

Selecting the USRP FPGA Flavor

- Common Capability Feature Set
- Larger RF bandwidths require higher streaming data rates.
- 100GbE requires High-Performance PC

→Increasing variety of standard, prebuilt bitfiles and FPGA Flavors

UHD naming scheme for FPGA flavors encodes maximum RF bandwidth and streaming interface

<interface0>[<interface1>][_<bandwidth>], for example CG_400

Interface id	H	X	C	G	U	Q	A	W
Type	1GbE	10GbE	100GbE	All	Unused	N x 10GbE	Aurora	Whiterabbit
BW Limit	20MHz	200MHz	2GHz	n/a	n/a	N x 200MHz	*	n/a

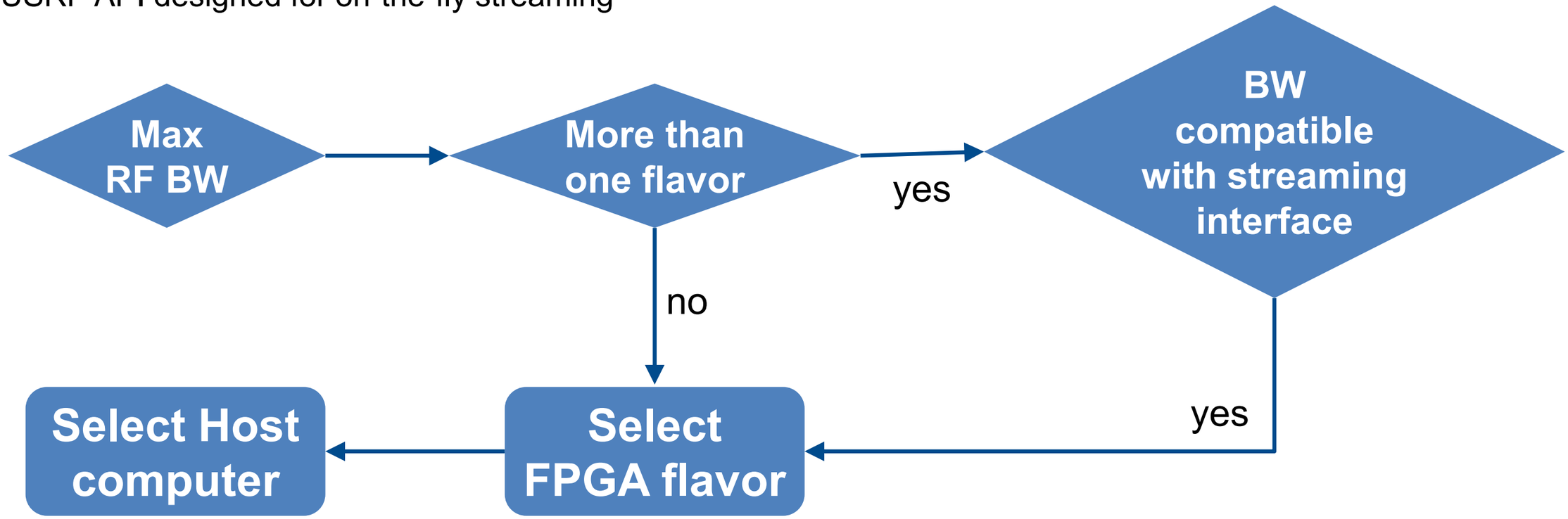
Special case: HG – 1GbE + 10GbE

* Depends on implementation

Selecting the USRP FPGA Flavor for MultiUSRP

* ReplayBlock is part of several predefined FPGA personalities, but as of UHD 4.11 still requires using the RFNoC API

MultiUSRP API designed for on-the-fly streaming



Multi Device sync and timed commands

Use Case

Use multiple phase-coherent channels within a single device or across multiple devices for applications such as

- MIMO
- Distributed MIMO
- Phased Arrays
- Beam-Forming (BF)
- Direction-Finding (DF)
- Phase-Coherent Recording

Limitations:

- Relative stability across devices lower than within a single device
- Multi-Device stability performance only specified for X440
- Multi-Device stability requires common clock and time source

Out of scope: Details about how phase coherence is characterized → see talk by Jan Schirok, at GRcon 25 on Wednesday Sep 10, 2025, 4:15 PM

Core concept

Define time at which coordinated (configuration) actions should happen

Define what (configuration) actions should happen

→ # of actions limited by FPGA instruction FIFO depth and device type

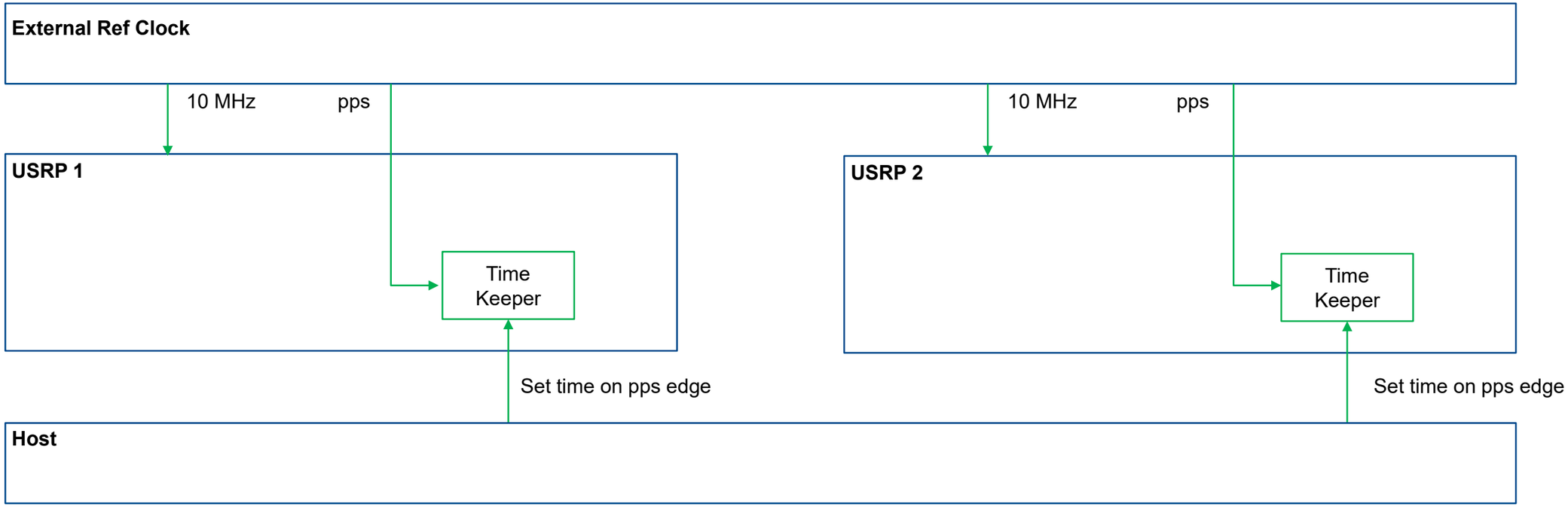
Commonly used for tune commands

Stream start time is also a timed command.

Time is defined relative to a time “reference”

- Time Now
- Time unknown PPS
- Time next PPS

Key Assumption



1. An external reference clock provides synchronous pps and 10 MHz ref clock to multiple USRPs
   - 10 MHz and pps are edge aligned and do not drift relative to each other
2. Time: timekeepers in all USRPs are set synchronously (edge of the same pps pulse) from host
   → common understanding of time across USRPs established
3. Frequency: all USRPs using a common reference, no drift between USRPs

Common pitfalls

- Defining timed commands with conflicting reference time
  → Late error on streamer
- Not accounting for timed commands to occur
  → code often requires sleep statements
- Not realizing that some devices do not (fully) support timed commands
  → case X410 tuning commands
- Defining too many timed commands at once
  → overflow timed command queue
- Set command time too close to current time (insufficient time to enqueue command)
  → account for finite code execution time (call get current times multiple times in a row will not return same time)

How to share clocks for phase coherency that suits your needs

24 Sept 2026, 09:45

30m

Mountain Ballroom (Talley Student Union)

Talk | Massive MIMO and ... | Main Track

Speaker

Jan Schirok (Emerson)

Description

Building multi antenna systems with many RF channels is a challenge that includes multiple aspects - apart from the RF frontend, cabling, the SDR itself, it usually requires sharing clocks between SDRs.

There are many ways to do that - from only using internal references, distributing 1 PPS (1 Hz) signals, sharing 10 MHz references, all the way to sharing multi GHz Local Oscillator clocks between devices. It could even include additional synchronization signals being shared.

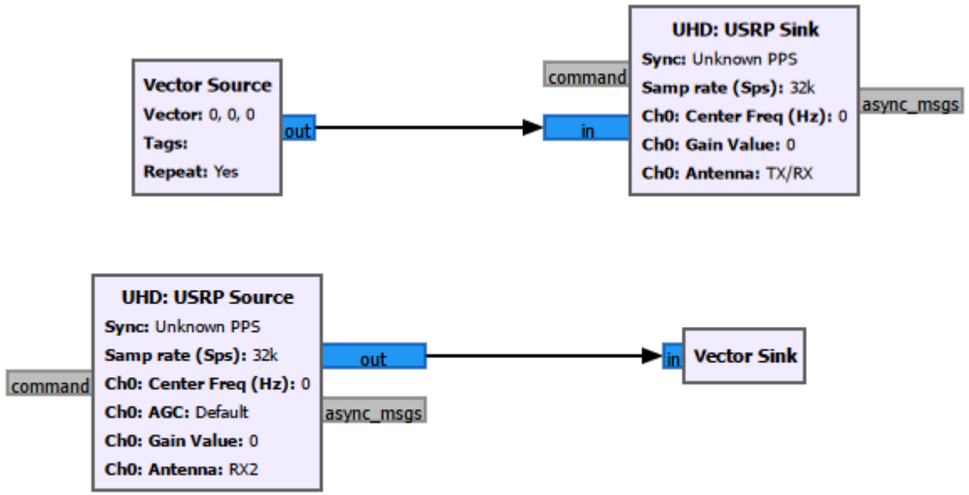
This talk will explain the various ways and complexities of sharing clocks and LOs as well as showing actually measured accuracies that can be achieved using some recent SDRs. It will also map those accuracies back to a few real world examples. That way - users can pick the best suitable approach and trade accuracy and clock distribution complexity.

Ethernet-Based Streaming

Streaming

What is it?

Transmission of data between a data source and a data sink



What streaming data rates are required?

- Assumptions:
  - IQ Data Sample size: 32bit (complex i16)
  - Max. Transport Medium utilization: 80%
- X310
  - Per channel: 32bit x 200MSps = 6.4 Gbps * 100/80 = 8 Gbps
- N3xx
  - Per channel: 32bit x 250MSps = 8 Gbps * 100/80 = 10 Gbps
- X440
  - Per channel: 32bit x 500MSps = 16 Gbps * 100/80 = 20 Gbps
  - Per device: 8 x 16 Gbps = 128 Gbps * 100/80 = 160 Gbps

Streaming

What kinds of streaming do we differentiate?

- Broadcast Streaming – Data Source sends data without regard to data sink's ability to consume data
- Stateful Streaming – Indicates that the stream maintains state about delivery, retries, and acknowledgments.
  - Flow-controlled Streaming – Data transmission between source and sink uses mechanism to “regulate” data flow
    - Stop-and-Wait Flow Control – Source transmits data one unit at a time, waiting until receipt is confirmed/acknowledged
    - Feedback-Based Flow Control – Relies on Sink to provide feedback (acknowledgement) of received data packets to source
      - Ex. TCP, RabbitMQ, gRPC
    - **Credit based Flow Control – Uses “credits” (available memory space) to regulate data transmission between source and sink**
      - **Ex. PCI-Express, NoC (Network on Chip), AXI-4, UHD**

Streaming - UHD

How?

- Uses the UDP Ethernet protocol
  → By design, this is broadcast streaming.
- Uses ratio transport protocol with custom flow control (https://files.ettus.com/manual/page_rtp.html
  - 1st, 2nd gen USRP (usrp1, N2xx): VRT
  - B2xx: legacy CHDR
  - 3rd or later gen USRP (b/e/n/x3xx, x4xx): CHDR
    - 64-bit header in addition to data

- Streamer,

  is limited to single interface (ip address), but

  multiple streamers may share a single interface

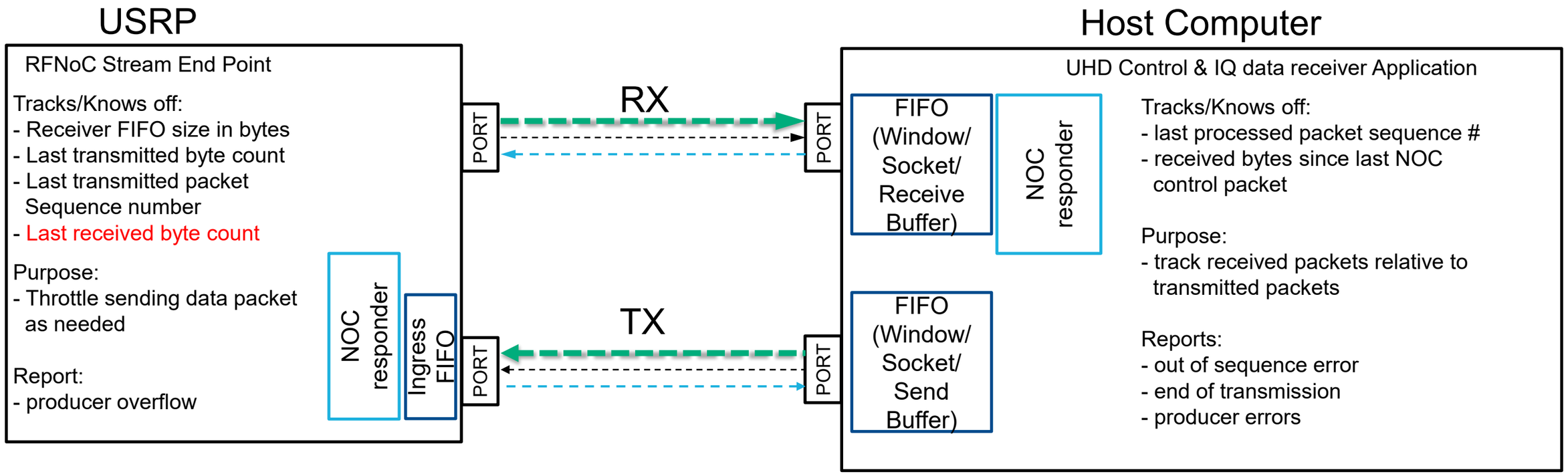
- Multiple Channels

  using Single Streamer is more efficient than streamer per

  channel, but Multiple Streamer can use multiple interfaces.

UHD 4.x -- Credit based flow control over lossy link

Goal: Prevent receive buffer overflows and track sequence order



Data packet with CHDR header

Control packet with CHDR header

NOC control packet with CHDR header

UHD 4.x Streaming Architecture Pro/Con

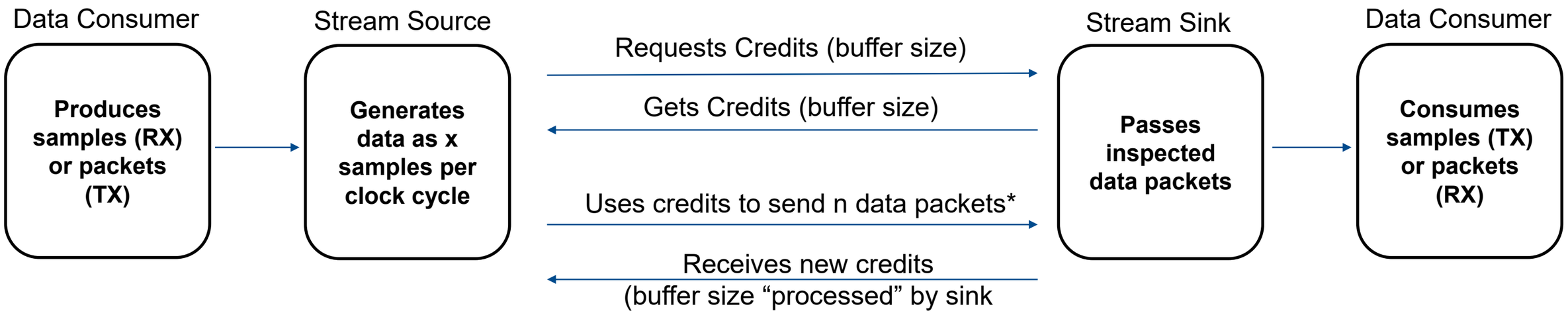
Pro

- UHD session management that including radio configuration and data transport management
- UHD abstracts flow control with goal of lossless transmission and transmission monitoring
  - flow control to improve streaming reliability by throttling RX/TX producers
  - reports out of sequence and lost packets

Con

- UHD driver needs to inspect and dissect every data packet
  - Impacts streaming performance
  - Differentiate between data and control packets
  - packet or unpacket IQ data to interface with consumers
- UHD control and streamer need to be on single host (host may have multiple 10/100GbE ports and devices)

Streaming – Credit-based Flow Control in UHD



Streamer also knows:
- Maximum Packet Size
  Transport Medium

* Source will continue to send packets until it runs out of credits,
  reports an overflow if data packet can not be send because of missing credits

Streaming – Credit-based Flow Control in UHD (cont.)

Stream Source (Sender) Tasks:

- If new (complete) packet is available
  - **and no credits are available, raise overflow error**
  - else decrease credit cache and send packet

- When new credits are received, update credit cache

Note to TX: When generation data to be transmitted to the USRP the host application is responsible to implement mechanisms to pass data only to the streamer after previous data has been transmitted. In contrast to RX (on the FPGA) were the stream source assumes that new samples will arrive at the clock per cycle interval

Streaming – Credit-based Flow Control in UHD (cont.)

Stream Sink (Receiver) Tasks:

- Read Packet from Buffer (Ingress, or Window)
  - UDP header contains packet size
- Inspect (CHDR) Header
  - Packet Sequence Number
- Compare Packet Sequence Number to against previous packet sequence number,
  - **if they are not consecutive, raise Sequence Error (TX) or Dropped Samples Error (RX)**
  - else pass packet to consumer (FPGA, or HOST application), and increase credit cache

- Every so often send source new credits (i.e. let source know that more buffer space is available)

Streaming – Credit-based Flow Control in UHD (cont.)

Data Consumer Tasks:

- TX:
  - Pass x samples per cycle on (to RFNoC blocks) at the time stamp defined in CHDR header
    - **If desired time is in the past, raise late samples error**
    - **If no samples are available, raise underrun error**

- RX:
  - Pass all samples in packet to host application,

    after (optionally) converting samples from “over the wire” to “CPU” format

Streaming – Credit-based Flow Control in UHD (cont.)

Data Producer Tasks:

- RX:
  - Collect x samples per cycle (from RFNoC blocks) and pass them to streamer

- TX:
  - **After** (optionally) converting samples from “CPU” to “over the wire” format pass them to streamer

Streaming – UHD Flow Control Error Types

UHD Flow Control Status Errors

- U – (Buffer) Underrun -- TX Streamer failed to transmit data fast enough
- O – (Buffer) Overrun -- RX Streamer failed to process transmitted data fast enough
- S – Sequence Error -- TX Stream sink received data packet with inconsistent/unexpected packet sequence number
- D – Dropped -- RX Stream sink received non-consecutive packets
- L – Late samples -- TX Data packet Timestamp is older than expected time spec

Streaming – UHD Streamer Config Parameters

UHD User Manual: Streaming Arguments (Stream Args)

Common streaming arguments:

- CPU data format – sample data type in user application (sample data type defined or expected by user program)
- OTW data format – sample data type used by the streamer implementation (sample data type used during transmission)
- SPP – samples per packet (limited by HW capabilities and software configuration)
- Throttle – Desired maximum transmission bandwidth utilization (1 eq 100%), useful only for RX
- Channels – List of channels for which to transmit data in a (single) stream

Streaming Performance – What influences it?

- Host Capabilities
  - CPU (number, clock rate, ..)
  - Memory (amount, speed)
  - Network Interface Hardware
    - Core Interface Capabilities (i.e. rated transfer speed)
    - Kernel Driver
  - Host Operating System
    - Process jitter
    - Multi-Tasking (be aware of foreground and background task)

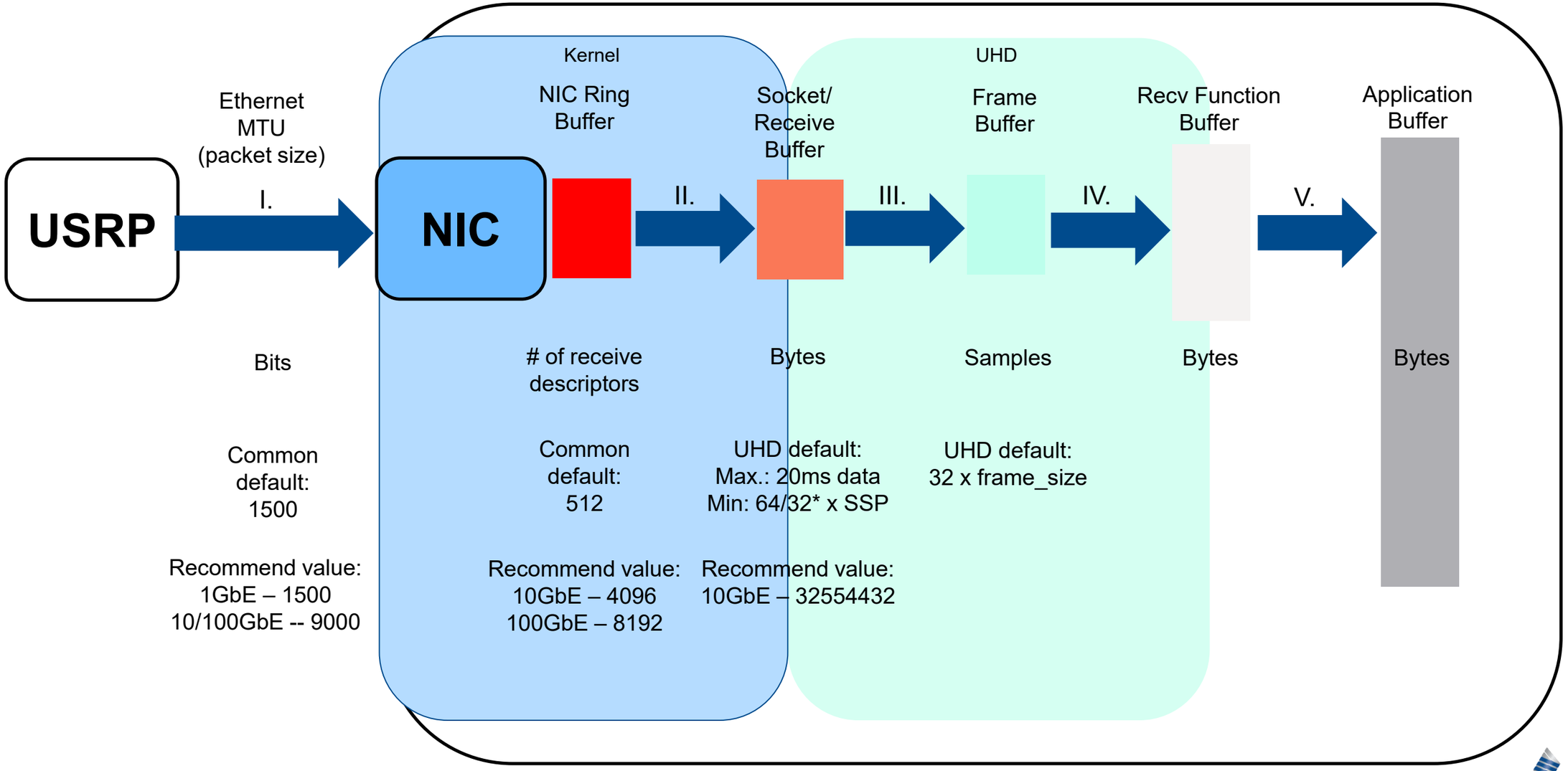
Streaming Performance – What influences it? (cont.)

- Software
  - Driver (UHD)
    - Transmission Overhead
    - Flow Control
    - Data Copies
  - Application (User program)
    - Code Architecture

167

33554432

Streaming – Data Handling Buffers (RX Case)



EMERSON

Streaming – RX Error mitigation

O – (Buffer) Overrun -- RX Streamer failed to process transmitted data fast enough

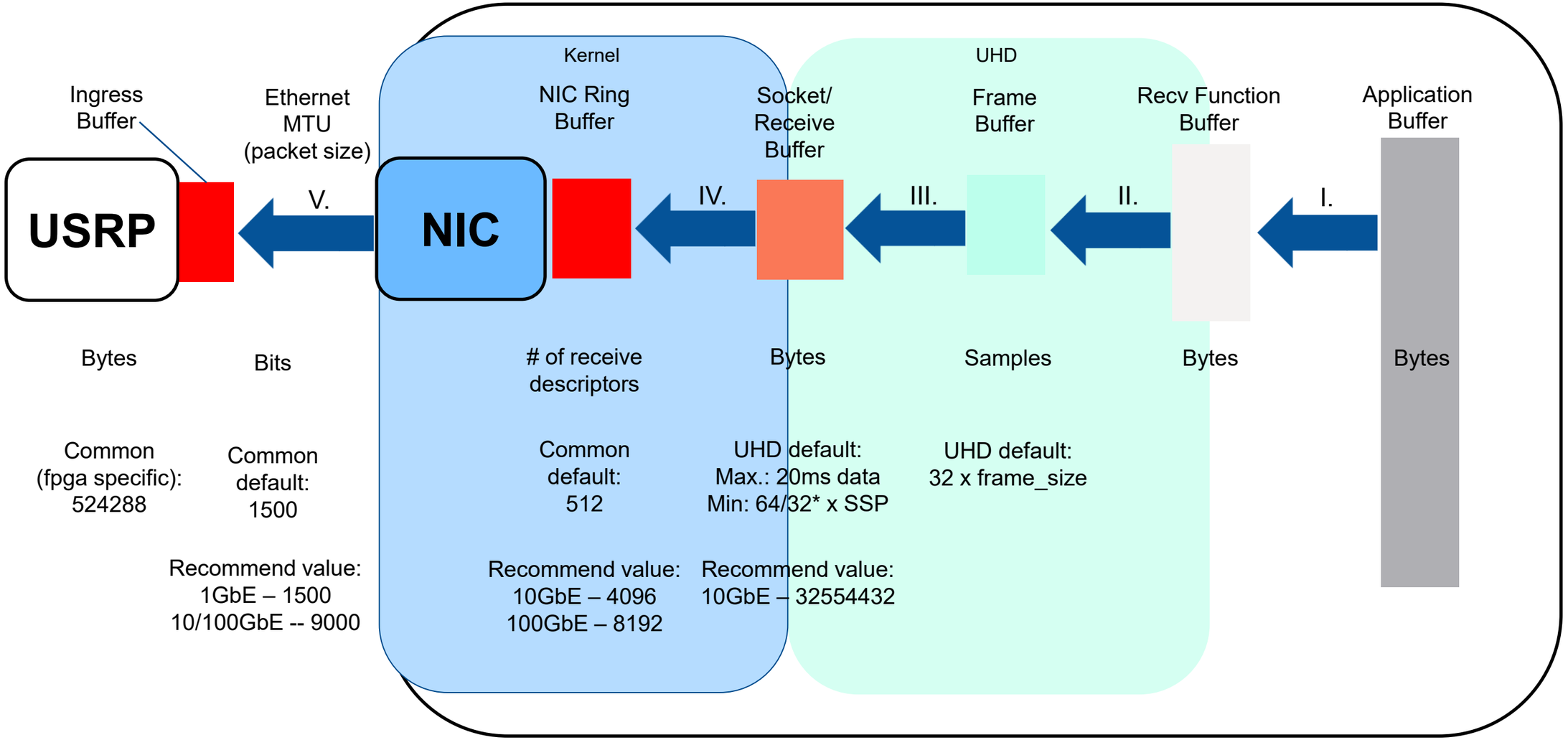
- Reduce CPU sensitivity to system interrupts
  - Increase buffers through which data packets “travel”
  - Reduce data fragmentation on data transfer to final user buffer (align ‘nsamples_per_buffer’ to multiple of frame size)
  - Avoid Datatype conversion

- Reduce randomness of CPU time slice
  - Isolate CPUs used by application (and UHD)
  - Use (more) deterministic CPU scheduler

D – Dropped -- RX Stream sink received non-consecutive packet

- Data packets get lost, or are received in the wrong order
- Most likely caused during ethernet transmission

Streaming – Data Handling Buffers (TX Case)



Streaming – TX Error mitigation

U – (Buffer) Underrun -- TX Streamer failed to transmit data fast enough

- Reduce CPU sensitivity to system interrupts
  - Increase buffers through which data packets “travel”
  - Reduce data fragmentation on data transfer to final user buffer (align ‘??’ to multiple of frame size)

- Reduce randomness of CPU time slice
  - Isolate CPUs used by application (and UHD)
  - Use (more) deterministic CPU scheduler

S – Sequence Error -- TX Stream sink received data packet with inconsistent/unexpected packet sequence number

- Data packets get lost, or are received in the wrong order
- Most likely caused during ethernet transmission

Streaming – TX Error mitigation

L – Late samples -- TX Data packet Timestamp is older than expected time spec

- Error means a specific start time for radio transmission start was used
  - Increase the relative time between “Sending” data, and transmission start
  - Move call to query reference time in code closer to uhd::tx_streamer::send call (see also section about timed commands)

    Note: Only the first packet is timed.

RX Raw (Remote) UDP Streaming

Why, what does it solve?

- Enable higher RX streaming rates
- Removes the CHDR packet inspection overhead

  and reduces related CPU utilization

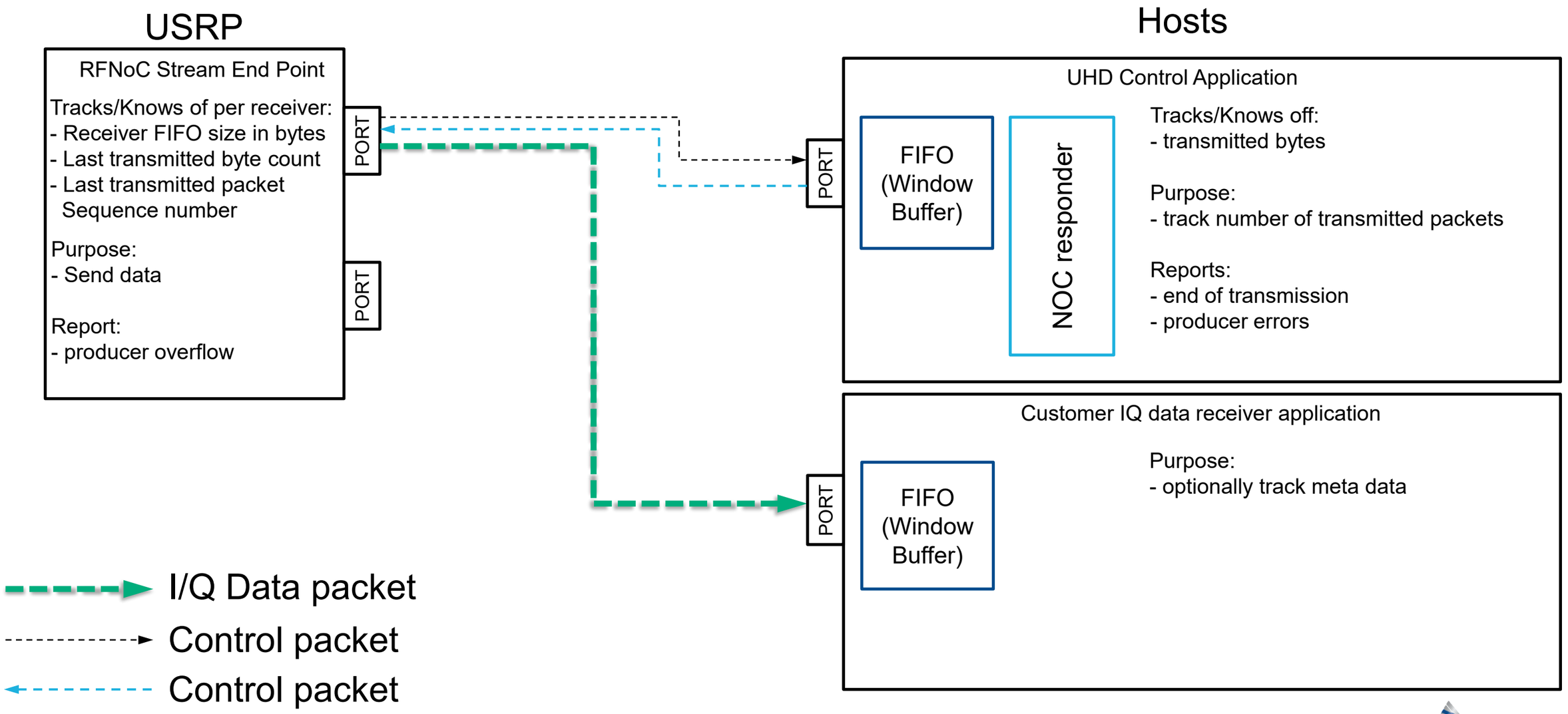
What are its limitations?

- No flow control (Broadcast Streaming)
  - No stream status (about dropped samples)
- User application responsibility to collect samples from socket buffer
-

Other benefit?

- Enables directing data to arbitrary network target (IP address)
- Enables single USRP to send data to multiple network targets
  - each Streamer (rx channel) can target different network target

RX Raw (Remote) UDP Streaming



RX Raw (Remote) UDP Streaming – How to use it?

How to enable RAW UDP Streaming?

- stream.args_t()
  - stream_mode: full_packet|raw_payload

How to configure data target address?

- stream.args_t()
  - dest_addr IP address of Remote Destination Interface
  - dest_port List of UDP Ports of Remote Destination interface
  - adapter USRP adapter to stream out of (e.g. ‘sfp0’)
  - dest_mac_addr Mac Address of Remote Destination Interface

RX Raw UDP Streaming – Example Review

https://github.com/EttusResearch/uhd/blob/master/host/examples/python/rx_to_remote_udp.py

Holoscan and USRP: A GPU and SDR case study

24 Sept 2026, 10:15

30m

Mountain Ballroom (Talley Student Union)

Talk | Hardware for GNU R... | Main Track

Speaker

Martin Braun (GNU Radio)

Description

Next to CPUs and FPGAs, it is obvious that GPUs play an important role in SDR system implementations. But how do we enable GPUs in a way that is compatible with the modular paradigm of GNU Radio?

In this talk, we shall present how Nvidia's Holoscan has approached this problem, specifically for use with Nvidia GPUs. We present joint work between NI and UT Austin in which we used Nvidia Holocan to build a high-bandwidth, real-time capable signal analysis system.

To conclude, we see which design choices from Holoscan could be useful for GNU Radio.

Logging

UHD Logging

- Defaults controlled by compile options → check ‘uhd_config_info --cxx-flags’
- Differentiate 3 Scopes (core, console, file) and 7 Levels (Trace .. Fatal)
- Default overridden via environment variables or uhd.conf file
  - UHD_LOG_LEVEL
  - UHD_LOG_CONSOLE_LEVEL
  - UHD_LOG_FILE_LEVEL
  - UHD_LOG_FILE

- Settings of Binary Release
  - UHD_LOG_MIN_LEVEL – 1 (i.e. Debug)
  - UHD_LOG_CONSOLE_LEVEL – 2 (i.e. Info)
  - UHD_LOG_FILE_LEVEL – 2 (i.e. Info)

Demo

- uhd_config_info --cxx-flags

- u = uhd.usrp.MultiUSRP()

Power Level API

Use Case

High-performance USRP use heterodyne (X310, X410) designs,

that have varying, (carrier) frequency and/or master clock rate dependent power loss behavior and

and dynamic gain control capabilities.

To combat this non-linear performance, the power API was created.

Reference Power Level API

Goal: User specifies desired absolute power level representing 0dBFS

How: Automated adjustment of gain use calibration tables

- User defines set_rx_power_reference()/set_tx_power_reference()
- UHD driver adjusts gain to maintain desired reference power
  - NOTE: UHD is limited by maximum available gain range
  - Gain table hierarchy (in order of preference)
    - Host ($HOME/.local/share/uhd/cal_data)
    - Device EEPROM
    - Driver (UHD)*

* average gain for product type, least accurate

Reference Power Gain Table

User tables created via calibration

/usr/lib/uhd/utils/uhd_power_cal.py --args <device args> -d <direction> --meas-dev <device class to use>

Default meas-dev classes are (see uhddev/host/python/uhd/usrp/cal/meas_device.py):

- RX: manual, rfsg, usrp,
- TX: manual, rfsa, visa

For an example custom meas-dev class see:
https://github.com/EttusResearch/uhd/blob/master/host/python/uhd/usrp/cal/visa.py

NOTE: Class attempts to auto-detect measurement device

GPIO API

Purpose and Use Cases

ATR (Automatic Transmit/Receive)

- GPIO state automatically follows radio state (Idle, RX, TX, Full Duplex)
- Deterministic FPGA-timed switching
- Ideal for T/R switches, LAs, PAs, external RF hardware

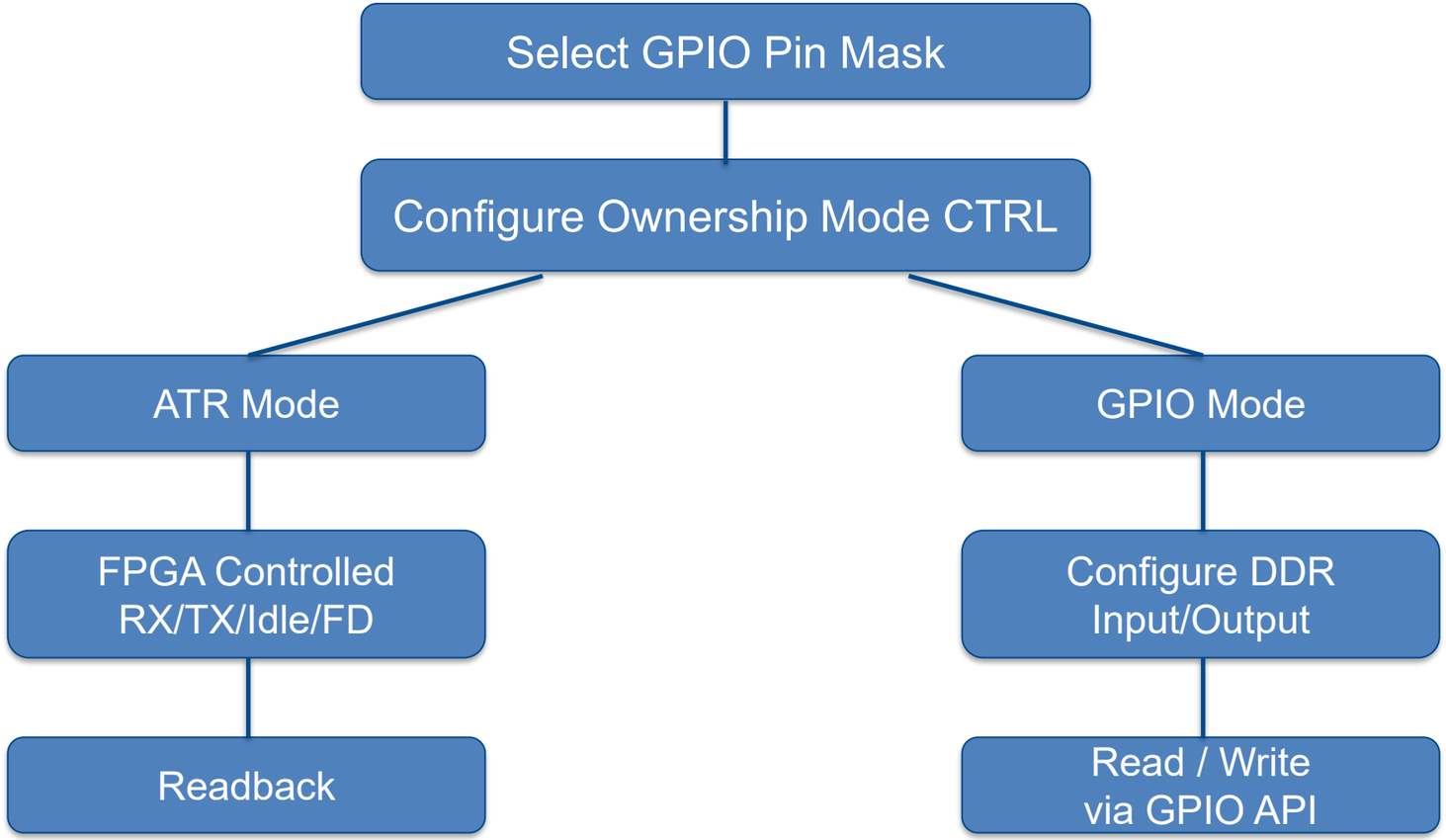
General Digital I/O

- Software-controlled input/output pins
- Read sensors, status signals, trigger external equipment
- Bit-banging and custom control applications

Key distinction:
ATR is event-driven by radio activity; GPIO is explicitly controlled by application software.

Note: Custom FPGA designs may also control GPIO from RFNoC Blocks

Programming Pattern



Device-Specific Differences

Device	Connector	GPIOs	Voltage	Bank Names[1]	Notes
B2x0	GPIO Header	8	3.3 V	FP0	Board-level GPIO
B20xmini	Internal Connector	8	3.3 V	FP0	Adapter required
B310	Mini-HDMI Type-C	10	3.3 V	?	
E31x	Internal 10-pin Header	6	3.3 V	GPIO0	Pin1 - source, up to 500mA
E320	Mini-HDMI Type-C	8	3.3 V	GPIO0	
N3xx[2]	DB15	12	3.3 V	FP0	Not designed for high currents, less than 5 mA per pin
X3xx	DB15	12	3.3 V	GPIO	Not designed for high currents, less than 5 mA per pin
X4xx	2× HDMI Type-A	24 (12+12)	1.8 / 2.5 / 3.3 V	GPIO0, GPIO1	Optional source pin with up to 450mA with overcurrent protection

[1] use get_gpio_banks() function to query available banks and their names
[2] not exposed on N321

Mixing ATR and Manual GPIO

```
# 1. Select ATR-controlled pins
usrp.set_gpio_attr(" FP0", "CTRL", 0x6, 0x6)

# 2. Configure manual GPIO direction
usrp.set_gpio_attr(" FP0", "DDR", 0x4, 0x4)
usrp.set_gpio_attr(" FP0", "DDR", 0x0, 0x5)

# 3. Program ATR state values
usrp.set_gpio_attr("FP0", "ATR_0X", 0, 0x6);
usrp.set_gpio_attr("FP0", "ATR_RX", 0, 0x6);
usrp.set_gpio_attr("FP0", "ATR_TX", 0x6 0x6);

# 4. Drive manual outputs
usrp.set_gpio_attr(" FP0", "OUT", 0x4, 0x4)

# 5. Readback outputs
usrp.get_gpio_attr(" FP0", “READBACK", 0x5)
```

CTRL
↓
DDR
↓
ATR_*
↓
OUT
↓
READBACK

UHD Extension Framework

Use case

- Adapt UHD driver behavior to custom equipment attached to RF ports of USRP
- Options:
  - A) Alter Driver source
    - Pro: Freedom to do anything
    - Con: Assume maintenance responsibilities, Need to “re-apply” modification when UHD changes
  - B) Dynamic function override
    - Pro: Limited code base, does not alter UHD code, could continue to use “official” binaries, easy to adapt to new UHD releases
    - Con: limited functions that can be overridden, new version required when ABI changes

Extension Framework

Highlights

- Released with UHD 4.4
- Per Session activation
- Launch product: SignalCraft SC2430
- Supported by all USRP via MultiUSRP API
- Support for overriding/extending functions, for list of supported functions see https://uhd.readthedocs.io/en/latest/page_extension.html

- Shipping example to demonstrate concept

Example User: SignalCraft Signal Conditioning Module SC2430

- 4 channel RF front-end solution that provides signal conditioning and amplification for Software Defined Radio (SDR)
- Designed for use with USRP X410, but could also be integrated with other SDR types
  - When used with USRP X410, uses UHD Extension Framework
  - SCM controlled via the HDMI physical interface and binary protocol
  - Code: https://github.com/SignalCraftTechnologies/SC2430-UHDExtension.git

\<install-path\>/share/uhd/modules directory

Review example

https://github.com/EttusResearch/uhd/tree/master/host/examples/extension_example

Install created extension module to:

<install-path>/share/uhd/modules

Use MultiUSRP as normal (and review DEBUG/INFO log)

Accessing RFNoC features

X4xx SPI Controller

Why is this needed?

Access to optional features, that are not part of common (across all USRP families) feature set,

see uhd::features::discoverable_feature Class Reference

Review example https://github.com/EttusResearch/uhd/blob/master/host/examples/python/spi.py

Debugging CHDR Messages

CHDR Packet Inspection with Wireshark

Purpose: Decode RFNoC CHDR traffic between UHD applications and DUTs.

UHD Application

↓

USRP Device

↓

CHDR Packets

↓

Ethernet Capture

↓

Wireshark + CHDR Lua Dissector

↓

Decode Data, Flow Control, RFNoC Commands, Responses

Common Use Cases

- RFNoC graph debugging
- Control transaction analysis
- GPIO register verification
- Stream troubleshooting
- Custom RFNoC block development

Dissector location:

uhd/tools/chdr_dissector/lua

Installing Wireshark and the CHDR Lua Dissector

Setup required to inspect CHDR traffic.

1. Install Wireshark

   Linux:
   sudo apt install wireshark

   Windows:
   - Install Wireshark
   - Install Npcap (required)
   - Enable WinPcap-compatible mode if offered

2. Install CHDR Lua plugin

   Copy:
   uhd/tools/chdr_dissector/chdr.lua

   Linux:
   ~/.local/lib/wireshark/plugins/

   Windows:
   %APPDATA%\Wireshark\plugins\

3. Restart Wireshark

4. Capture traffic on the USRP Ethernet interface and verify CHDR decoding.

Q&A

RFNoC Shop Competition

Win a brand new USRP B310!

1) Submit a block to the NoC Shop by 03/10/27
2) The NI team will pick blocks in 2 categories:
   - Most creative block
   - Most impactful contribution
3) The top two winners will get a USRP B310!

Learn more

It’s that simple. Submit your block today

EMERSON | NI

Enter to Win a USRP B206mini-i!

Scan the QR code to enter for your chance to win.

NI Ettus USRP B206mini-i

No purchase necessary. One entry per person.
Winner will be selected at random and notified after the event.

EMERSON | NI