RF Network-On-Chip (RFNoC™) Specification

Version 1.0

RF Network-On-Chip (RFNoC™) Specification

About this Guide

This guide describes RFNoC, which is a heterogeneous processing framework used to implement high throughput DSP in the FPGA for Software Defined Radio (SDR) systems in an easy-to-use and flexible way.

Intended Audience

This guide is written for hardware and software engineers who want to become familiar with the RF Network-on-Chip (RFNoC™) architecture or want to develop intellectual property (IP) using the RFNoC™ architecture.

Release Information

The following changes have been made to this specification:

Version	Date	Changes
0.1	2/1/2019	First Revision
0.2	5/6/2019	- Updated CHDR format to add virtual channel number and remove user defined flags and 2 bits of metadata - Dropped redundant “ctrlport” signal name in Table 18 - Updated management NodeInfo response value - Renamed AXIS Raw Data to AXIS Payload Context
0.3	7/23/2019	- Added motherboard controllers - Added default properties
0.4	8/6/2019	- Added description of RFNoC block user interface reset behavior - Added “AXI-Stream Data” interface option to the Data-Plane - Updated port numbering for user interfaces (eliminated “m<p>_” and “s<q>_” in favor of concatenating multiple ports)
0.5	9/4/2019	- Update interfaces from Doxygen
0.6	11/20/2019	- Made various corrections and clarifications
0.7	8/28/2020	- Updated document title to make it consistent throughout - Updated copyright year - Added clarification about Initialize stream command behavior based on NumBytes/NumPkts field values - Added information on `SIDEBAND_AT_END` parameter - Added YAML names to NoC Shell generation options
1.0	10/1/2020	- Changed `version` from number to string in YAML

TABLE OF CONTENTS

1 Introduction

1.1 What is RF Network-on-Chip (RFNoC™)?

RFNoC is a heterogeneous processing framework that can be used to implement high throughput DSP in the FPGA, for Software Defined Radio (SDR) systems, in an easy-to-use and flexible way. RFNoC and GNU Radio can be used to implement heterogenous DSP systems that can span CPU-based hosts, embedded systems and FPGAs.

RFNoC can be used to implement DSP "flow-graphs" where DSP algorithms and IP blocks are represented as nodes in the graph and the data-flow between them as edges. RFNoC, which is a network-on-chip architecture, abstracts away the setup associated with the nodes and edges of the graph and provides seamless and consistent interfaces to implement IP in the FPGA and software.

1.2 RFNoC Basics

As a network-on-chip architecture, RFNoC employs the following design philosophies for its choice of topology, routing, flow and microarchitecture.

1.2.1 Components

RFNoC flow graphs have the following components:

- *NoC Block*: A core processing block that implements user-defined IP like DSP, radio communication, hardware communication, etc.
- *Stream Endpoint*: A block that serves at the starting point or termination point for a data or control stream.
- *Transport Adapter*: An abstraction for physical transports like Ethernet, USB, PCIe, etc. Transport adapters are typically specific to the hardware that RFNoC is running on.
- *Routers*: Modules that connect NoC Blocks, Stream Endpoints and Transport Adapters to allow the user to build a DSP flow-graph.

Each NoC block has two communication planes: 1) Data and 2) Control. The control plane is used for setup and configuration and is assumed to be a low-throughput transaction-based interface. The data plane is a high-throughput streaming interface for samples, bits, etc. It is possible to inject optional, high-throughput metadata into the data-plane.

1.2.2 Topology

The topology is defined as the set of connections between the various RFNoC components. The topology of an RFNoC network is completely user-defined, given that the network meets the bandwidth and resource requirements of the underlying hardware. RFNoC allows the user to connect their own DSP blocks to the available Ettus Research SDR-specific blocks in a flexible and arbitrary fashion to create any custom flow graph. RFNoC also provides the ability to

reconfigure the graph within certain user specified constraints. Reconfigurability can fall into the following categories:

1. *Run-time Reconfiguration*: A part of the topology can be modified at runtime by changing software settings or physical connections between USRPs and FPGA accelerators. Run-time reconfiguration allows the software application to change the topology dynamically.
2. *Build-time Reconfiguration*: A part of the topology is hard-coded into the FPGA image and requires an FPGA rebuild (or partial bitstream download using partial reconfiguration) to reconfigure.
3. *No Reconfiguration*: There are hard-coded connections, primarily due to hardware design decisions, that do not allow certain parts of the topology to be modified.

Run-time reconfiguration provides the most flexibility but has a higher implementation cost in terms of FPGA resources and upper limits on processing blocks. Build-time reconfiguration provides less flexibility but reduces some of the resource costs. RFNoC allows users to choose between build-time and run-time topology reconfiguration. Automated tools and scripts will allow users to make these tradeoffs in an easy-to-use way.

1.2.3 Routing

The routing backbone in RFNoC is responsible for moving data from block to block using a clearly defined strategy. RFNoC uses the following routing strategies for the control and data plane.

- *Source Routing*: A routing algorithm that chooses the entire path at the source. For source routing to be possible, the source must know every hop that a transaction will take and the local router port at each hop. This is different from, say, distributed or incremental routing, where the transit decision is taken locally at each router instead of globally.
- *Deterministic Routing*: If there are two paths from the source to the destination, then the source routing algorithm will pick the path deterministically.
- *(Data Only) Circuit Switched*: A circuit (a path between a source and destination) must be established and reserved when a stream between two ports on NoC blocks is active. When a circuit is reserved, the source port cannot talk to a different destination.
- *(Control Only) Packet Switched*: Any NoC block can send and receive control transactions from any other NoC block without restrictions. The source and destination are encoded in the packet.

1.2.4 Flow

The smallest unit of transfer in RFNoC is a packet or datagram, the Condensed Hierarchical Datagram for RFNoC (CHDR). Both data-plane and control-plane traffic is packetized in the CHDR format, and the packet-type is encoded within the packet. Data streams are always bidirectional. Within the FPGA, data flows in AMBA AXI4-Stream packets and uses the standard ready/valid flow control scheme (flit-buffer flow control). For lossy transports, the stream endpoint implements a high-level flow control scheme which is packet based (packet-buffer flow control).

1.3 The RFNoC Flow Graph

As shown in Figure 1 an RFNoC flow graph has the following major components

- NoC Blocks
- Stream Endpoints
- Transport Adapters
- Routing Core (Routers and Crossbars)

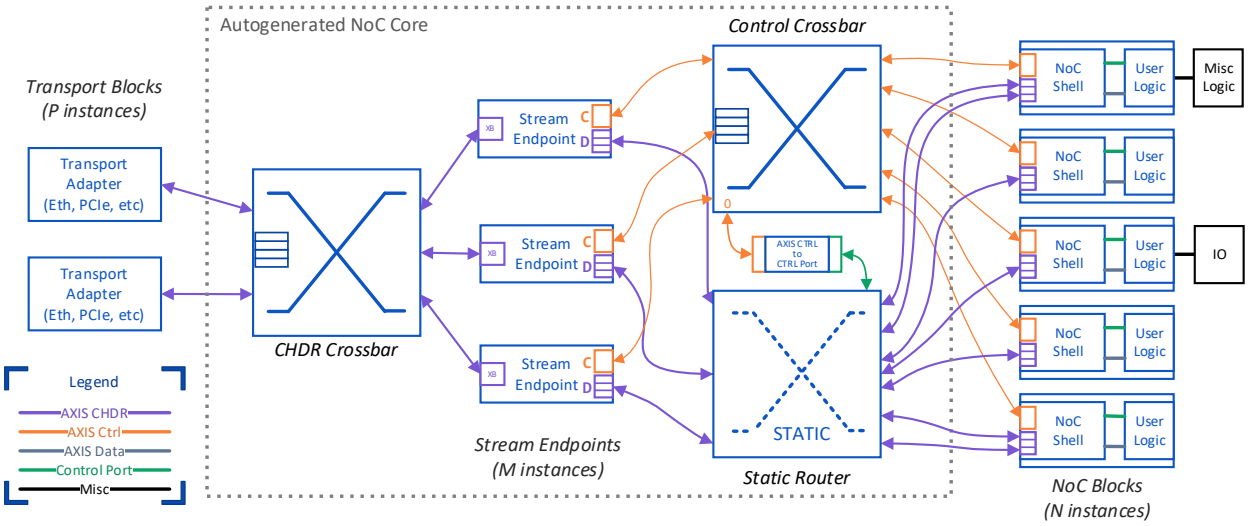

Figure 1: A typical RFNoC flow graph

1.3.1 NoC Block

A NoC block contains the core processing IP (user logic) sandboxed from the rest of the blocks and from the framework. The user logic interacts with the RFNoC infrastructure using the NoC Shell module. The NoC shell provides a separate control and data interface that the user logic can use to send and receive control transactions and processing data, respectively. The details of each interface will be covered in later sections. A NoC block may also interface with outside logic or IO that is unmanaged by RFNoC. An RFNoC flow graph can have at most about 1000 NoC blocks per device (if they fit in the FPGA). *This maximum number of ports in each FPGA is limited by a 10-bit address field which is shared for blocks, stream endpoints and transports.*

1.3.2 Stream Endpoint

A stream endpoint serves as the start and end for a unique sample stream. The number of stream endpoints in a USRP design must scale with the number of parallel streams of data to/from the device. A stream endpoint can exist in the FPGA or in software. A bidirectional stream can be initiated between any two endpoints dynamically at any point in the application. Streams can be destroyed and recreated without having to rebuild or partially reconfigure the FPGA image. RFNoC implements flow control between stream endpoints, so they can flow over any transport. An RFNoC flow graph can have a user-selectable number of stream endpoints. The number of stream endpoints is independent of the number NoC blocks. The stream

endpoint can optionally support multiple virtual streams that are multiplexed through the same physical transport. The multiplexing and demultiplexing will be performed by the framework using the “virtual channel” field in the packet header.

1.3.3 Transport Adapter

A transport adapter is a wrapper around a specific transport implementation like Ethernet, Aurora, PCI Express, etc. Transport adapters provide logic to enable RFNoC-formatted dataflow between two FPGAs, or FPGA and software in a hardware-transparent way. The number of stream endpoints is independent of the number of transport adapters; a transport is capable of multiplexing multiple streams of data.

1.3.4 Routing Core

The routing core handles connecting NoC blocks, stream endpoints and transport adapters. The routing core has three main routers:

- *CHDR Crossbar*: This crossbar is a full-bandwidth full-mesh dynamic crossbar. It connects the transport adapters to the stream endpoints. The CHDR crossbar enables communication within an FPGA between any two of its crossbar ports. This allows communication between two stream endpoints or between a stream endpoint and another FPGA through a transport adapter.
- *Control Crossbar*: This crossbar is a local crossbar, also full-mesh, but with reduced bandwidth. It allows control transactions to be sent between any two of its ports. This allows control transactions to be sent from software to a NoC block, from a NoC block to software, between two NoC blocks, or from a NoC block to another FPGA.
- *Static Router*: The static router encodes a fixed topology between data ports of NoC blocks. This topology can only be reconfigured by rebuilding the FPGA image. A static router requires significantly fewer FPGA resources than a dynamic router.

1.3.5 Example Topology

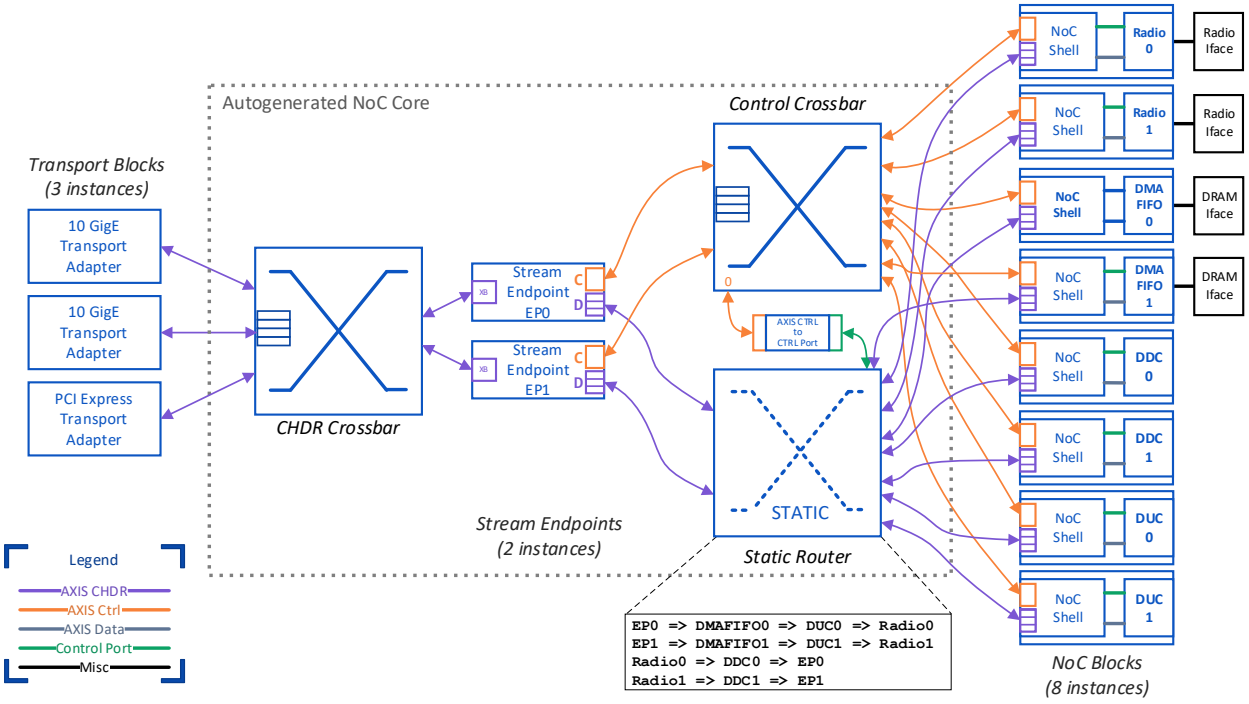

Figure 2: Example topology with for a multi_usrp compatible image (X310_XG)

Figure 2 shows an example topology for the USRP-X310 that can function with the multi_usrp API and a UBX daughter board (i.e., it has all the necessary radio and DSP blocks to implement the API). This design has:

- *3 transport adapters*: 2 for 10 GigE and one for PCIe
- *2 stream endpoints*: Each X310 supports 2 UBX daughter boards with a total 2 transmit and 2 receive channels. So, we instantiate 2 (bidirectional) stream endpoints.
- *8 NoC blocks*: We have 2 each of the radio, DMA FIFO, DDC and DUC blocks. Together they form 4 chains (subgraphs). These chains hook up two of the 4 ports (TX and RX) of the stream endpoints.
- *2 crossbars*: The 8 blocks in this image tunnel through 2 stream endpoints into the 5-port CHDR crossbar. The control crossbar has 11 ports for full control connectivity between blocks and endpoints.

1.3.6 Workflow

The primary goal for RFNoC is to allow DSP engineers to build heterogenous applications that may be comprised of standard blocks provided by Ettus Research, as well as custom user-authored blocks. The framework provides tools to 1) allow users to create custom blocks and 2) assemble an FPGA image and a software application that uses standard or custom blocks.

The general workflow for a user to build an RFNoC application thus is:

1. Partition the DSP/algorithm problem into software components and FPGA components (this can be done iteratively).

2. For the FPGA components, partition the problem into basic functions i.e. blocks.
3. Identify if any of the basic functions (blocks) are already available. Blocks can be found in the standard Ettus Research repositories or in third-party/open-source repositories.
4. Develop the FPGA and software source code for each new block using the tools provided by RFNoC.
    a. FPGA: Develop the core acceleration algorithm in Verilog, VHDL, SystemVerilog or Vivado HLS
    b. FPGA: Write testbenches for the block using the RFNoC framework
    c. Software: Write a block definition and an optional C++ controller to command and control the FPGA block from UHD software
5. Use the provided RFNoC tools to assemble an FPGA image that contains all the necessary blocks to implement the desired application.
    a. Connections between blocks can be fixed in the FPGA (for performance) or dynamic (for flexibility)
    b. Blocks can also be connected to transports on the USRP to build multi-FPGA applications
6. Once an FPGA image is ready, write an application in UHD (in C, C++ or Python) or GNU Radio to control and connect the dynamic blocks in the design to implement the desired application.

Usage Guidelines

- The user develops individual blocks, so the user interface in the FPGA and software will be abstracted at a block level.
- Blocks have a control and data plane, and those planes will be the primary interface points in the software and the FPGA.
- To build an application, the user must compose blocks in a specified topology, so the framework will provide tools to do so on the FPGA and provide APIs in the software to build a graph of blocks.
- Device specific details and the board support package for a USRP will be abstracted away by the framework.

NOTE: RFNoC has several features that are marked as "advanced" that may be disabled or not exposed in the standard interfaces for performance or efficiency reasons. The advanced features will allow users to implement more complex applications but that may require detailed understanding of the framework.

2 RFNoC FPGA Framework Overview

2.1 Basics

2.1.1 Block Capabilities

Fundamentally, an RFNoC block has three main types of interfaces:

- *Control*: A transaction-based interface that can be used for low-speed control through software or other blocks. The three basic transaction types are *register read*, *register write* and *bus sleep*. More complex transactions are possible, but most applications should be possible with the basic three. All transactions on this interface can be deterministic and executed at user-specified times.
- *Data*: A streaming interface that can be used for high-speed and low-latency data movement between blocks. This interface also supports deterministic and timed streaming with optional (advanced) capabilities to insert inline metadata in the stream.
- *External*: A block may need access to other IO in addition to control and data. Blocks that control USRP hardware (advanced) can have access to low-level pins. Blocks can also get access to time to implement hardware timed operations. More advanced blocks can get access to user-defined IO ports. *Most RFNoC processing blocks will not need the miscellaneous interfaces.*

Each block has *one* bidirectional slave (or master and slave) control interface, *zero or more* data ports and *zero or more* external IO.

RFNoC is a network-on-chip and has a packetized transport network. Utilities are available to abstract packets into simple interfaces (discussed later), however the understanding of the data flow and packet formats should allow users to build better and more efficient applications. RFNoC provides the following capabilities for the control and data planes:

2.1.1.1 Control-Plane Capabilities

- The control plane is transaction based. RFNoC has pre-defined transactions like reads, writes and sleeps, but it is possible to add more transactions (advanced). Transactions have a bit width of 32 bits and each transaction has a 20-bit address and a payload of up to eight 32-bit data words.
- Transactions are blocking and have an optional execution status.
- Transactions can be executed immediately or have an associated timestamp for deterministic execution or alignment with data samples.
- Any block can send transactions to any other block at any time. Blocks within an FPGA have connectivity through a control crossbar, so other blocks can be addressed with a 10-bit “port” whereas blocks on remote FPGAs can be addressed through a stream endpoint by specifying an “endpoint-ID” and a “port” on the remote FPGA.

2.1.1.2 Data-Plane Capabilities

- The data plane has a streaming interface where it is possible to stream “bursts” of “vectors” of “items”. An *item* is defined as a single atomic data word with a user-defined bit-width (e.g., a common item would be an RF data sample). A *vector* is a 1-dimensional collection of items. A *burst* is a collection of vectors.
- Data is received in packets which is independent of the items, vectors and bursts. The parameters of a packet, like the size, are hardware dependent and can be used to make low-level throughput/latency tradeoffs.
- Each packet can have user-defined metadata (advanced)
- Each packet can have a 64-bit timestamp (which is a counter in a time-base clock domain)
- It is possible to build a sequence of packets using an embedded sequence number field.

2.1.2 Integration with USRP Hardware

RFNoC provides seamless integration with USRP Hardware. As an SDR, a USRP has the following external input/output interfaces:

- ADCs/DACs
- RF Signal Chain Control
- Memory Interfaces (DDR, SRAM, etc.)
- Digital IO
- Transports (Ethernet, PCIe, etc.)

Each USRP will come equipped with NoC Blocks that seamlessly connect to the above IO. Transports will have corresponding transport adapters. It is possible to reassign that IO to other blocks in the design but that is an advanced feature.

2.2 RFNoC Packet Network

Before looking at the FPGA interfaces, it is important to understand how data flows between blocks and stream endpoints. With the provided RFNoC tools, it is possible to choose between a simple interface that abstracts the data movement or a low-level interface that gives the block full control (and responsibility).

2.2.1 CHDR Overview

The Condensed Hierarchical Datagram for RFNoC (CHDR) is a protocol that defines the fundamental unit of data transfer in an RFNoC network. As shown in Table 1, it has a header that encodes packet info, routing info, metadata and the data payload. CHDR is used as a transport protocol between stream endpoints. CHDR can handle control, data, flow control and status messages. The format is dependent on the width of the CHDR bus in the FPGA (CHDR_W). *NOTE: CHDR_W can be a power of 2 that is equal to or greater than 64 bits.*

#	Memory Layout <------ CHDR_W = 64 bits ------>								Required?
0	VC (6)	EOB (1)	EOV (1)	PktType (3)	NumMData (5) *Value=M*	SeqNum (16)	Length (16) *Value=L*	DstEPID (16)	Y
1	Timestamp (64)								N
2	Metadata[0] (CHDR_W)								N
.	...								.
M+1	Metadata[M-1] (CHDR_W)								N
M+2	Payload[0] (CHDR_W)								Y
.	...								.
M+N+1	Payload[N-1] (CHDR_W)								N

Table 1: Memory layout of a CHDR packet

The individual fields are described in detail in Table 2.

Field	Width	Description	Type
Virtual Channel (VC)	6	The virtual channel number for a stream. It is possible to have multiple virtual streams flowing over the same physical stream (EPID-pair). This field identifies the index of the virtual stream. The default value of this field is zero. *NOTE: Any virtual streams that are incorrectly addressed will go to port 0.*	Required
Delimiters (EOV/EOB)	2	Delimiter flags for the user logic to use. These bits are unused by the core framework but have the following definitions: • Delimiter[0] = EOV (End of Vector) • Delimiter[1] = EOB (End of Burst) *NOTE: Data in RFNoC has three kinds of delimiters: 1) Packets, 2) Vectors and 3) Bursts. A vector is a collection of packets (of items), and a burst is a collection of vectors.*	Required
PktType	3	The type of this CHDR packet. Can be one of the following: 0x0 = Management 0x1 = Stream Status 0x2 = Stream Command 0x3 = <Reserved>	Required

		0x4 = Control Transaction 0x5 = <Reserved> 0x6 = Data Packet without a Timestamp 0x7 = Data Packet with a Timestamp	
NumMData	5	The number of metadata words in this packet. Each metadata word is CHDR_W bits wide. If NumMData is zero, then the packet has no metadata. The maximum value for NumMData is 30. *NOTE: Metadata is considered to be an advanced feature of RFNoC, and its interpretation is assumed to be block-specific. The framework will provide the ability for the user logic to extract and insert metadata into a packet but the user logic in the block is responsible for defining its format.*	Required
SeqNum	16	Packet sequence number. The value shall start at 0 and increment by 1 for every packet of a given type in a stream. The counter shall roll over to 0 after 65535 (2^{16}-1). *NOTE: The sequence number is useful for detecting gaps and reordering issues in a stream. During error-free operation, the sequence number will increase monotonically (by 1) for every packet for each:* • *Stream (unique source and dest. endpoints)* • *Packet type* *The sequence should thus be independently monotonic for each stream and each packet type. A gap in the sequence number at any point is considered a sequence error.*	Required
Length	16	Length of the packet in bytes. This includes the header, timestamp, metadata and payload.	Required
DstEPID	16	The Endpoint ID of the stream endpoint that this packet is destined for. The EPID is used to make routing decisions. (The details of routing are covered in the following sections) *NOTE: EPID = 0 is reserved and may not be used*	Required
Timestamp	64	A 64-bit integer timestamp for the payload in the packet. This field is valid only when the packet type is “Data Packet with a Timestamp”	Optional
Metadata	Variable	User-defined metadata. These bits are unused by the core framework and their format is undefined. The	Optional

		definition of the format can be block-specific.	
Payload	Variable	User-defined payload *NOTE: Every CHDR packet must have at least one line of payload.*	Required

Table 2: CHDR field descriptions

The memory layout for various CHDR widths and configurations is shown below.

Byte	CHDR_W = 64
0	HEADER (64)
8	METADATA[0]
16	METADATA[1]
24	PAYLOAD[0]
32	PAYLOAD[1]
…	…
…	PAYLOAD[N-1]

Table 3: Memory layout for CHDR_W = 64 (Example without a timestamp and 2 metadata words)

Byte	CHDR_W = 64
0	HEADER (64)
8	TIMESTAMP (64)
16	METADATA[0]
24	METADATA[1]
32	PAYLOAD[0]
40	PAYLOAD[1]
…	…
…	PAYLOAD[N-1]

Table 4: Memory layout for CHDR_W = 64 (Example with a timestamp and 2 metadata words)

Byte	CHDR_W = 128	
0	TIMESTAMP (64)	HEADER (64)
16	METADATA[0]	
32	METADATA[1]	
48	PAYLOAD[0]	
64	PAYLOAD[1]	
…	…	
…	PAYLOAD[N-1]	

Table 5: Memory layout for CHDR_W = 128 (Example with a timestamp and 2 metadata words)

Byte	CHDR_W = 256 or higher		
0	RESERVED	TIMESTAMP (64)	HEADER (64)
32	METADATA[0]		
64	METADATA[1]		
96	PAYLOAD[0]		
128	PAYLOAD[1]		
…	…		
…	PAYLOAD[N-1]		

Table 6: Memory layout for CHDR_W = 256 (Example with a timestamp and 2 metadata words)

2.2.2 Data Packets

When the CHDR PktType field is 0x6 or 0x7, the payload is interpreted as a data packet. The data packet is the simplest type of CHDR packet because the format is flexible, and the payload is defined by the blocks generating and consuming it. When the PktType is 0x7, the header contains a valid timestamp. When the PktType is 0x6, the timestamp word is ignored. Note that when the PktType is 0x6 and CHDR_W is 64, there is no timestamp word and the first word of metadata or payload immediately follows the header word.

The stream endpoints separate control traffic from data traffic so that the *AXIS-CHDR Data* ports on the client side of the stream endpoint only carry data packets (see Figure 1). Data packets are designed to have the lowest overhead to enable low-latency and high-throughput streaming of samples.

2.2.2.1 Timestamps and Data Bursts

The exact meaning of the timestamp field in data packets is a block-dependent feature. For example, the radio will add the current timestamp to each outgoing packet but will interpret the timestamp on incoming packets as an instruction to start sending at this time. Other blocks may also have block-specific behavior regarding timestamps. To harmonize the usage of

timestamps, the following conventions should be used, where possible, to design blocks and/or software that uses timestamps:

- A burst is understood to be a contiguous string of samples or other data units. For example, the software might request 10000 samples from a radio, at a packet size of 1000 samples per packet. The burst will thus consist of 10 packets of 1000 samples each.
- The last packet of a burst **must** be tagged with an end-of-burst (EOB) marker.
- Assuming the burst is carrying timestamps, the first packet of the burst must carry the timestamp (the PktType field must be set to 0x7, and the 64-bit timestamp must be filled).
- The following packets of the burst *are not required* to carry a timestamp. The assumption is that timestamps can be calculated in the receiver, since the number of samples is known per packet.
- If mid-burst timestamps are set, then it is up to the downstream consumer to make use of them or ignore them.
    - Example: The DDC block will calculate timestamps internally within a burst. This is because the DDC typically comes directly after a radio, and thus the input to the DDC is predictable. The RX Streamer (in software) however *does* read all incoming timestamps and passes them on the user. This is because the data link between the FPGA and the host computer can be lossy (e.g., when using UDP), and thus, the host software will not assume it can internally calculate new timestamps.
- The first packet after an EOB must carry a timestamp again, if the new burst is timed.

The rationale for not requiring timestamps mid-burst is twofold: First, timestamps mid-burst are redundant, and thus leaving them out might make block designs simpler, and potentially reduce bandwidth usage. The second reason is due to the fixed-point nature of timestamps. Take the example of a radio block producing data at a rate of 200 Msps, which is in the same clock domain as the timekeeper, running at 200 MHz. Following the radio block is a fractional resampler which turns the 200 Msps into a 122.88 Msps stream. Due to the fractional relationship between input and output rates at the resampler, it will not be able to calculate mid-burst timestamps without rounding errors. The timestamp in the first packet, however, does not need to be converted, since the beginning of the packet keeps the same time regardless of the sampling rate. The redundancy of the mid-burst timestamps is thus used to avoid potential pitfalls of fixed-point rounding errors.

2.2.3 Control Packets

When the CHDR PktType field is `0x4`, the payload is interpreted as a control packet. The control packet encodes memory-mapped transactions. It has a variable length that can range from 16 bytes (no timestamp and NumData = 1) to 80 bytes (timestamp and NumData = 15).

Table 7 shows the format of the CHDR payload of a control packet. For simplicity, the rest of the CHDR packet is not shown. Note that a timestamp may be present in both the CHDR packet header and in the control packet contents. This simplifies the parsing of control and data packets.

#	Memory Layout <------ 64-bits ------>								Required ?
0	Reserved (16)	SrcEPID (16)	IsACK (1)	HasTime (1)	SeqNum (6)	NumData (4)	SrcPort (10)	DstPort (10)	Y
1	Timestamp (64)								N
2	Data[0] (32)		Status (2)	Reserved (2)	OpCode (4)	ByteEnable (4)	Address (20)		Y
3	Data[2] (32)				Data[1] (32)				N
…	…				…				N
9	Data[14] (32)				Data[13] (32)				N

Table 7: Memory layout of the CHDR payload of a control packet

A detailed description of the fields is listed in Table 8. Each control packet has the source and destination stream endpoint. The packet also has a source and destination port which allows addressing up to 1024 NoC blocks from each endpoint.

Field	Width	Description	Type
SrcEPID	16	The ID of the stream endpoint that this packet is originated from. *Note: EPID = 0 is reserved*	Required
IsACK	1	Is this an acknowledgement of a transaction completion?	Required
HasTime	1	A bit that indicates if the control transaction has the timestamp field	Required
SeqNum	6	Packet sequence number. For each master, the value shall start at 0, increment by 1 and roll over to 0 after 63 (2^6-1). This control-specific sequence number is independent of the CHDR sequence number. *NOTE: The sequence number may not be sequential over the wire in a multi-master case. It will be sequential in the masters' ingress queue because the slave and the transport modules will not modify it.*	Required
NumData	4	Number of 32-bit lines in the Data field *Note: NumData = 0 is reserved*	Required
SrcPort	10	The port within the source stream endpoint that	Required

<table>
<tr><td></td><td></td><td>this transaction originated from.</td><td></td></tr>
<tr><td>DstPort</td><td>10</td><td>The port within the stream endpoint that this transaction needs to go to</td><td>Required</td></tr>
<tr><td>Timestamp</td><td>64</td><td>If the transaction is timed, then this field signifies the start time of the transaction. The Timestamp word is not present if HasTime is 0.</td><td>Optional</td></tr>
<tr><td>Status</td><td>2</td><td>When IsACK is high, this field indicates the transaction completion status:<table><tr><th>Value</th><th>Status</th></tr><tr><td>0x0</td><td>OKAY (Transaction successful)</td></tr><tr><td>0x1</td><td>CMDERR (Slave asserted a command error)</td></tr><tr><td>0x2</td><td>TSERR (Slave asserted a timestamp error)</td></tr><tr><td>0x3</td><td>WARNING (Slave asserted a non-critical error)</td></tr></table></td><td>Required</td></tr>
<tr><td>OpCode</td><td>4</td><td>The operation code of this transaction. See OpCode definitions below.</td><td>Required</td></tr>
<tr><td>ByteEnable</td><td>4</td><td>A bitmask of the bytes to keep from the Data field</td><td>Required</td></tr>
<tr><td>Address</td><td>20</td><td>The byte address for the transaction</td><td>Required</td></tr>
<tr><td>Data[i]</td><td>Variable</td><td>The transaction data. Number of data values depends on the NumData field and their interpretation depends on the OpCode.</td><td>Optional</td></tr>
</table>

Table 8: CHDR Control field definitions

A control transaction is a memory mapped transaction that contains a 20-bit Address field and a 4-bit byte-enable field (with behavior similar to tkeep/tstrb in AXI4). It may have one to fifteen 32-bit data fields. A transaction can be timed, i.e., only executed when the sample timestamp matches a command timestamp. The OpCode determines the behavior of the transaction. All register transactions must be acknowledged after they are consumed. The packet size of the response will be the same as the packet size of the request. Using this information, the sender is responsible for flow controlling control transactions to ensure that the control packet FIFO is not overrun.

Note that the use of some control transaction features is block-dependent. For example, some NoC blocks may ignore ByteEnable and/or the Timestamp if those blocks do not support those features. This allows NoC blocks to be simpler if such features are not required.

Table 9 shows the meaning of the OpCode field values.

OpCode	Operation	Arguments	Description
0	Sleep	[0]: Stall cycles	Do nothing and stall the control endpoint for *Data[0]* clock cycles of the control interface clock.
1	Write	[0]: Data	Write *Data* to a single register at *Address* at all bytes *p* where by *ByteEnable*[p] = 1.
2	Read	[0]: Scratch	Read a single register at *Address*.
3	Read then Write	[0]: Data	Read the register at *Address then* Write *Data* to it at all bytes *p* where by *ByteEnable*[p] = 1.
4	Block Write	[0]: Data[0] .. [N-1]: Data[N-1]	Write Data[n] to registers sequentially at (*Address + 4n)* at all bytes *p* where by *ByteEnable*[p] = 1 where n = 0 .. N-1.
5	Block Read	[0]: Scratch [0] .. [N-1]: Scratch [N-1]	Read sequentially from registers at (*Address + 4n)* where n = 0 .. N-1.
6	Poll	[0]: Data [1]: Mask [2]: Timeout	Poll on *Address* until its value for all bits in *Mask* matches *Data&Mask*, or until *Timeout* cycles of control interface clock have elapsed. Acknowledge with CMDERR if timeout occurs, otherwise with OKAY.
7-9	Reserved	Reserved	Reserved
>9	User Defined	User Defined	6 opcodes are reserved for user-specific implementation.

Table 9: OpCode definitions for control transactions

2.2.3.1 AXI-Stream Control (AXIS-Ctrl) Interface

The CHDR Control packet is an example of a hierarchical packet format because the control payload itself forms another packet type, called AXIS-Ctrl, that is routed through the control infrastructure. AXIS-Ctrl is a 32-bit bus which is a serialized version of the payload of a CHDR Control packet. The stream endpoint will serialize CHDR to AXIS-Ctrl, where it is passed to the control crossbar. Each NoC Block will also receive and send control transactions/responses in the AXIS-Ctrl format. The stream endpoint will then de-serialize these transactions back to CHDR.

NOTE: The AXIS-Ctrl data width is always 32 bits, regardless of the value of CHDR_W.

2.2.4 Stream Status Packets [Internal Only]

NOTE: This is an internal-only packet, i.e., the NoC blocks will never see this type of packet. The RFNoC infrastructure is responsible for generating and consuming this packet type.

When the CHDR PktType field is `0x1`, the payload is interpreted as a stream status packet. Data streams in RFNoC are always bidirectional. Stream status packets always flow in the opposite direction of a data packet stream to communicate stream health and flow control information.

The following is a 64-bit serialized representation of the stream status packet. For CHDR widths larger than 64, serialization/de-serialization to 64 bits is done least-significant word first.

#	Memory Layout <------ 64-bits ------>				Required?
0	CapacityBytes (40)	Reserved (4)	Status (4)	SrcEPID (16)	Y
1	XferCountPkts (40)		CapacityPkts (24)		Y
2	XferCountBytes (64)				Y
3	StatusInfo (48)		BuffInfo (16)		Y

Table 10: Memory layout of the CHDR payload of a stream status packet

Field	Width	Description	Type
Capacity Bytes	40	The buffer capacity of the downstream endpoint in bytes	Required
Status	4	The current status of the stream. Possible values: `0x0` = Okay (No Error) `0x1` = Command Error (Command execution failed) `0x2` = Sequence Error (Sequence number discontinuity) `0x3` = Data Error (Data integrity check failed) `0x4` = Routing Error (Unexpected destination) `Others` = Reserved	Required
SrcEPID	16	Endpoint ID of the source of this message *NOTE: The endpoint ID of the destination is present*	Required

		in the CHDR header	
XferCount Pkts	40	Number of packets received by the destination stream endpoint.	Required
Capacity Pkts	24	The buffer capacity of the downstream endpoint in packets	Required
XferCount Bytes	64	Number of bytes received by the destination stream endpoint.	Required
StatusInfo	48	Extended information about the status. *NOTE: The format of this field is unspecified. It shall be used for diagnostics only.*	Required
BuffInfo	16	Extended information about the buffer state. *NOTE: The format of this field is unspecified. It shall be used for diagnostics only.*	Required

Table 11: Stream status packet field definitions

2.2.5 Stream Command Packets [Internal Only]

NOTE: This is an internal-only packet, i.e., the NoC blocks will never see this type of packet. The RFNoC infrastructure is responsible for generating and consuming this packet type.

When the CHDR PktType field is `0x2`, the payload is interpreted as a stream command. Data streams in RFNoC are always bidirectional. Stream command packets always flow in the direction of a data packet stream to trigger stream state changes.

The following is a 64-bit serialized representation of the stream status packet. For CHDR widths larger than 64, serialization/de-serialization to 64 bits is done least-significant word first.

#	Memory Layout <------ 64-bits ------>				Required?
0	NumPkts (40)	OpData (4)	OpCode (4)	SrcEPID (16)	Y
1	NumBytes (64)				Y

Table 12: Memory layout of the CHDR payload of a stream command packet

Field	Width	Description	Type
NumPkts	40	The number of packets associated with the operation. The exact interpretation of this field depends on the	Required

<table>
<tr><td></td><td></td><td>OpCode.</td><td></td></tr>
<tr><td>OpData</td><td>4</td><td>The data associated with the operation. The exact interpretation of this field depends on the OpCode.</td><td>Required</td></tr>
<tr><td>OpCode</td><td>4</td><td>A code that describes what needs to be done.
<table><tr><th>Value</th><th>Operation</th></tr><tr><td>0x0</td><td>Initialize stream
Flush buffers and reset stream state.

NOTE: When an Initialize stream command packet with NumBytes==0 and NumPkts==0 is received by the RFNoC infrastructure, one and only one stream status packet shall be sent in response. No flow control stream status packets shall be sent in response to incoming data on the given stream until an Initialize stream command packet with either NumBytes>0 or NumPkts>0 is received.</td></tr><tr><td>0x1</td><td>Ping
Trigger a stream status response at endpoint.</td></tr><tr><td>0x2</td><td>Resynchronize flow control
Use NumPkts and NumBytes to resync flow control.</td></tr><tr><td>Others</td><td>Reserved</td></tr></table></td><td>Required</td></tr>
<tr><td>SrcEPID</td><td>16</td><td>Endpoint ID of the source of this message.
NOTE: The endpoint ID of the destination is present in the CHDR header</td><td>Required</td></tr>
<tr><td>NumBytes</td><td>64</td><td>The number of bytes associated with the operation. The exact interpretation of this field depends on the OpCode.</td><td>Required</td></tr>
</table>

Table 13: Stream command packet field definitions

2.2.6 Management Packets [Internal Only]

NOTE: This is an internal-only packet, i.e., the NoC blocks will never see this type of packet. The RFNoC infrastructure is responsible for generating and consuming this packet type.

When the CHDR PktType field is `0x0`, the payload is interpreted as a management packet. Management packets are sent and received by internal RFNoC framework components for discovery and internal configuration. The following information can be discovered:

- The RFNoC protocol version and capabilities
- The physical connection topology including all transport endpoints and routers

A management packet can configure and discover information on the various nodes in the network. Nodes can be transport endpoints, crossbars and stream endpoints. The packet is a multi-hop transaction where operations are encoded in layers that are “peeled off” as they are consumed by the various nodes. A hop may contain several operations to execute (with a minimum of one). Each operation has an 8-bit opcode and a 48-bit payload. The interpretation of the payload is operation specific. The various opcodes defined below can allow the following:

- Discovering the RFNoC connection topology one node at a time (in DFS or a BFS manner)
- Configuring transport endpoints to setup EPID-specific settings
- Configuring stream endpoints with flow-control and other settings

Configuration is done via a basic memory mapped writes with a 16-bit address and 32-bit data. In the case of a route setup, the management packet can be configured to terminate at the stream endpoint. For other situations, it can be configured to return to the host.

The following is a 64-bit serialized representation of the stream status packet. For CHDR widths larger than 64, all MSBs are assumed to be zero and will be ignored. Management packets are NOT serialized.

#	Memory Layout <------ 64-bits ------>					Required?
0	ProtoVer (16)	CHDRWidth (3)	Reserved (19)	NumHops (10)	SrcEPID (16)	**Y**
1	OpPayload (48)			OpCode (8)	OpsPending (8)	**Y**
…	…			…	…	**N**
N-1	OpPayload (48)			OpCode (8)	OpsPending (8)	**N**

Table 14: Memory layout of the CHDR payload of a Route Setup packet

Field	Width	Description	Type
ProtoVer	16	RFNoC protocol version The top 8 bits represent the major version and the bottom 8 bits represent the minor version	Required
CHDRWidth	3	RFNoC CHDR bus width (CHDR_W) `0x0` = 64 bits	Required

<table>
<tr><td></td><td></td><td>0x1 = 128 bits
0x2 = 256 bits
0x3 = 512 bits
Others = Reserved</td><td></td></tr>
<tr><td>NumHops</td><td>10</td><td>Number of hops that this management packet will take before it is consumed completely</td><td>Required</td></tr>
<tr><td>SrcEPID</td><td>16</td><td>Endpoint ID of the source of this message</td><td>Required</td></tr>
<tr><td>OpsPending</td><td>8</td><td>Number of operations left to be executed for the current node/hop. Each node (hop) must have at least one operation associated with it.</td><td>Required</td></tr>
<tr><td>OpCode</td><td>8</td><td>Operation code (what to do)
0x0 = No-op
0x1 = Advertise
0x2 = Select Destination
0x3 = Return To Sender
0x4 = Node Info Request
0x5 = Node Info Response
0x6 = Config Write
0x7 = Config Read Request
0x8 = Config Read Response
Others = Reserved</td><td>Required</td></tr>
<tr><td>OpPayload</td><td>48</td><td>The payload associated with the specified operation (instruction). The format of the payload is operation specific.
<table>
<tr><th>Operation</th><th>Format</th></tr>
<tr><td>No-op</td><td>N/A</td></tr>
<tr><td>Advertise</td><td>N/A</td></tr>
<tr><td>Select Destination</td><td>Dest = Payload[9:0]</td></tr>
<tr><td>Return to Sender</td><td>N/A</td></tr>
<tr><td>Node Info Request / Response</td><td>DeviceID = Payload[15:0]
NodeType = Payload[19:16]
NodeInst = Payload[29:20]
ExtendedInfo = Payload[47:30]</td></tr>
<tr><td>Config</td><td>Address = Payload[15:0]
Data = Payload[47:16]</td></tr>
</table></td><td>Required</td></tr>
</table>

Table 15: Route define packet field definitions

Management Operations

- *No-op*: Do nothing. The minimum number of operations per hop is 1, and a no-op can be used to meet that requirement.
- *Advertise*: The operation is effectively a no-op but it asserts a strobe that advertises the passing management packet to the outside logic. An advertisement includes the associated source and destinations EPIDs.
- *Select Destination*: Select the downstream destination for this management packet. Useful for situations where a router is expected downstream but it has not been configured yet. The select destination command can be used to temporarily allocate a route to send this packet to the specified `Dest` port.
- *Return to Sender*: Turn the packet around and return it to the sender. The return command can be coupled with a Node Info Request or a Config Read Request to allow an upstream node to query data from a downstream node.
- *Node Info Request*: Request the current node/hop to return information about itself. This operation will route the packet back to the sender.
- *Node Info Response*: This is the response to the above info request.
- *Config Write*: Perform a Control-Port write using the specified Address and Data.
- *Config Read Request*: Request a read of the specified Address.
- *Config Read Response*: The read data for the last read request.

2.3 NoC Block User Interface

Figure 3 shows the anatomy of a NoC Block in the FPGA. It consists of two main components: 1) the user logic and 2) the NoC Shell. The NoC Shell is the user logic's interface to the rest of the RFNoC framework. A NoC Shell is custom generated for each block based on user-specified interface options. It is also possible to generate IO interfaces to outside logic from a NoC Block, but that feature is advanced. RFNoC provides a utility (see RFNoC ModTool below) to generate a unique instantiation of a NoC Shell that is custom for each block. Depending on the level of abstraction desired, for most interfaces, there is an option for a simple but potentially less featured interface and a low-level but full-featured interface.

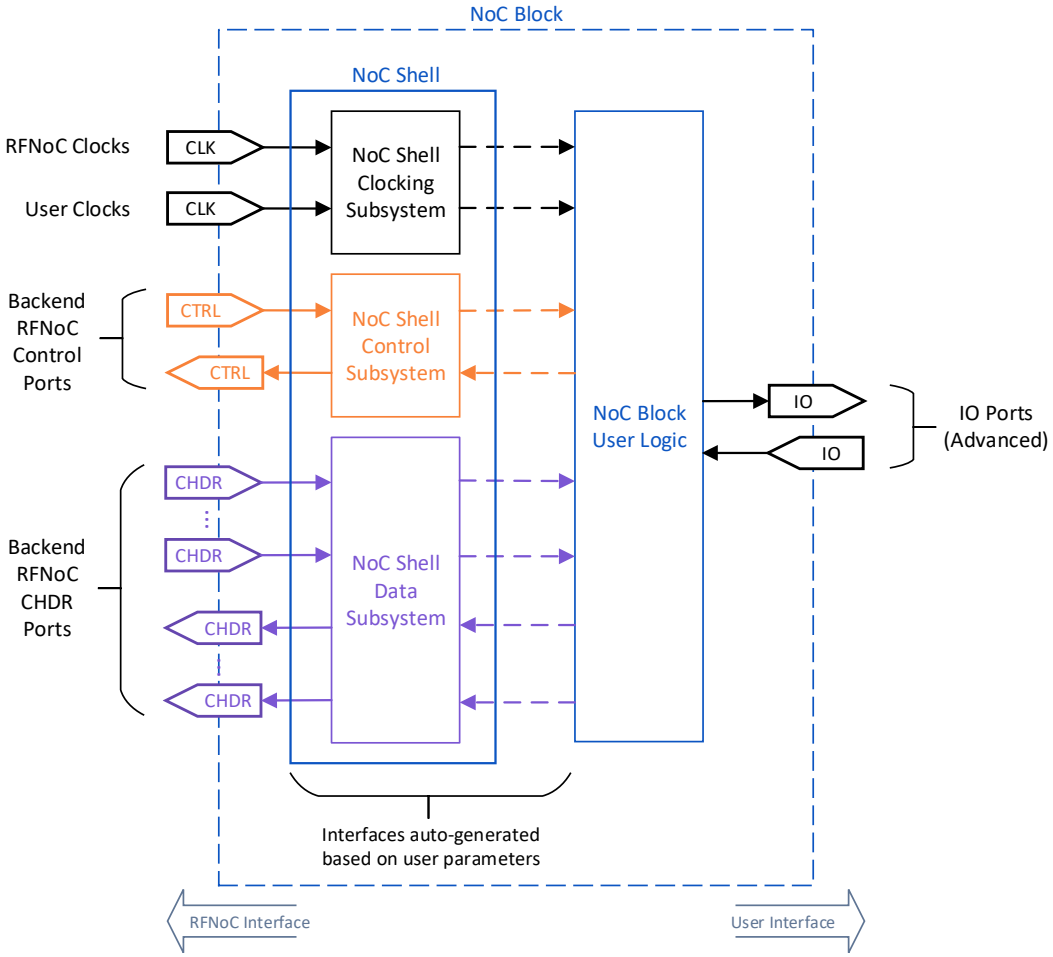

Figure 3: Anatomy of a NoC Block (FPGA)

2.3.1 Basic Signals

2.3.1.1 Bus Widths

Each block can choose the CHDR width that it wishes to support. A block will generally have a fixed CHDR width and a block can only be used in designs that use the same CHDR width. In an FPGA design, the CHDR widths of all blocks and the device must be the same.

2.3.1.2 Clocks and Resets

2.3.1.2.1 RFNoC Clocks

The following two clocks are always available for the user logic to use:

- `rfnoc_chdr_clk`
  This is the clock for the `rfnoc_chdr` port (described in Section 2.3.5).
- `rfnoc_ctrl_clk`
  This is the clock for the `rfnoc_ctrl` port (described in Section 2.3.5).

These are always-on clocks that will be used by the framework for data movement. Their frequencies are USRP device dependent.

2.3.1.2.2 RFNoC Resets

Two resets are exposed through the user interface, named `rfnoc_chdr_rst` and `rfnoc_ctrl_rst`. These resets are both synchronous to their respective clocks and are driven by the backend interface toward user logic. Both resets will assert for at least 32 of their respective clock cycles to ensure a sufficiently long reset for connected user IP. These resets should be used to reset user IP so that the entire block is reset when a reset is requested by the backend interface. A synchronizer may be used to import these resets to other clock domains, if needed.

2.3.1.2.3 User Clocks

If a block needs additional clocks, it is possible to add additional clock ports to a block. User clocks for a block must be driven by device clocks when a design is assembled. Frequency ranges can be specified on clocks to ensure that block requirements are met. RFNoC assumes asynchronous data processing so, it is not possible specify the phase or synchronization of optional clocks. If there is a need for low level synchronization with hardware or other blocks, then the advanced IO Ports must be used. These are described in Section 2.3.4.2.

2.3.1.3 NoC Shell Generator Options

RFNoC ModTool has the following options to generate the basic interface for a NoC block.

- CHDR Width (`chdr_width`)
  - Definition: Width of the CHDR bus
  - Options: 64, 128, 256, ...
  - Constraints: None
- Optional Clocks
  - Definition: An option that indicates if additional clocks are needed by the block
  - Options: A list of clock names and frequency ranges
  - Constraints: None

2.3.2 Control-Plane

The control-plane in the FPGA can be exposed using a low-level AXI4-Stream interface called *AXI-Stream CTRL* or using a simpler abstracted interface called *Control Port*.

2.3.2.1 AXI-Stream Control (Low-level Interface)

AXI-Stream Control (AXIS-Ctrl) defines an interface and a packet format to encode control transactions in a standard 32-bit wide AXI-Stream bus. Regardless of the CHDR widths, AXIS-Ctrl will always be 32-bit wide. The data transferred over this interface is identical to the payload of a CHDR control packet except for the top 32 bits of the first payload line. All other fields are identical. Table 16 shows the various fields of an AXIS-Ctrl packets formatted with a 32-bit word width. Note that the payload is identical to that of Table 7, except for the second line in the packet. The fields are described in Table 8 and Table 17.

AXIS-Ctrl packets traverse over the control network which consists of the control crossbar. This network is different for the typical CHDR network in RFNoC. It allows transactions to originate

from and terminate in any NoC block in the device, despite the static data connections. The host software can issue an AXIS-Ctrl transaction going to any FPGA block and any FPGA block can send a transaction to any other FPGA block or to software. It is also possible to communicate with blocks in different devices. These are defined as *remote transactions* and require the use of two additional fields, `RemDstEPID` and `RemDstPort`.

#	Memory Layout <------ 32-bits ------>						Required?
0	IsACK (1)	HasTime (1)	Seqnum (6)	NumData (4)	SrcPort (10)	DstPort (10)	Y
1	Reserved (6)		RemDstPort (10)		RemDstEPID (16)		Y
2	Timestamp[31:0] (32)						N
3	Timestamp[63:32] (32)						N
4	Status (2)	Reserved (2)	OpCode (4)	ByteEnable (4)	Address (20)		Y
5	Data[0] (32)						Y
…	…						N
19	Data[14] (32)						N

Table 16: Memory layout of an AXIS-Ctrl packet

Field	Width	Description	Type
RemDstEPID	16	Remote Destination Endpoint ID: The ID of the remote stream endpoint that this packet is destined towards. *Note: EPID = 0 implies that the transaction is local*	Required
RemDstPort	10	The port index of the crossbar downstream of the remote stream endpoint that this packet is destined towards.	Required

Table 17: Additional AXIS-Ctrl field definitions

For the NoC block interface, AXIS-Ctrl is a simple 32-bit AXI-Stream interface. Users can request this interface in a clock domain of their choice and are responsible for implementing the framer/de-framer for control packets. When the AXIS-Ctrl port is instantiated, the NoC Shell will expose the following signals for the user-logic to use:

- **`axis_ctrl_clk`**
  This is the clock that all the control port signals are synchronous to. The user may choose which clock source drives this clock. This is an output of the NoC Shell.

- **axis_ctrl_rst**
  This is the synchronous reset for the AXIS-Ctrl logic. This is an output of the NoC Shell. This reset will be asserted for at least one clock cycle after which the client logic will have 100 us to complete the following tasks:
    - Abort all pending transactions. Pending transactions may not be acknowledged
    - Reset all software configuration block state to the initial powerup/startup values
- **m_axis_ctrl_<signal>**
  This is the master AXI-Stream port from which the user logic will receive all requests for incoming transactions and responses for outgoing ones. This is an output of the NoC Shell. <signal> refers to the following standard AXI4-Stream signals: tdata (32 bits), tvalid, tready and tlast. Each AXI-Stream packet will contain the contents of Table 7, that the user logic will have to interpret manually.
- **s_axis_ctrl_<signal>**
  This is the slave AXI-Stream port where the user logic will send all requests for outgoing transactions and responses for incoming ones. This is an input to the NoC Shell. <signal> refers to the following standard AXI4-Stream signals: tdata (32 bits), tvalid, tready and tlast. Each AXI-Stream packet will contain the contents of Table 7, that the user logic will have to interpret manually.

2.3.2.2 Control Port (Simple Interface)

The control port provides a simpler interface to generate and consume control transactions. This interface supports blocking reads/writes, timed commands, backpressure and (N)ACKs, and allows the users to not worry about parsing the AXIS-Ctrl packet. The NoC Shell will internally de-frame AXIS-Ctrl packets, post a transaction on the slave bus and then frame the response back to AXIS-Ctrl. The simplicity of the interface does yield the following limitations:

- Only the read, write and sleep (trivially) opcodes are supported
- Block reads and writes will be split into multiple single reads and writes respectively
- The priority bit is not supported

When the control port is instantiated, NoC Shell will expose the following ports for the user-logic to use:

- **ctrlport_clk**
  This is the clock that all the control port signals are synchronous to. The user may choose which clock source drives this clock. This is an output of the NoC Shell.
- **ctrlport_rst**
  This is the synchronous reset for the control port logic. This reset will be asserted for at least one clock cycle and the client logic will have 100 us to complete the following tasks:
    - Abort all pending transactions. Pending transactions may not be acknowledged
    - Reset all software configuration block state to the initial powerup/startup values
- **m_ctrlport_<signal>**
  This is the master control port from which the user logic will receive all transaction requests and to which the user logic will send responses. A slave port is always instantiated. Table 18 shows the various signals represented by <signal>.
- **s_ctrlport_<signal>**
  This is the slave control port to which the user logic will send all transaction requests and

from which it will receive responses. The slave port is optional. Table 18 shows the various signals represented by <signal>

Signal	Direction (Master)	Width	Purpose	Usage
req_wr	out	1	A single-cycle strobe that indicates the start of a write transaction.	Required
req_rd	out	1	A single-cycle strobe that indicates the start of a read transaction.	Required
req_addr	out	20	Address for transaction. *This field is valid only when req_rd or req_wr is high.*	Required
req_portid	out	10	Port ID within the device to send the transaction to. This is the local port number. *This field is valid only when req_rd or req_wr is high.*	Required (Master Only)
req_rem_epid	out	16	Endpoint ID of the stream endpoint to send the transaction to. *This field is valid only when req_rd or req_wr is high.*	Required (Remote Master Only)
req_rem_portid	out	10	Port ID within the stream endpoint to send the transaction to. *This field is valid only when req_rd or req_wr is high.*	Required (Remote Master Only)
req_data	out	32	Data for write transaction. *This field is valid only when req_wr is high.*	Required
req_byte_en	out	4	A bitmask indicating which of the 4 bytes to use for transaction. If bit ‘i’ is high in keep then byte ‘i’ will be used from req_data. (If not present, use all 32 bits) *This field is valid only when req_rd or req_wr is high.*	Optional
req_has_time	out	1	Does the transaction need to happen at a given time? (If not present, perform transaction ASAP) *This field is valid only when req_rd or req_wr is high.*	Optional

req_time	out	64	Timestamp to execute the transaction at. (If not present, perform transaction ASAP) *This field is valid only when req_rd or req_wr is high.*	Optional
resp_ack	in	1	A strobe that indicates transaction completion	Required
resp_status	in	2	The status associated with the transaction ack. The interpretation of these bits is defined in Table 8. (If not present, the value is 0 i.e. OKAY) *This field is valid only when resp_ack is high.*	Optional
resp_data	in	32	Response data for a read transaction. *This field is valid only when resp_ack is high.*	Required

Table 18: Control Port signal definitions

READ and WRITE Transaction

A write transaction is defined as the assertion of reg_wr for 1 clock cycle and a read transaction is defined as a similar assertion of reg_rd. The value of reg_addr and reg_data (and other optional signals) can be used as arguments for the write. An untimed write will start executing in the same cycle as the assertion of reg_wr. The example in Figure 4 shows two writes (A0, A1) that execute in 0 clock cycles and one write that takes multiple cycles to execute. Figure 5 shows two 0 cycle reads and one multi-cycle read.

Control-Port Transaction Rules

- After the transaction completes, the client must assert resp_ack (along with other optional response signals) to indicate transaction completion. For a read, the resp_data is used for the readback data. resp_ack must be asserted at least 1 clock cycle after the assertion of the req_wr or req_rd signal.
- It is permissible for a read or write to take multiple clocks cycles. Regardless of the execution time, the ack must be asserted 1 clock cycle after completion.
- There is no upper limit on the execution time of a transaction; this allows blocking transactions that wait on hardware, but it also requires flow control on the sender's part to guarantee that transactions don't clog upstream routers.
- After a response ACK, the ctrlport slave must be ready to receive the next transaction in the next clock cycle.
- If reg_wr and reg_rd are asserted in the same clock cycle, then the read must be executed before the write.

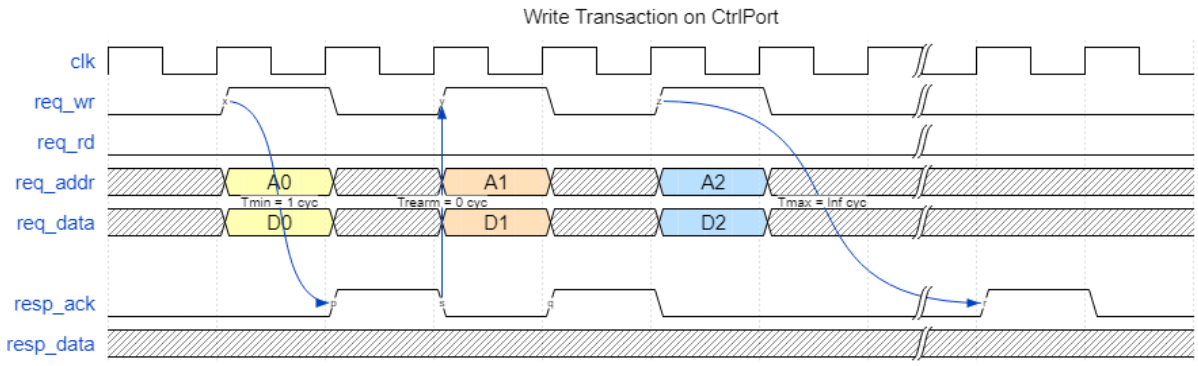

Figure 4: ctrlport write transaction

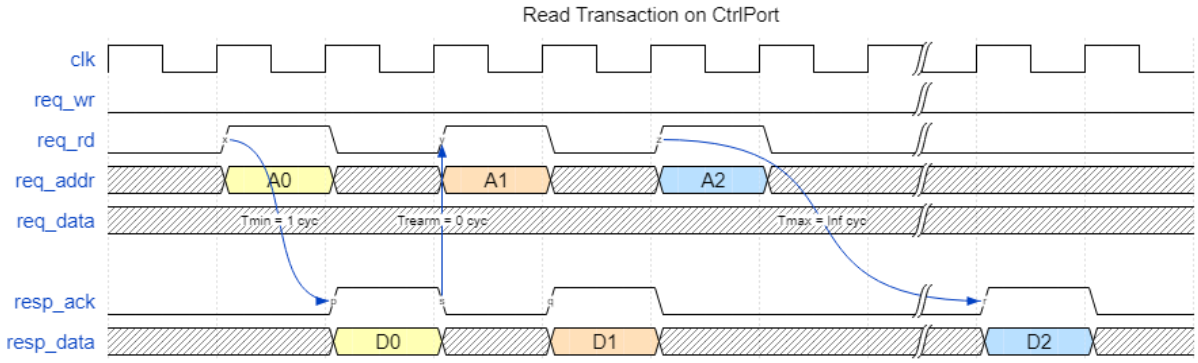

Figure 5: ctrlport read transaction

Transaction Status

It is possible for a control slave to acknowledge a transaction with an optional status. The status bits must have the appropriate value when resp_ack is high. Figure 6 shows two transaction where the first one was successful and the second one failed.

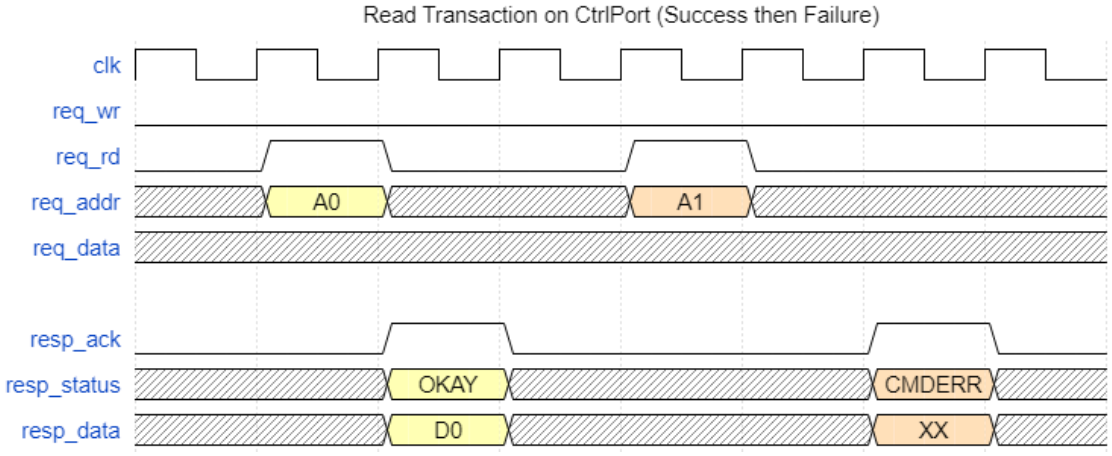

Figure 6: Read completion status (Success and Failure)

Timed Transactions

A transaction (read or write) can also be timed i.e. the execution of the transaction will begin at the specified time. The optional signals req_has_time will be asserted to indicate that a

transaction is timed. The contents of req_time will be used as the timestamp at which transaction execution should start. It is permissible for the transaction to take multiple clock cycles to finish executing, after which the resp_ack must be asserted. Figure 7 shows three timed transactions: The first one executes immediately (because time = req_time) and executes in 1 clock cycle. The second one must wait for the time to tick up to 2000 at which point it executes (in 1 clock cycle) and asserts an ack. The third one is late and responds with a Timestamp error (TSERR).

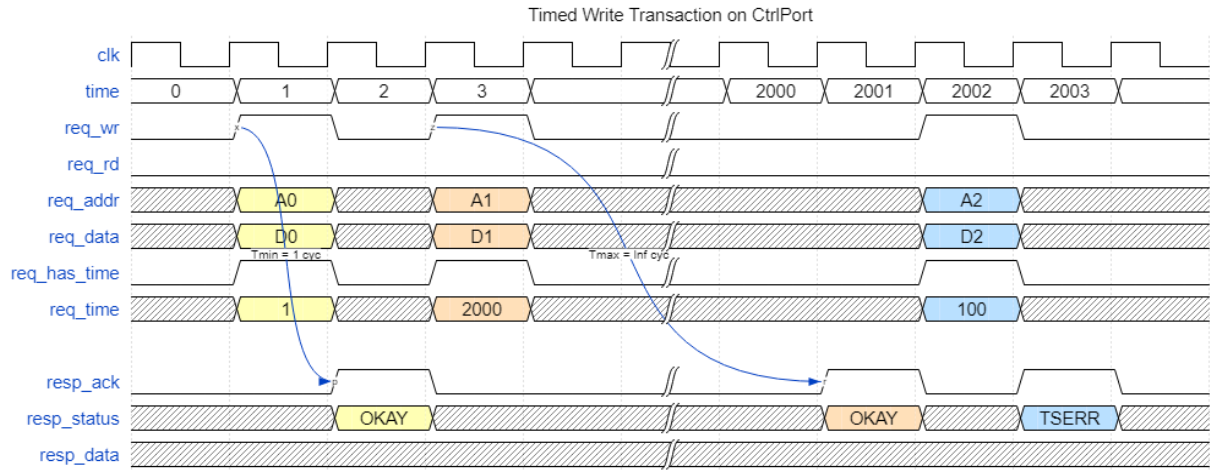

Figure 7: Timed write transactions

2.3.2.3 NoC Shell Generation Options

RFNoC ModTool has the following options to generate the control interface for the NoC Shell of a NoC block.

- Control Interface (`fpga_iface`)
  - Definition: Which HDL interface to expose
  - Options: “AXIS-Ctrl” (`axis_ctrl`) or “Control Port” (`ctrlport`)
  - Constraints: None
- Interface Direction (`interface_direction`)
  - Definition: Direction of the interface
  - Options: “Slave Only” (`slave`), “Master and Slave” (`master_slave`), or “Remote-Master and Slave” (`remote_master_slave`)
  - Constraints: “Slave Only” not allowed for “AXIS-Ctrl” interface
- Buffer Depth (`fifo_depth`)
  - Definition: Depth of the input AXI-Stream Control FIFO in words
  - Options: 32 – 4096 (in powers of 2)
  - Constraints: None
- Clock Domain (`clk_domain`)
  - Definition: Clock domain to export the interface in
  - Options: All available RFNoC and User clocks
  - Constraints: None
- Control Port Settings
  - Byte Mode (`byte_mode`)
    - Definition: Expose the “req_byte_en” field on the interface.
    - Options: “On” (`True`) or “Off” (`False`) (Off implies 32-bit mode)

        - Constraints: None
    - Timed Commands (`timed`)
        - Definition: Expose the “req_has_time” and “req_time” fields on the interface.
        - Options: “On” (`True`) or “Off” (`False`) (Off implies immediate or non-timed commands)
        - Constraints: None
    - Transaction Status (`has_status`)
        - Definition: Expose the “resp_status” field on the interface.
        - Options: “On” (`True`) or “Off” (`False`) (Off implies transactions that are always successful)
        - Constraints: None

2.3.3 Data-Plane

The data plane in the FPGA can be exposed using a low-level AXI4-Stream interface called *AXI-Stream CHDR* (AXIS-CHDR) or using the simpler abstracted interfaces *AXI-Stream Payload Context* and *AXI-Stream Data*. This plane of communication is intended for high-throughput data transfer between blocks. The CHDR header information is retained in the data plane so blocks can attach additional information like metadata and timestamps to packets. The CHDR header information (Table 2) must be accurate for all packets entering and leaving a block except for the destination endpoint ID (`DstEPID`). The destination endpoint is used for routing between stream endpoints and is not relevant between adjacent blocks; the value of this field is reserved and will be overwritten by the framework.

2.3.3.1 AXI-Stream CHDR (Low-level Interface)

The AXI-Stream CHDR (AXIS-CHDR) interface provides direct access to the data ports. The client can request this interface for maximum control over the stream, but the client is responsible for implementing the framer/de-framer for CHDR packets.

A block may have between 0 and 64 input/output data ports. For a block with P input ports, the NoC Shell will contain P separate slave CHDR streams. For a block with Q output ports, the NoC Shell will contain Q separate master CHDR streams. All the streams share the same clock and reset.

When the AXI-Stream CHDR interface is used, the NoC Shell will expose the ports listed below for the user-logic to connect to. In this list, `<name>` refers to the name provided by the user for this port and `<signal>` refers to one of the standard AXI4-Stream signals: tdata (CHDR width), tvalid, tready and tlast. Additionally, these signals may be a concatenation of multiple data streams if a parameter is used to define the number of ports. For example, the signal `s_myports_chdr_tvalid[1]` would refer to tvalid of the slave stream for port 1 of “myports”.

- **`axis_chdr_clk`**
  This is the clock that all the `axis_chdr` signals are synchronous to. The user may choose which clock source drives this clock. This is an output of the NoC Shell.
- **`axis_chdr_rst`**
  This is the synchronous reset for the data-path logic. This is an output of the NoC Shell. This reset will be asserted for at least one clock cycle after which the client logic will have 1 ms to complete the following tasks:

  - Reset the data-path state to the initial powerup/startup values
  - Stop generating data on the master interface
  - Drop all data on the slave interface (Note that the slave interface may have partial CHDR packets that need to be dropped)
- **`s_<name>_chdr_<signal>`**
  This is the slave interface to which the user logic will send all outgoing items/samples. Each AXI-Stream packet must be of the format described in Table 1 and must be a CHDR Data Packet (PktType = 6 or 7).
- **`m_<name>_chdr_<signal>`**
  This is the master interface from which the user logic will receive incoming items/samples. Each AXI-Stream packet will be in the format described in Table 1 and will be a CHDR Data Packet (PktType = 6 or 7). The user logic will be required to parse this packet format.

2.3.3.2 AXI-Stream Payload Context (Simple Interface)

The payload context interface provides a simpler interface to connect processing IP. The payload context interface abstracts away the CHDR stream into two separate AXI-Stream interfaces: Payload and context. The payload stream contains the payload data of a CHDR packet and can be directly connected to processing blocks that support AXI-Stream. The payload stream is comprised of *items* (the smallest processing unit; e.g., a data sample) and can deliver one or more items per cycle. The context stream contains additional information about the payload stream such as the header, timestamp and metadata. Splitting the payload and context streams allows separate (but coupled) state machines for data and header processing. The following abbreviations are used below:

- CHDR_W: The bit-width of the CHDR bus that the block can support.
- ITEM_W: The bit-width of a raw data item. ITEM_W must be a multiple of 8 (AXI-Stream requires transfers to be in bytes).
- NIPC: The number of items delivered per cycle between the interface and the processing IP.

A block may have 0 to 64 input/output data ports. For a block with P input ports, the NoC shell will contain P separate master CHDR streams. For a block with Q output ports, the NoC shell will contain Q separate slave CHDR streams. All the streams share the same clock and reset.

When the AXI-Stream Payload Context interface is used, the NoC Shell will expose the ports listed below for the user-logic to connect to. In this list, `<name>` refers to the name provided by the user for this port and <signal> refers to one of the standard AXI4-Stream signals: tdata (CHDR width), tvalid, tready and tlast. Additionally, these signals may be a concatenation of multiple data streams if a parameter is used to define the number of ports. For example, the signal `s_myports_payload_tvalid[1]` would refer to tvalid of the slave payload stream for port 1 of “myports”.

- **`axis_data_clk`**
  This is the clock that all the AXI-Stream signals are synchronous to. The user may choose which clock source drives this clock. This is an output of the NoC Shell.
- **`axis_data_rst`**
  This is the synchronous reset for the data-path logic. This is an output of the NoC Shell.

This reset will be asserted for at least one clock cycle after which the client logic will have 1 ms to complete the following tasks:
  - Reset the data-path state to the initial powerup/startup values
  - Stop generating data on the master interface
  - Drop all data on the slave interface (Note that the slave interface may have partial CHDR packets that need to be dropped)
- **`s_<name>_payload_<signal>`**, **`s_<name>_context_<signal>`**
  These are the slave interfaces to which the user logic will send outgoing items. Table 19 shows the various signals represented by `<signal>`.
- **`m_<name>_payload_<signal>`**, **`m_<name>_context_<signal>`**
  These are the master interfaces from which the user logic will receive incoming items. Table 19 shows the various signals represented by `<signal>`.

<table>
<tr><th>Signal</th><th>Direction (Master)</th><th>Width</th><th>Purpose</th><th>Usage</th></tr>
<tr><td>payload_tdata</td><td>out</td><td>NIPC * ITEM_W</td><td>The primary data payload word for this transfer</td><td>Required</td></tr>
<tr><td>payload_tkeep</td><td>out</td><td>NIPC</td><td>An item qualifier that indicates whether the content of the associated item in tdata is processed in the stream.
NOTE: The granularity of this field is item and not byte. This behavior is different from the standard AXI4-Stream tkeep.
NOTE: This may only used to indicate trailing items at the end of a packet.</td><td>Required for NIPC > 1</td></tr>
<tr><td>payload_tlast</td><td>out</td><td>1</td><td>Indicates the last word (transfer) in the current payload packet</td><td>Required</td></tr>
<tr><td>payload_tvalid</td><td>out</td><td>1</td><td>Indicates that the master is driving a valid packet payload word (transfer)</td><td>Required</td></tr>
<tr><td>payload_tready</td><td>in</td><td>1</td><td>Indicates that the slave can accept a payload word (transfer) in the current cycle</td><td>Required</td></tr>
<tr><td>context_tdata</td><td>out</td><td>CHDR_W</td><td>The primary context word for this transfer</td><td>Required</td></tr>
<tr><td>context_tuser</td><td>out</td><td>4</td><td>Indicates the type of context word<table><tr><th>Value</th><th>Type</th></tr><tr><td>0x0</td><td>CHDR Header (HDR)</td></tr><tr><td>0x1</td><td>CHDR Header + Timestamp (HDR_TS)</td></tr><tr><td>0x2</td><td>Timestamp Only (TS)</td></tr><tr><td>0x3</td><td>Metadata (MDATA)</td></tr><tr><td>Rest</td><td>Reserved</td></tr></table></td><td>Required</td></tr>
<tr><td>context_tlast</td><td>out</td><td>1</td><td>Indicates the last word (transfer) in the current context packet.</td><td>Required</td></tr>
<tr><td>context_tvalid</td><td>out</td><td>1</td><td>Indicates that the master is driving a valid context word (transfer)</td><td>Required</td></tr>
<tr><td>context_tready</td><td>in</td><td>1</td><td>Indicates that the slave can accept a context word (transfer) in the current cycle</td><td>Required</td></tr>
</table>

Table 19: AXI-Stream Payload Context port signal definitions

NOTE: The data in a context packet represents CHDR header information and thus must be in the same order as the CHDR field. The following sequences on context_tuser are valid, and all others will be regarded as a protocol violation for the context port.

- Packet with no timestamp and no metadata (all CHDR Widths)

HDR

- Packet with timestamp and no metadata (CHDR Width = 64)

HDR	TS

- Packet with timestamp and no metadata (CHDR Width > 64)

HDR_TS

- Packet with timestamp and metadata (CHDR Width = 64)

HDR	TS	MDATA	…	MDATA

- Packet with timestamp and metadata (CHDR Width > 64)

HDR_TS	MDATA	…	MDATA

- Packet with no timestamp and metadata (all CHDR Widths)

HDR	MDATA	…	MDATA

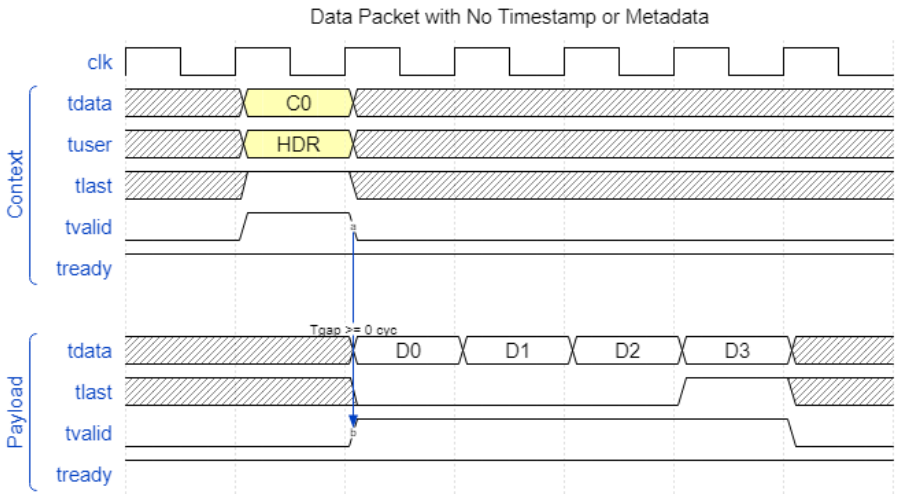

Figure 8: A 4-word packet with only the header on AXIS Payload Context port

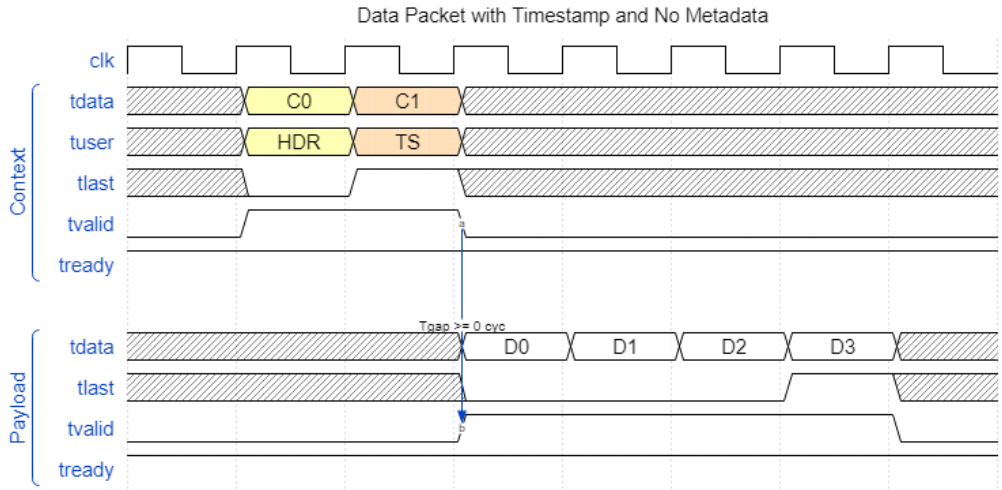

Figure 9: A 4-word packet with a header and timestamp on the AXIS Payload Context port (CHDR_W = 64)

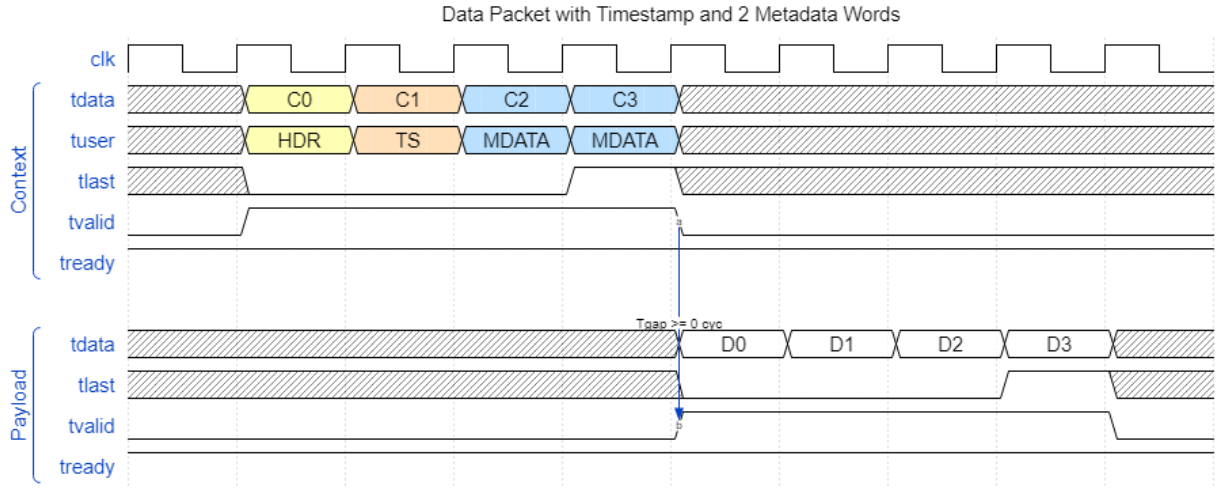

Figure 10: A 4-word packet with a header, timestamp and 2 metadata words on the AXIS Payload Context port (CHDR_W = 64)

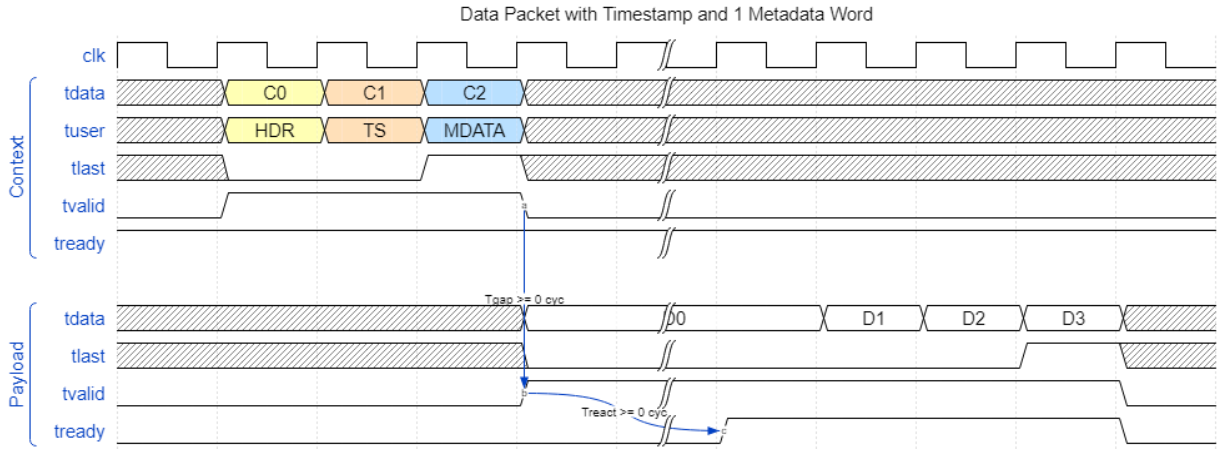

Figure 11: A 4-word packet on the AXIS Payload Context port with a gap between the context and payload (CHDR_W = 64)

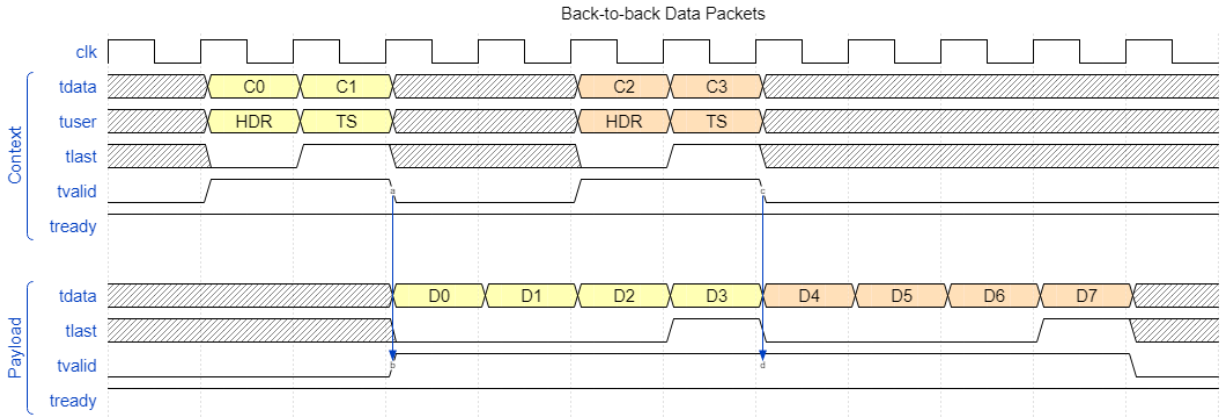

Figure 12: Two back-to-back packets on the AXIS Payload Context port (with header prefetching; CHDR_W = 64)

2.3.3.3 AXI-Stream Data (Simple Interface)

The AXI-Stream Data interface provides another simple user interface. It uses an AXI-Stream data interface but does not require the user to packetize header information. It also supports timestamps, EOB, and EOV. The following abbreviations are used below:

- CHDR_W: The bit-width of the CHDR bus that the block can support.
- ITEM_W: The bit-width of a raw qq1qqqdata item. ITEM_W must be a multiple of 8 (AXI-Stream requires transfers to be in bytes).
- NIPC: The number of items delivered per cycle between the interface and the processing IP.

A block may have 0 to 64 input/output data ports. For a block with P input ports, the NoC shell will contain P separate master CHDR streams. For a block with Q output ports, the NoC shell will contain Q separate slave CHDR streams. All the streams share the same clock and reset.

When the AXI-Stream Data interface is used, the NoC Shell will expose the ports listed below for the user-logic to connect to. In this list, `<name>` refers to the name provided by the user for

this port and <signal> refers to one of the standard AXI4-Stream signals: tdata (CHDR width), tvalid, tready and tlast. Additionally, these signals may be a concatenation of multiple data streams if a parameter is used to define the number of ports. For example, the signal s_myports_axis_tvalid[1] would refer to tvalid of the slave stream for port 1 of "myports".

- **axis_data_clk**
  This is the clock that all the AXI-Stream signals are synchronous to. The user may choose which clock source drives this clock. This is an output of the NoC Shell.
- **axis_data_rst**
  This is the synchronous reset for the data-path logic. This reset will be asserted for at least one clock cycle and the client logic will have 1 ms to complete the following tasks:
  - Reset the data-path state to the initial powerup/startup values
  - Stop generating data on the master interface
  - Drop all data on the slave interface (Note that the slave interface may have partial CHDR packets that need to be dropped)
- **s_<name>_axis_<signal>**
  This is the slave interface to which the user logic will send outgoing items. Table 20 shows the various signals represented by <signal>.
- **m_<name>_axis_<signal>**
  This is the master interface from which the user logic will receive incoming items. Table 20 shows the various signals represented by <signal>.

The signals tlength, ttimestamp, thas_time, teov, and teob are sideband signals and behave like tuser in traditional AXI4-Stream. Rather than having a single tuser signal, these signals have been separated into individual signals for ease of use. On the NoC Shell's master data interface, these signals are valid for the duration of the packet (i.e., whenever tvalid is true).

When the sideband signals are read by the NoC Shell's slave data interface depends on the SIDEBAND_AT_END parameter. If SIDEBAND_AT_END is True then these signals must be valid on the last transfer of each packet (i.e., when tlast is asserted) and tlength is calculated automatically by the NoC Shell. If SIDEBAND_AT_END is False, then these signals must be valid on the first transfer of each packet and tlength must be provided as an input to indicate the length of the packet.

The SIDEBAND_AT_END = True setting is required in situations where one or more items associated with the CHDR header (e.g., length, timestamp, EOB, EOV) are not known until the end of the packet is ready to be output. An important side-effect of this setting is that all output packets sent to the NoC Shell's slave interface will be completely buffered before they are sent out. This adds latency to the packets and requires that the NoC Shell implement an MTU-sized buffer to store outgoing packets.

Signal	Direction (Master)	Width	Purpose	Usage
tdata	out	NIPC * ITEM_W	The data payload word for this transfer	Required
tkeep	out	NIPC	An item qualifier that indicates whether the content of the associated item in tdata is processed in the stream. *NOTE: The granularity of this field is item and not byte. This behavior is different from the standard AXI4-Stream tkeep.* *NOTE: This may only be used to indicate trailing items at the end of a packet.*	Required for NIPC > 1
tlast	out	1	Indicates the last word (transfer) in the current payload packet.	Required
tvalid	out	1	Indicates that the master is driving a valid packet payload word (transfer)	Required
tready	in	1	Indicates that the slave can accept a payload word (transfer) in the current cycle	Required
ttimestamp	out	64	The timestamp for the first item in the packet	Optional
thas_time	out	1	Indicates if the ttimstamp field is being used. This will be 0 if there is not timestamp for the current packet.	Optional
tlength	out	16	The byte length of the data packet. This signal is only used by the NoC Shell's master interface and is not required by the slave interface. *NOTE: This port is used by the slave interface only when* `SIDEBAND_AT_END` *is True and is ignored by the slave interface when* `SIDEBAND_AT_END` *is False.*	Optional
teov	out	1	Indicates if the EOV bit was set in the packet	Optional
teob	out	1	Indicates if the EOB bit was set in the packet	Optional

Table 20: AXI-Stream Data port signal definitions

2.3.3.4 NoC Shell Generation Options

RFNoC Modtool has the following options to generate the data interface for the NoC Shell of a NoC block.

- Data Interface (`fpga_iface`)
  - Definition: Which HDL interface to expose
  - Options: “AXI-Stream CHDR” (axis_chdr), “AXI-Stream Payload Context” (axis_`pyld_ctxt`), or “AXI-Stream Data” (`axis_data`)
  - Constraints: None
- Number of Input Ports
  - Definition: The number of input ports
  - Options: 0 - 64
  - Constraints: None
- Number of Output Ports
  - Definition: The number of output ports
  - Options: 0 - 64
  - Constraints: None
- Port Specific Settings (for each input and output port)
  - Clock Domain (`clk_domain`)
    - Definition: The clock domain for the payload or data interface
    - Options: All available RFNoC and User clocks
    - Constraints: None
  - Item Width (`item_width`)
    - Definition: Bit width of each data item
    - Options: 32, 64, 128, etc.
    - Constraints: Only valid when using the AXI-Stream Payload Context or AXI-Stream Data interfaces
  - Number of Items per Cycle (`nipc`)
    - Definition: Number of data items to deliver per clock cycle
    - Options: 1-256 (in powers of 2)
    - Constraints: Only valid when using the AXI-Stream Payload Context or AXI-Stream Data interfaces
  - Payload FIFO Depth (`payload_fifo_depth`)
    - Definition: Depth of the AXI-Stream buffer for the payload data path
    - Options: 1 or larger (in powers of 2)
    - Constraints: Only valid when using the AXI-Stream Payload Context or AXI-Stream Data interfaces
  - Context FIFO Depth (`context_fifo_depth`)
    - Definition: Depth of the AXI-Stream buffer for the context data path
    - Options: 1 or larger (in powers of 2)
    - Constraints: Only valid when using the AXI-Stream Payload Context interface
  - Info FIFO Depth (`info_fifo_depth`)
    - Definition: Depth of the AXI-Stream buffer for queued packet information
    - Options: 1 or larger (in powers of 2)
    - Constraints: Only valid when using the AXI-Stream Data interface
  - Context Prefetching

- Definition: Allow prefetching context data for the next packet when the current packet is in flight.
- Options: “On” or “Off”
- Constraints: Only valid when using the AXI-Stream Payload Context interface

2.3.4 IO Ports (Advanced)

IO Ports are interfaces to the user-logic that don’t interact with the RFNoC framework. IO Ports may interact with other blocks in an assembled design (for backdoor inter-block communication) or with IO on the USRP device.

2.3.4.1 Hardware Timestamp Interface

The user logic can get access to a hardware time-base and timestamp. The capabilities of a hardware time-base are device specific. The timestamp can be used with real-time blocks like the radio which interfaces with ADCs/DACs.

- **`tb_clk`**: The time-base clock.
- **`tb_rst`**: A synchronous reset in tb_clk domain. tb_rst = 1 indicates that the time-base is disabled.
- **`tb_timestamp`**: A 64-bit global timestamp that is synchronous to *tb_clk*. The timestamp is a counter that may start at an arbitrary value and count up by one every clock cycle of tb_clk after tb_rst is released.
- **`tb_period_ns_q32`**: A 64-bit fixed point number in the Q32 format that represents the period of the time-base in nanoseconds.

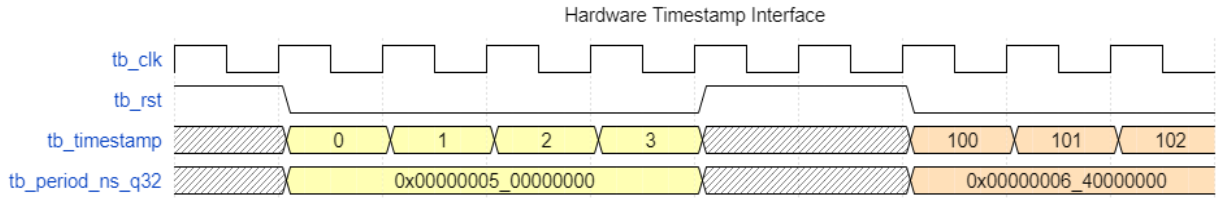

Figure 13: An example of a time-base reconfiguration from 200 MHz to 160 MHz

2.3.4.2 Generic IO Ports

It is possible to add more generic IO to a NoC block. A generic IO port is a collection of signals, their types, widths and directions. This collection is called an IO Signature. An IO Signature can be inherited from a specific USRP device or be user-defined. Each IO Signature contains the following information:

- *Name*: A unique name for this IO Signature
- *Drive*: The drive direction of this IO Port. The driver direction can be “Slave” (driven by a master), “Master” (driven a single slave), “Listener” (a special slave with only inputs) or “Broadcaster” (a special master with only outputs).
- *Port List*: A list of signals each with the following properties:
  - *Name*: Name of the signal
  - *Type*: Is this a “Clock”, “Reset” or “Generic” signal?

- *Direction*: Is this an input or an output on the master?
- *Width*: The bit-width of the signal

If a block defines a generic IO Port, then the IO port must be assigned to another IO Port with the same signature during design image assembly. The other IO Port may be a part of the USRP device or an IO Port on another block. The following connection rules apply:

- A Master can drive exactly one Slave
- A Slave can be driven by exactly one Master
- A Broadcaster can drive zero or more listeners
- A Listener must be driven by at least one Broadcaster
- A Master cannot drive a Listener
- A Broadcaster cannot drive a Slave

2.3.5 Backend RFNoC Interface

Because NoC Shell is a part of the user NoC block, there will be certain interfaces exposed as inputs/outputs from the block that the user logic can ignore. These interfaces are termed as "backend" and are used by NoC Shell to communicate with the rest of the framework.

2.3.5.1 CHDR

- **`rfnoc_chdr_clk`**
  This is the clock for the rfnoc_chdr port (described below).
- **`rfnoc_chdr_rst`**
  This is the synchronous reset for the rfnoc_chdr port. This reset is driven by the backend interface.
- **`s_rfnoc_chdr_<signal>`**
  The slave rfnoc_chdr port. This interface accepts CHDR packets from the framework. <signal> refers to the following standard AXI4-Stream signals: tdata (CHDR_W bits), tvalid, tready and tlast. The widths of these signals depend on the number of input ports and the CHDR_W setting.
- **`m_rfnoc_chdr_<signal>`**
  The master rfnoc_chdr port. This interface outputs CHDR packets to the framework. <signal> refers to the following standard AXI4-Stream signals: tdata (CHDR_W bits), tvalid, tready and tlast. The widths of these signals depend on the number of output ports and the CHDR_W setting.

2.3.5.2 Control

- **`rfnoc_ctrl_clk`**
  This is the clock for the rfnoc_ctrl port (described below).
- **`rfnoc_ctrl_rst`**
  This is the synchronous reset for the rfnoc_ctrl port. This reset is driven by the backend interface.
- **`s_rfnoc_ctrl_<signal>`**
  The slave rfnoc_ctrl port. This interface accepts AXIS-Ctrl packets from the framework. <signal> refers to the following standard AXI4-Stream signals: tdata (32 bits), tvalid, tready and tlast.

- **`m_rfnoc_ctrl_<signal>`**
  The master rfnoc_ctrl port. This interface outputs AXIS-Ctrl packets to the framework. <signal> refers to the following standard AXI4-Stream signals: tdata (32 bits), tvalid, tready and tlast.

2.3.5.3 Configuration and Status

- **`rfnoc_core_config`**
  A 512-bit interface for the framework to configure the state of the NoC shell logic. The interpretation of the bits in this bus is determined by the framework. Client logic is not expected to use this signal.
- **`rfnoc_core_status`**
  A 512-bit interface for the framework to read the state of the NoC shell logic. The interpretation of the bits in this bus is determined by the framework. Client logic is not expected to use this signal.

2.4 RFNoC FPGA Image

The RFNoC FPGA image is a standalone design for a USRP that has a collection of block instantiations and a partial topology preconfigured in the FPGA (static connections). This design can be configured using software to create a full flow-graph or be a part of a multi-USRP flow-graph.

2.4.1 Workflow

After all blocks are developed, the RFNoC framework has built-in tools to generate an FPGA image with user-specified block instantiations and a central router core with user-specified block connections. The following user information will be used to define a topology of the blocks and build an FPGA bitfile.

- USRP Device Info
- CHDR Width
- Number of Stream endpoints
  - Number of Input and Output ports for each stream endpoint
  - Optional: Buffer size for each endpoint
- NoC blocks to Build in the FPGA Image
  - Optional: Clock choices for each block
  - Optional: IO Port connections for each block
- Static connections between blocks, stream endpoints and transport adapters

2.4.2 Design Assembly Toolflow

The following information is inferred based on the user-specified preferences, device info (part of the board support package) and meta-data from each block:

- CHDR Width (W_{chdr})
- Stream endpoints (M)
  - Buffer size for each endpoint (B_{ep})
- NoC blocks (N_{user})

    - Number of input and output ports
    - Datapath connection topology
    - Clock choices for the data and control clock
- Number of transport adapters (P)
- Number of IO-based NoC blocks (Radio, DDR-based blocks, etc.) (N_{fixed})

The code generator will determine the following parameters using that info:

- CHDR Crossbar
    - Number of ports = P + M
    - Data Width = W_{chdr}
- Stream Endpoints
    - Number of endpoints = M
    - Data Width = W_{chdr}
    - Buffer Size = B_{ep}
- Control Crossbar
    - Number of Ports = (M + N_{fixed} + N_{user} + 1)
- Static Router
    - Number of Ports = (M + Ports(N_{fixed}) + Ports(N_{user}))
    - Inter-port connections
    - Adjacency list

Using this info, the code generator will do the following

1. For a given device, look the number of transport adapters (P) and read the number of requested stream endpoints (M), then instantiate a CHDR Crossbar with P+M ports
2. Instantiate M stream endpoints
3. Connect the P transport adapters to the first P ports of the chdr_crossbar and the next M ports to the stream endpoints
4. Read the number of NoC blocks (N) and instantiate a Control Crossbar with N_{fixed} + N_{user} + M + 1 ports
5. Connect the first port of the control crossbar to a core config endpoint, then connect the next M ports to the M control endpoints and then connect the remaining N ports to the NoC blocks' control interfaces.
6. Generate a static router with M + Ports(N_{fixed}) + Ports(N_{user}) ports and hook it up to the M stream endpoints and X NoC block ports (each block may have an arbitrary number of ports)
7. Read in the inter-block connections and build a table of the connections as an adjacency list that is readable by UHD.

2.4.3 Initialization and Usage

The RFNoC software will ensure that all the blocks in the FPGA image are initialized before use. RFNoC defines the following as three phases of initialization:

1. *All Blocks Idle*: Each block in the design (regardless of its previous state) must first be put in an idle streaming state, i.e. no data is streaming though the data-path.
2. *All Blocks Reset*: Each block in the design (regardless of its previous state) must then be put into a known state for settings and software configurable registers.
3. *Network Ready*: All blocks are initialized, and the core framework is ready to begin executing an application.

RFNoC is a network that may consist of multiple FPGA designs so “All Blocks” above refers to all blocks in the collection of USRPs controlled by the RFNoC software. Both, the RFNoC framework and the individual blocks share the responsibility for initialization. The control and data path resets (Section 2.3.2 and 2.3.3) will be asserted for each block to begin the reset operations and the framework imposes a time limit to allow the block to finish its reset procedure. Before the resets are asserted, the framework will also *flush* data at the input and output of each block. Flushing is an internal framework operation (not visible to the NoC blocks or the user) that ensures that no data is generated downstream of the flush point and all data is consumed at the flush point. Figure 14 shows the full initialization sequence for an image with multiple blocks (Block 0 … Block N) and multiple stream endpoints (SEP 0 … SEP N).

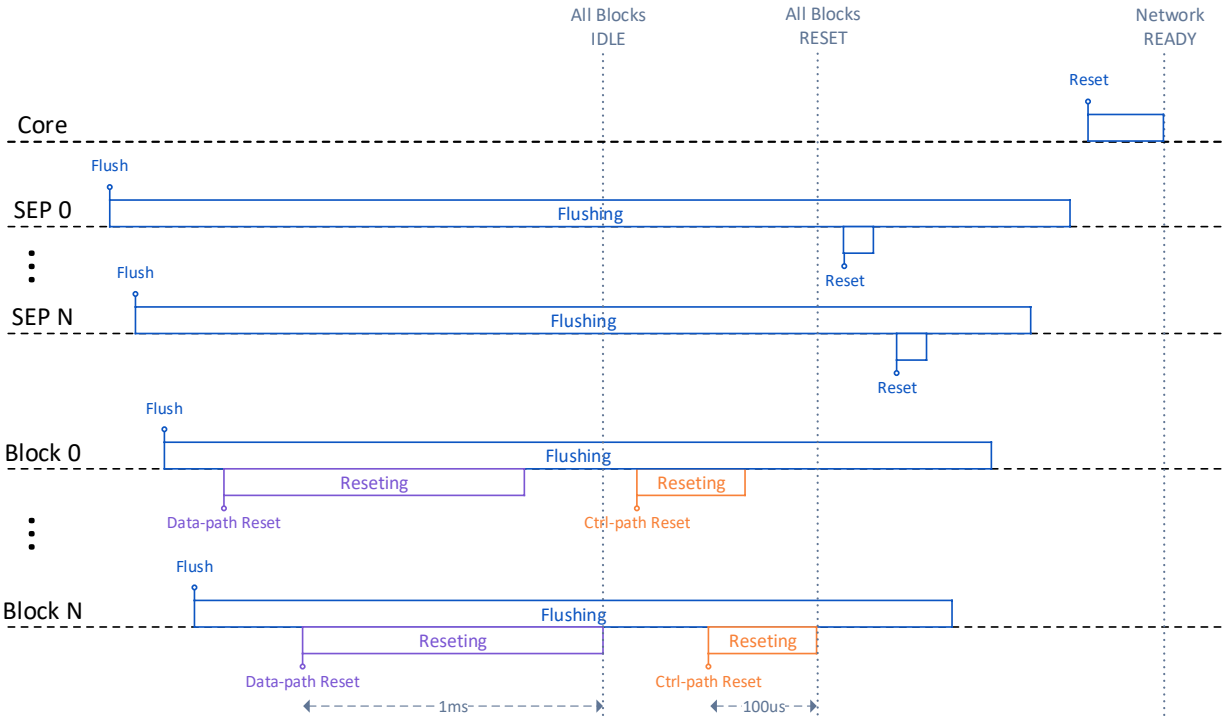

Figure 14: Initialization sequence for an RFNoC flow-graph

3 RFNoC Software Framework Overview

3.1 Basics

On the software side (UHD) RFNoC has the following interfaces:

- *Block Controller*: This is a control interface to each block in the design. Block controllers are discovered automatically by UHD based on the blocks in an FPGA, and they can be retrieved by the user to send and receive commands from blocks.
- *Graph*: The graph object manages a topology of blocks in an application. The static connections in the FPGA and the dynamic connections made by the user will be reflected in the graph. The graph will ensure state integrity between blocks if there are inherent dependencies.
- *Streamers*: Streamers are used to send/receive data to/from blocks in the FPGA.

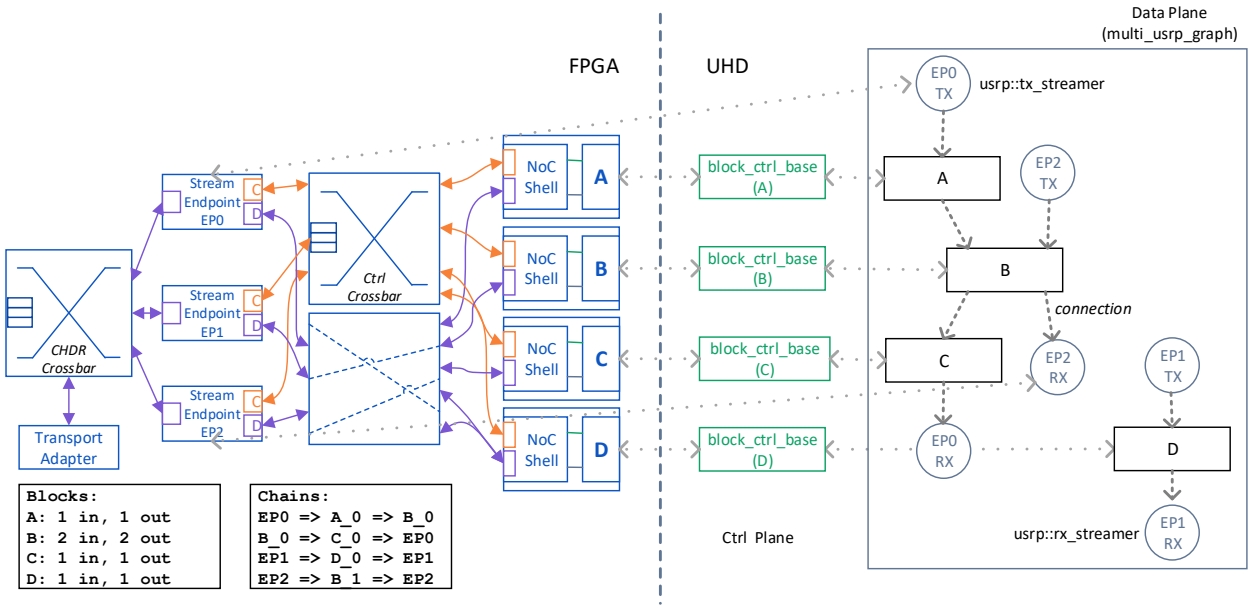

Figure 15: Example FPGA and SW objects in an RFNoC graph

3.2 Block Controller

The *block controller* is the UHD (C++) counterpart of the NoC Shell. It is a block-level API to interface with the clients of NoC shell. UHD has a default block controller object but it is possible to override it (via inheritance) with a custom controller C++ class. UHD maintains a list of block controllers for each block type and dynamically creates instances of it in software when the block is present in the FPGA image for the target USRP.

3.2.1 Block IDs

An identifier is needed to retrieve a block controller from UHD. Each block has two kinds of identification tags:

1. Each block in the FPGA must have a *NoC ID* which serves as a unique identifier of the function of the block. *NoC IDs* are 32 bits wide and can take on any value as long as it is

unique among blocks. For example, the FIR filter block will have a unique NoC ID that is different from the FFT block NoC ID. Multiple instances of the FIR block have the same NoC ID.

2. Each instance of a block in an FPGA has a *Block ID*. It is used to locate a specific block in an RFNoC network. For instance, if the same NoC block is instantiated twice, then the two instances will have the same *NoC ID* but different *Block ID*s. The syntax of a *Block ID* is as follows: `<Device>/<BlockName>:<BlockInstance>`

blockid_t Class Reference

Public Member Functions

1. **block_id_t** ()
2. **block_id_t** (const std::string &block_str)
3. **block_id_t** (const size_t device_no, const std::string &block_name, const size_t block_ctr=0)
4. std::string **to_string** () const

Return a string like this: "0/FFT#1" (includes all components, if set)

5. bool **match** (const std::string &block_str)

Check if block_str matches this block.

6. std::string **get** () const

*Short for **to_string()***

7. std::string **get_local** () const

*Like **get()**, but only returns the local part ("FFT#1")*

8. uhd::fs_path **get_tree_root** () const

Returns the property tree root for this block (e.g. "/mboards/0/xbar/FFT#1/")

9. size_t **get_device_no** () const

Return device number.

10. size_t **get_block_count** () const

Return block count.

11. std::string **get_block_name** () const

Return block name.

12. bool **set** (const std::string &new_name)

Set from string such as "0/FFT#1", "FFT#0", ...

13. bool **set** (const size_t device_no, const std::string &block_name, const size_t block_ctr=0)

*Sets from individual compontents, like calling **set_device_no()**, **set_block_name()***

14. void **set_device_no** (size_t device_no)

Set the device number.

15. bool **set_block_name** (const std::string &block_name)
Set the block name. Will return false if invalid block string.

16. void **set_block_count** (size_t count)
Set the block count.

Static Public Member Functions

17. static bool **is_valid_blockname** (const std::string &block_name)
Check if a given string is valid as a block name.

18. static bool **is_valid_block_id** (const std::string &block_id)
Check if a given string is valid as a block ID.

Detailed Description

Identifies an RFNoC block.

An RFNoC block ID is a string such as: 0/FFT#1

The rules for formatting such a string are:

DEVICE/BLOCKNAME#COUNTER

DEVICE: Identifies the device (usually the motherboard index) BLOCKNAME: A name given to this block COUNTER: If is are more than one block with a BLOCKNAME, this counts up.

So, 0/FFT#1 means we're addressing the second block called FFT on the first device.

This class can represent these block IDs.

Constructor & Destructor Documentation

block_id_t::block_id_t ()
block_id_t::block_id_t (const std::string & *block_str*)
block_id_t::block_id_t (const size_t *device_no*, const std::string & *block_name*, const size_t *block_ctr* = 0)

Parameters

device_no	Device number
block_name	Block name
block_ctr	Which block of this type is this on this device?

Member Function Documentation

std::string block_id_t::get () const

Short for **to_string()**

size_t block_id_t::get_block_count () const

Return block count.

std::string block_id_t::get_block_name () const

Return block name.

size_t block_id_t::get_device_no () const

Return device number.

std::string block_id_t::get_local () const

Like **get()**, but only returns the local part ("FFT#1")

uhd::fs_path block_id_t::get_tree_root () const

Returns the property tree root for this block (e.g. "/mboards/0/xbar/FFT#1/")

static bool block_id_t::is_valid_block_id (const std::string & *block_id*) [static]

Check if a given string is valid as a block ID.

static bool block_id_t::is_valid_blockname (const std::string & *block_name*) [static]

Check if a given string is valid as a block name.

bool block_id_t::match (const std::string & *block_str*)

Check if block_str matches this block.

bool block_id_t::set (const size_t *device_no*, const std::string & *block_name*, const size_t *block_ctr* = 0)

Sets from individual compontents, like calling **set_device_no()**, **set_block_name()**

bool block_id_t::set (const std::string & *new_name*)

Set from string such as "0/FFT#1", "FFT#0", ...

void block_id_t::set_block_count (size_t *count*)

Set the block count.

bool block_id_t::set_block_name (const std::string & *block_name*)

Set the block name. Will return false if invalid block string.

void block_id_t::set_device_no (size_t *device_no*)

Set the device number.

std::string block_id_t::to_string () const

Return a string like this: "0/FFT#1" (includes all components, if set)

3.2.2 Registers

Each block has a unique register space for low-level configuration. In the FPGA, this space is accessible using the AXIS-Ctrl port or the CtrlPort interface. Registers have a fixed bit width of

32 bits. Each block in software will provide an implementation for the rfnoc::register_iface interface that can be used to access registers in the FPGA. The rfnoc::register_iface interface supports the following:

- Peek/poke functionality for 32-bit registers
- Sleep functionality to time-sequence operations
- Timed commands
- Callbacks for asynchronous messages from the block
- Optional resilience parameters to trade throughput for robustness

register_iface Class Reference

Public Types

- using **sptr** = std::shared_ptr< **register_iface** >
- using **async_msg_validator_t** = std::function< bool(uint32_t addr, const std::vector< uint32_t > &data)>
- using **async_msg_callback_t** = std::function< void(uint32_t addr, const std::vector< uint32_t > &data, boost::optional< uint64_t >)>

Public Member Functions

- virtual void **poke32** (uint32_t addr, uint32_t data, uhd::time_spec_t time=uhd::time_spec_t::ASAP, bool ack=false)
- void **poke64** (uint32_t addr, uint64_t data, time_spec_t time=uhd::time_spec_t::ASAP, bool ack=false)
- virtual void **multi_poke32** (const std::vector< uint32_t > addrs, const std::vector< uint32_t > data, uhd::time_spec_t time=uhd::time_spec_t::ASAP, bool ack=false)
- virtual void **block_poke32** (uint32_t first_addr, const std::vector< uint32_t > data, uhd::time_spec_t time=uhd::time_spec_t::ASAP, bool ack=false)
- virtual uint32_t **peek32** (uint32_t addr, time_spec_t time=uhd::time_spec_t::ASAP)
- uint64_t **peek64** (uint32_t addr, time_spec_t time=uhd::time_spec_t::ASAP)
- virtual std::vector< uint32_t > **block_peek32** (uint32_t first_addr, size_t length, time_spec_t time=uhd::time_spec_t::ASAP)
- virtual void **poll32** (uint32_t addr, uint32_t data, uint32_t mask, time_spec_t timeout, time_spec_t time=uhd::time_spec_t::ASAP, bool ack=false)
- virtual void **sleep** (time_spec_t duration, bool ack=false)
- virtual void **register_async_msg_validator** (**async_msg_validator_t** callback_f)
- virtual void **register_async_msg_handler** (**async_msg_callback_t** callback_f)
- virtual void **set_policy** (const std::string &name, const uhd::device_addr_t &args)
- virtual uint16_t **get_src_epid** () const
- virtual uint16_t **get_port_num** () const

Detailed Description

A software interface to access low-level registers in a NoC block.

This interface supports the following:

19. Writing and reading registers
20. Hardware timed delays (for time sequencing operations)
21. Asynchronous messages (where a block requests a "register write" in software)

class has no public factory function or constructor.

Member Typedef Documentation

using register_iface::async_msg_callback_t = std::function<void(uint32_t addr, const std::vector<uint32_t>& data, boost::optional<uint64_t>)>

Callback function for acting upon an asynchronous message.

When a block in the FPGA sends an asynchronous message to the software, and it has been validated, the async message callback function is called. An async message can be modelled as a simple register write (key-value pair with addr/data) that is initiated by the FPGA.

When this message is called, the async message was previously verified by calling the async message validator callback.

using register_iface::async_msg_validator_t = std::function<bool(uint32_t addr, const std::vector<uint32_t>& data)>

Callback function for validating an asynchronous message.

When a block in the FPGA sends an asynchronous message to the software, the async message validator function is called. An async message can be modelled as a simple register write (key-value pair with addr/data) that is initiated by the FPGA. If this message returns true, the message is considered valid.

using register_iface::sptr = std::shared_ptr<register_iface>

Constructor & Destructor Documentation

virtual register_iface::~register_iface ()[virtual], [default]

Member Function Documentation

virtual std::vector<uint32_t> register_iface::block_peek32 (uint32_t *first_addr*, size_t *length*, time_spec_t *time* = uhd::time_spec_t::ASAP)

Read multiple 32-bit consecutive registers implemented in the NoC block.

Parameters

first_addr	The byte address of the first register to read from (truncated to 20 bits).
length	The number of 32-bit values to read
time	The time at which the transaction should be executed.

Returns

data New value of this register.

Example: If `first_addr` is set to 0, and length is 4, then this function will return a vector of length 4, with the content of registers at addresses 0, 4, 8, and 12, respectively.

Note: There is no guarantee that under the hood, the implementation won't separate the reads.

Exceptions

op_failed	if the transaction fails
op_timeout	if no response is received
op_seqerr	if a sequence error occurs

virtual void register_iface::block_poke32 (uint32_t *first_addr*, const std::vector< uint32_t > *data*, uhd::time_spec_t *time* = uhd::time_spec_t::ASAP, bool *ack* = false)

Write multiple consecutive 32-bit registers implemented in the NoC block.

This function will only allow writes to adjacent registers, in increasing order. If addr is set to 0, and the length of data is 4, then this method will trigger four writes, in order, to addresses 0, 4, 8, 12. For arbitrary addresses, cf. **multi_poke32()**.

Note: There is no guarantee that under the hood, the implementation won't separate the writes.

Parameters

first_addr	The byte addresses of the first register to write
data	New values of these registers
time	The time at which the first transaction should be executed.
ack	Should transaction completion be acknowledged?

Exceptions

op_failed	if an ACK is requested and the transaction fails
op_timeout	if an ACK is requested and no response is received
op_seqerr	if an ACK is requested and a sequence error occurs
op_timeerr	if an ACK is requested and a time error occurs (late command)

virtual uint16_t register_iface::get_port_num () const

Get the port number of the software counterpart of this register interface. This information is useful to send async messages to the host.

Returns

The 10-bit port number

virtual uint16_t register_iface::get_src_epid () const

Get the endpoint ID of the software counterpart of this register interface. This information is useful to send async messages to the host.

Returns

The 16-bit endpoint ID

virtual void register_iface::multi_poke32 (const std::vector< uint32_t > *addrs*, const std::vector< uint32_t > *data*, uhd::time_spec_t *time* = uhd::time_spec_t::ASAP, bool *ack* = false)

Write multiple 32-bit registers implemented in the NoC block.

This method should be called when multiple writes need to happen that are at non-consecutive addresses. For consecutive writes, cf. **block_poke32()**.

Parameters

addrs	The byte addresses of the registers to write to (each truncated to 20 bits).
data	New values of these registers. The lengths of data and addr must match.
time	The time at which the first transaction should be executed.
ack	Should transaction completion be acknowledged?

Exceptions

uhd::value_error	if lengths of data and addr don't match
op_failed	if an ACK is requested and the transaction fails
op_timeout	if an ACK is requested and no response is received
op_seqerr	if an ACK is requested and a sequence error occurs

op_timeerr	if an ACK is requested and a time error occurs (late command)

virtual uint32_t register_iface::peek32 (uint32_t *addr*, time_spec_t *time* = uhd::time_spec_t::ASAP)

Read a 32-bit register implemented in the NoC block.

Parameters

addr	The byte address of the register to read from (truncated to 20 bits).
time	The time at which the transaction should be executed.

Exceptions

op_failed	if the transaction fails
op_timeout	if no response is received
op_seqerr	if a sequence error occurs

uint64_t register_iface::peek64 (uint32_t *addr*, time_spec_t *time* = uhd::time_spec_t::ASAP)[inline]

Read two consecutive 32-bit registers implemented in the NoC block and return them as one 64-bit value.

Note: This is a convenience call, because all register peeks are 32-bits. This will concatenate two peeks in a block peek, and then return the combined result of the two peeks.

Parameters

addr	The byte address of the lower 32-bit register to read from (truncated to 20 bits).
time	The time at which the transaction should be executed.

Exceptions

op_failed	if the transaction fails
op_timeout	if no response is received
op_seqerr	if a sequence error occurs

virtual void register_iface::poke32 (uint32_t *addr*, uint32_t *data*, uhd::time_spec_t *time* = uhd::time_spec_t::ASAP, bool *ack* = false)

Write a 32-bit register implemented in the NoC block.

Parameters

addr	The byte address of the register to write to (truncated to 20 bits).
data	New value of this register.
time	The time at which the transaction should be executed.
ack	Should transaction completion be acknowledged?

Exceptions

op_failed	if an ACK is requested and the transaction fails
op_timeout	if an ACK is requested and no response is received

op_seqerr	if an ACK is requested and a sequence error occurs
op_timeerr	if an ACK is requested and a time error occurs (late command)

void register_iface::poke64 (uint32_t *addr*, uint64_t *data*, time_spec_t *time* = uhd::time_spec_t::ASAP, bool *ack* = false)[inline]

Write two consecutive 32-bit registers implemented in the NoC block from one 64-bit value.

Note: This is a convenience call, because all register pokes are 32-bits. This will concatenate two pokes in a block poke, and then return the combined result of the two pokes.

Parameters

addr	The byte address of the lower 32-bit register to read from (truncated to 20 bits).
data	New value of the register(s).
time	The time at which the transaction should be executed.

Exceptions

op_failed	if the transaction fails
op_timeout	if no response is received
op_seqerr	if a sequence error occurs

virtual void register_iface::poll32 (uint32_t *addr*, uint32_t *data*, uint32_t *mask*, time_spec_t *timeout*, time_spec_t *time* = uhd::time_spec_t::ASAP, bool *ack* = false)

Poll a 32-bit register until its value for all bits in mask match data&mask

This will insert a command into the command queue to wait until a register is of a certain value. This can be used, e.g., to poll for a lock pin before executing the next command. It is related to **sleep()**, except it has a condition to wait on, rather than an unconditional stall duration. The timeout is hardware-timed. If the register does not attain the requested value within the requested duration, ${something bad happens}.

Example: Assume readback register 16 is a status register, and bit 0 indicates a lock is in place (i.e., we want it to be 1) and bit 1 is an error flag (i.e., we want it to be 0). The previous command can modify the state of the block, so we give it 1ms to settle. In that case, the call would be thus:

```
// iface is a register_iface::sptr:
iface->poll32(16, 0x1, 0x3, 1e-3);
```

Parameters

addr	The byte address of the register to read from (truncated to 20 bits).
data	The values that the register must have
mask	The bitmask that is applied before checking the readback value
timeout	The max duration that the register is allowed to take before reaching its new state.
time	When the poll should be executed
ack	Should transaction completion be acknowledged? This is typically only necessary if the software needs a condition to be

	fulfilled before continueing, or during debugging.

Exceptions

op_failed	if an ACK is requested and the transaction fails
op_timeout	if an ACK is requested and no response is received
op_seqerr	if an ACK is requested and a sequence error occurs
op_timeerr	if an ACK is requested and a time error occurs (late command)

virtual void register_iface::register_async_msg_handler (async_msg_callback_t *callback_f*)

Register a callback function for when an async message is received

Only one callback function can be registered. When calling this multiple times, only the last callback will be accepted.

Parameters

callback_f	The function to call when an asynchronous message is received.

virtual void register_iface::register_async_msg_validator (async_msg_validator_t *callback_f*)

Register a callback function to validate a received async message

The purpose of this callback is to provide a method to the framework to make sure a received async message is valid. If this callback is provided, the framework will first pass the message to the validator for validation. If the validator returns true, the async message is ACK'd with a ctrl_status_t::CMD_OKAY response, and then the async message is executed. If the validator returns false, then the async message is ACK'd with a ctrl_status_t::CMD_CMDERR, and the async message handler is not excecuted.

This callback may not communicate with the device, it can only look at the data and make a valid/not valid decision.

Only one callback function can be registered. When calling this multiple times, only the last callback will be accepted.

Parameters

callback_f	The function to call when an asynchronous message is received.

virtual void register_iface::set_policy (const std::string & *name*, const uhd::device_addr_t & *args*)

Set a policy that governs the operational parameters of this register bus. Policies can be used to make tradeoffs between performance, resilience, latency, etc.

Parameters

name	The name of the policy to apply
args	Additional arguments to pass to the policy governor

virtual void register_iface::sleep (time_spec_t *duration*, bool *ack* = false)

Send a command to halt (block) the control bus for a specified time. This is a hardware-timed sleep.

Parameters

duration	The amount of time to sleep.
ack	Should transaction completion be acknowledged?

Exceptions

op_failed	if an ACK is requested and the transaction fails
op_timeout	if an ACK is requested and no response is received
op_seqerr	if an ACK is requested and a sequence error occurs

3.2.3 Block Properties

A block *property* is a high-level representation of the state of the block. The author of the block can define zero or more properties to represent the total state of the block. The data type and number of properties is flexible and user-defined. Properties can be read and written using the public API of the block. A change in a particular property must be resolved for the state to correctly reflect in the FPGA counterpart of the block. This is done using user-defined *resolver* functions that depend on one or more properties. For example, for the Radio block, the analog gain can be a property of the type float and translate to a sequence of registers writes to program attenuators, etc. that will be sent using rfnoc::register_iface.

Block properties have two scopes:

- *User Scope*: User-scoped properties are defined by the block designer to be accessible by the user of the block. These are properties that can be read and written by the public API of the block using getter and setter functions, respectively. User-scoped properties are inherently related to the behavior of the block and not influenced by how the block is used (i.e., topology).
- *Port Scope*: Port-scoped properties are defined by the block designer to represent state inherited from the topology that the block is connected to. These properties travel along the data ports of the block. Port-scoped properties allow a state change in one block to accurately propagate to other blocks if there are dependencies as a result of the connection topology of the block. For this reason, port-scoped properties cannot be accessed directly using the public block API.

Properties are defined in the constructor of a block and must be registered with the framework. The API for a property is detailed below.

property_t<data_t> Class Template Reference

Public Member Functions

- const std::string & **get_id** () const
  Gets the string identifier for this property.
- const **res_source_info** & **get_src_info** () const
  Returns the source info for the property.
- bool **read_access_granted** () const
  Returns true if read access has been granted for this property.
- bool **write_access_granted** () const
  Returns true if write access has been granted for this property.
- void **set** (const data_t &value)
  Set the value of this property.

- const data_t & **get** () const
  Get the value of this property.
- **operator const data_t &** () const
- bool **operator==** (const data_t &rhs)
- bool **operator!=** (const data_t &rhs)
- **property_t**< data_t > & **operator=** (const data_t &value)
- **property_t**< data_t > & **operator=** (const **property_t**< data_t > &value)

3.2.3.1 Properties used by UHD

Properties are user-definable, and there is little limitation to what can be stored as properties. There are some properties that either have special meaning within UHD, or simply have a convention of what they mean and which type they are. While the framework imposes almost no restrictions on properties, it is highly recommended to follow these conventions so that RFNoC blocks stay compatible with one another. The following table contains a list of port properties (also known as edge properties) which parts of UHD interpret in a specific way.

Property ID	Type	Meaning
`tick_rate`	`double`	This property is created by the framework for every port. It signifies the number of ticks per second, i.e., it provides the data required to translate command times into ticks and vice versa. User-defined blocks cannot register properties with this key, since the property is already created. Interaction with these properties is also rarely required, as the API calls get_tick_rate() and set_tick_rate() are available to all block controllers.
`samp_rate`	`double`	These properties are read to or written by the Radio blocks, the DDC/DUC blocks, and the streamers. The unit is samples/sec. Blocks that require knowledge of the sampling rate, or produce data at a given rate, should implement these edge properties.
`type`	`string`	This port property identifies its data type. The value for this property is the same as the otw_type within uhd::stream_args, e.g., “sc16” for complex 16-bit data, “sc8”, “sc12”, “s16”, “f32”, “u8”, etc. Most blocks that implement this type are not actually capable of handling different data types, in which case the property resolvers should either throw an exception of type `resolve_error`, or should re-set the property value to the type they can handle. The Radio block, DDC, DUC, FFT, FIR filter, Vector IIR, and fosphor are all blocks that implement this property.

scaling	double	This property is relevant for DSP chains. For example, the Radio block, the DDC and the DUC, and the streamers all implement this. The only blocks that should implement this are blocks that have sample-in, sample-out flows, and that use fixed-point arithmetic on the FPGA that distorts the amplitude. The value of this property is the relative (multiplicative) amplitude error of a signal, also knows as its full-scale amplitude.

3.2.4 Block Actions

An action is an ephemeral operation that can be performed on a block. An action does not change the state of a block (although the block can implement an action handler that will change its state). Like a property, an action may propagate across the graph. Actions provide a mechanism for blocks to communicate in the software domain (i.e., not by sending command CHDR packets, but by calling into APIs provided by the graph). An example of an action is a stream command. A stream command that issues on a block that is not a source or a sink will eventually propagate through the topology to a source or sink, and be executed there. Actions are exposed through the public API via an action code and an action payload. Actions are defined by the block designer, and they can be handled internally in the block or be forwarded on an upstream or downstream port.

3.2.5 C++ API

The `noc_block_base` class provides functionality to access registers, properties and to execute actions on a block. The API for the block is detailed below (Note that all fields/functions that are public are intended for the users (clients) of a block; and all fields/functions that are protected are intended for the designers of custom block controllers).

noc_block_base Class Reference

Public Member Functions

- std::string **get_unique_id** () const
- size_t **get_num_input_ports** () const
- size_t **get_num_output_ports** () const
- noc_id_t **get_noc_id** () const
- const **block_id_t** & **get_block_id** () const
- double **get_tick_rate** () const
- size_t **get_mtu** (const **res_source_info** &edge)
- uhd::device_addr_t **get_block_args** () const
- uhd::property_tree::sptr & **get_tree** () const
- uhd::property_tree::sptr & **get_tree** ()

Protected Member Functions

- **noc_block_base** (make_args_ptr make_args)
- void **set_num_input_ports** (const size_t num_ports)
- void **set_num_output_ports** (const size_t num_ports)

- void **set_tick_rate** (const double tick_rate)
- void **set_mtu_forwarding_policy** (const **forwarding_policy_t** policy)
- void **set_mtu** (const **res_source_info** &edge, const size_t new_mtu)
- **property_base_t** * **get_mtu_prop_ref** (const **res_source_info** &edge)
- std::shared_ptr< mb_controller > **get_mb_controller** ()
- virtual void **deinit** ()

Detailed Description

The primary interface to a NoC block in the FPGA.

The block supports three types of data access:

- Low-level register access
- High-level property access
- Action execution

main difference between this class and its parent is the direct access to registers, and the NoC- and block IDs.

Member Function Documentation

virtual void noc_block_base::deinit ()

Safely de-initialize the block

This function is called by the framework when the RFNoC session is about to finish to allow blocks to safely perform actions to shut down a block. For example, if your block is producing samples, like a radio or signal generator, this is a good place to issue a "stop" command.

After this function is called, register access is no more possible. So make sure not to interact with regs() after this was called. Future access to regs() won't throw, but will print error messages and do nothing.

The rationale for having this separate from the destructor is because **rfnoc_graph** allows exporting references to blocks, and this function ensures that blocks are safely shut down when the rest of the device control goes away.

uhd::device_addr_t noc_block_base::get_block_args () const

Return the arguments that were passed into this block from the framework

const block_id_t& noc_block_base::get_block_id () const

Returns the unique block ID for this block.

Returns

block_id The block ID of this block (e.g. "0/FFT#1")

std::shared_ptr<mb_controller> noc_block_base::get_mb_controller ()[protected]

Get access to the motherboard controller for this block's motherboard

This will return a nullptr if this block doesn't have access to the motherboard. In order to gain access to the motherboard, the block needs to have requested access to the motherboard during the registration procedure. See also registry.hpp.

Even if this block requested access to the motherboard controller, there is no guarantee that UHD will honour that request. It is therefore important to verify that the returned pointer is valid.

size_t noc_block_base::get_mtu (const res_source_info & *edge*)

Return the current MTU on a given edge

The MTU is determined by the block itself (i.e., how big of a packet can this block handle on this edge), but also the neighboring block, and possibly the transport medium between the blocks. This value can thus be lower than what the block defines as MTU, but never higher.

Parameters

edge	The edge on which the MTU is queried. edge.type must be INPUT_EDGE or OUTPUT_EDGE!

Returns

the MTU as determined by the overall graph on this edge

Exceptions

uhd::value_error	if edge is not referring to a valid edge

property_base_t* noc_block_base::get_mtu_prop_ref (const res_source_info & *edge*)[protected]

Return a reference to an MTU property

This can be used to make the MTU an input to a property resolver. For example, blocks that have an spp property, such as the radio, can now trigger a property resolver based on the MTU.

noc_id_t noc_block_base::get_noc_id () const

Return the NoC ID for this block.

Returns

noc_id The 32-bit NoC ID of this block

size_t noc_block_base::get_num_input_ports () const, [virtual]

Number of input ports. Note: This gets passed into this block from the.

Implements **node_t** (*p.**Error! Bookmark not defined.***).

size_t noc_block_base::get_num_output_ports () const, [virtual]

Number of output ports. Note: This gets passed outto this block from the.

Implements **node_t** (*p.**Error! Bookmark not defined.***).

double noc_block_base::get_tick_rate () const

Returns the tick rate of the current time base

Note there is only ever one time base (or tick rate) per block.

uhd::property_tree::sptr& noc_block_base::get_tree ()

Return a reference to this block's subtree (non-const version)

uhd::property_tree::sptr& noc_block_base::get_tree () const

Return a reference to this block's subtree.

std::string noc_block_base::get_unique_id () const, [virtual]

Unique ID for an RFNoC block is its block ID.

Reimplemented from **node_t** (*p.73*).

void noc_block_base::set_mtu (const res_source_info & *edge*, const size_t *new_mtu*)[protected]

Update the MTU

This is another data point in the MTU discovery process. This means that the MTU cannot be increased using the method, only decreased.

void noc_block_base::set_mtu_forwarding_policy (const forwarding_policy_t *policy*)[protected]

Change the way MTUs are forwarded

The policy will have the following effect:

DROP: This means that the MTU of one port has no bearing on the MTU of another port. This is usually a valid choice if the FPGA is repacking data, for example, a block could be consuming continous streams of data, and producing small packets of a different type.

ONE_TO_ONE: This means the MTU is passed through from input to output and vice versa. This is typically a good choice if packets are being passed through without modifying their size. The DDC/DUC blocks will choose this policy, because the want to relay MTU information to the radio.

ONE_TO_ALL: This means the MTU is being set to the same value on all ports.

ONE_TO_FAN: This means the MTU is forwarded from any input port to all opposite side ports. This is an appropriate policy for the split-stream block.

The default policy is DROP.

void noc_block_base::set_num_input_ports (const size_t *num_ports*)[protected]

Update number of input ports.

void noc_block_base::set_num_output_ports (const size_t *num_ports*)[protected]

Update number of output ports.

void noc_block_base::set_tick_rate (const double *tick_rate*)[protected]

Update tick rate for this node and all the connected nodes

Careful: Calling this function will trigger a property propagation to any block this block is connected to.

3.2.6 Custom Block Controllers

Custom block controllers can be built by inheriting from `noc_block_base`. The protected functions in the base class must be used to correctly register properties and handle actions. UHD will provide a mechanism to retrieve controllers for discovered blocks, and it will have the capability to `dynamic_cast` the generic block controller to the user-defined type. The following is an example of a sample custom block controller.

```
class null_block_control_impl : public null_block_control
{
public:
    RFNOC_BLOCK_CONSTRUCTOR(null_block_control)
    {
        uint32_t initial_state = regs().peek32(REG_CTRL_STATUS);
        _nipc = (initial_state >> 24) & 0xFF;
        _item_width = (initial_state >> 16) & 0xFF;
        register_property(&_lpp);
        add_property_resolver(
            {&_lpp}, // Input/trigger list
            {&_lpp}, // Output list
            [this](){
                set_lines_per_packet(_lpp.get());
                _lpp = get_lines_per_packet();
            });
        register_issue_stream_cmd();
    }

    void issue_stream_cmd(const stream_cmd_t& stream_cmd)
    {
        /* Implement functionality to start and stop streaming */
```

```
    }

    void set_lines_per_packet(const uint32_t lpp)
    {
        const uint32_t reg_val = /* figure out from lpp */;
        regs().poke32(REG_SRC_LINES_PER_PKT, reg_val);
    }

    uint32_t get_lines_per_packet()
    {
        return regs().peek32(REG_SRC_LINES_PER_PKT) + 2;
    }

private:

    /*! Action API: Register a handler for stream commands
     */
    void register_issue_stream_cmd()
    {
        register_action_handler(ACTION_KEY_STREAM_CMD,
            [this](const res_source_info& src, action_info::sptr action) {
                stream_cmd_action_info::sptr stream_cmd_action =
                    std::dynamic_pointer_cast<stream_cmd_action_info>(action);
                if (!stream_cmd_action) {
                    throw uhd::runtime_error(
                        "Received stream_cmd of invalid action type!");
                }
                if (src.instance != 0 || src.type != res_source_info::OUTPUT_EDGE) {
                    throw uhd::runtime_error(
                        "The null source can only stream from output port 0!");
                }
                RFNOC_LOG_DEBUG("Received stream command action request!");
                issue_stream_cmd(stream_cmd_action->stream_cmd);
            });
    }

    void deinit()
    {
        // This is the last time we can do any kind of peek or poke
        issue_stream_cmd(stream_cmd_t::STREAM_MODE_STOP_CONTINUOUS);
    }


    /**************************************************************************
     * Attributes
     *************************************************************************/
    property_t<int> _lpp{"lpp", 100, {res_source_info::USER}};

    //! Number of items per clock
    uint32_t _nipc;

    //! Bits per item
    uint32_t _item_width;
};
```

3.3 RFNoC Graph

The graph object in UHD allows users to build functional applications using individual blocks by connecting them together in a meaningful topology. The RFNoC graph has the following properties:

- The *edges* of the graph represent the data flow paths. It is thus directed.
- There can be multiple connections between nodes (e.g., for blocks that handle MIMO applications). It is thus a multigraph.
- A block can have multiple inputs or outputs (e.g., a summation block might have two inputs and one output). A property of an edge is thus which port of a node it is connected to. If there are multiple edges between the same nodes, they are not interchangeable.
- It is *disconnected*, meaning there can be multiple, disconnected sub-graphs.
- It cannot have *loose (unconnected) edges*.
- If the blocks support such a usage, data connections can be cyclic.
- All nodes have a unique identifier of type `string`.

3.3.1 Capabilities

There are three types of connections (edges) supported by the graph:

- Block to Block: A connection between two blocks that are connected by some transport
- Block to Stream: A connection between a block in the FPGA and a host-resident RX streamer
- Stream to Block: A connection between a host-resident TX streamer and a block in the FPGA

The graph object is responsible for making the appropriate data connections between blocks and/or the host streamers. It is also responsible for propagating port (edge) properties between blocks. Multi-block state resolution is handled by the framework and is not the responsibility of the application designer.

Note: The architecture of RFNoC does not preclude software running on a remote computer that acts like an RFNoC device. From the perspective of the graph, this would also register as a block controller, even if the remote computer is using a streamer object. When the software is running on the same context as the UHD session, it is considered a streamer.

3.3.2 C++ API

rfnoc_graph Class Reference

Public Member Functions

- virtual ~rfnoc_graph ()
- virtual std::vector< **block_id_t** > **find_blocks** (const std::string &block_id_hint) const
- template<typename T > std::vector< **block_id_t** > **find_blocks** (const std::string &block_id_hint) const
- virtual bool **has_block** (const **block_id_t** &block_id) const

- template<typename T > bool **has_block** (const **block_id_t** &block_id) const
- virtual noc_block_base::sptr get_block (const block_id_t &block_id) const
- template<typename T > std::shared_ptr< T > **get_block** (const **block_id_t** &block_id) const
- virtual bool **is_connectable** (const **block_id_t** &src_blk, size_t src_port, const **block_id_t** &dst_blk, size_t dst_port)
- virtual void **connect** (const **block_id_t** &src_blk, size_t src_port, const **block_id_t** &dst_blk, size_t dst_port, bool skip_property_propagation=false)
- virtual void **connect** (uhd::tx_streamer::sptr streamer, size_t strm_port, const **block_id_t** &dst_blk, size_t dst_port, uhd::transport::adapter_id_t adapter_id=uhd::transport::NULL_ADAPTER_ID)
- virtual void **connect** (const **block_id_t** &src_blk, size_t src_port, uhd::rx_streamer::sptr streamer, size_t strm_port, uhd::transport::adapter_id_t adapter_id=uhd::transport::NULL_ADAPTER_ID)
- virtual std::vector< uhd::transport::adapter_id_t > **enumerate_adapters_from_src** (const **block_id_t** &src_blk, size_t src_port)
- virtual std::vector< uhd::transport::adapter_id_t > **enumerate_adapters_to_dst** (const **block_id_t** &dst_blk, size_t dst_port)
- virtual std::vector< graph_edge_t > **enumerate_static_connections** () const
- virtual std::vector< graph_edge_t > **enumerate_active_connections** ()
- virtual void **commit** ()
- virtual void **release** ()
- virtual **rx_streamer::sptr create_rx_streamer** (const size_t num_ports, const stream_args_t &args)
- virtual **tx_streamer::sptr create_tx_streamer** (const size_t num_ports, const stream_args_t &args)
- virtual size_t **get_num_mboards** () const
- virtual std::shared_ptr< mb_controller > **get_mb_controller** (const size_t mb_index=0)
- virtual bool **synchronize_devices** (const uhd::time_spec_t &time_spec, const bool quiet)
- virtual uhd::property_tree::sptr **get_tree** (void) const

Static Public Member Functions

- static **sptr make** (const device_addr_t &dev_addr)

Detailed Description

The core class for a UHD session with (an) RFNoC device(s)

This class is a superset of uhd::device. It does not only hold a device session, but also manages the RFNoC blocks on those devices. Only devices compatible with a modern version of RFNoC can be addressed by this class.

Member Function Documentation

virtual void rfnoc_graph::commit ()

Commit graph and run initial checks

This method needs to be called when the graph is ready for action. It will run checks on the graph and run a property propagation.

Exceptions

uhd::resolve_error	if the properties fail to resolve.

virtual void rfnoc_graph::connect (const block_id_t & *src_blk*, size_t *src_port*, const block_id_t & *dst_blk*, size_t *dst_port*, bool *skip_property_propagation* = false)

Connect a RFNOC block with block ID `src_block` to another with block ID `dst_block`.

Note you need to also call this on statically connected blocks if you desire to use them.

Parameters

src_blk	The block ID of the source block to connect.

src_port	The port of the source block to connect.
dst_blk	The block ID of the destination block to connect to.
dst_port	The port of the destination block to connect to.
skip_property_propagation	Skip property propagation for this edge

Exceptions

uhd::routing_error	if the source or destination ports are statically connected to a *different* block

virtual void rfnoc_graph::connect (const block_id_t & *src_blk*, size_t *src_port*, uhd::rx_streamer::sptr *streamer*, size_t *strm_port*, uhd::transport::adapter_id_t *adapter_id* = uhd::transport::NULL_ADAPTER_ID)

Connect RX streamer to an output of an NoC block

Parameters

src_blk	The block ID of the source block to connect.
src_port	The port of the source block to connect.
streamer	The streamer to connect.
strm_port	The port of the streamer to connect.
adapter_id	The local device ID (transport) to use for this connection.

Exceptions

connect_disallowed_on_src	if the source port is statically connected to a *different* block

virtual void rfnoc_graph::connect (uhd::tx_streamer::sptr *streamer*, size_t *strm_port*, const block_id_t & *dst_blk*, size_t *dst_port*, uhd::transport::adapter_id_t *adapter_id* = uhd::transport::NULL_ADAPTER_ID)

Connect TX streamer to an input of an NoC block

Parameters

streamer	The streamer to connect.
strm_port	The port of the streamer to connect.
dst_blk	The block ID of the destination block to connect to.
dst_port	The port of the destination block to connect to.
adapter_id	The local device ID (transport) to use for this connection.

Exceptions

connect_disallowed_on_dst	if the destination port is statically connected to a *different* block

virtual rx_streamer::sptr rfnoc_graph::create_rx_streamer (const size_t *num_ports*, const stream_args_t & *args*)

Create a new receive streamer from the streamer arguments The created streamer is still not connected to anything yet. The graph::connect call has to be made on this streamer to start using it. If a different streamer is already connected to the intended source then that call may fail.

Parameters

num_ports	Number of ports that will be connected to the streamer
args	Arguments to aid the construction of the streamer

Returns

a shared pointer to a new streamer

virtual tx_streamer::sptr rfnoc_graph::create_tx_streamer (const size_t *num_ports*, const stream_args_t & *args*)

Create a new transmit streamer from the streamer arguments The created streamer is still not connected to anything yet. The graph::connect call has to be made on this streamer to start using it. If a different streamer is already connected to the intended sink then that call may fail.

Parameters

num_ports	Number of ports that will be connected to the streamer
args	Arguments to aid the construction of the streamer

Returns

a shared pointer to a new streamer

virtual std::vector<graph_edge_t> rfnoc_graph::enumerate_active_connections ()

Enumerate all the active connections in the graph

Returns

A vector containing all the active edges in the graph.

virtual std::vector<uhd::transport::adapter_id_t> rfnoc_graph::enumerate_adapters_from_src (const block_id_t & *src_blk*, size_t *src_port*)

Enumerate all the possible host transport adapters that can be used to receive from the specified block

If addr and second_addr were specified in device_args, the adapter_id_t associated with addr will come first in the vector, then second_addr.

Parameters

src_blk	The block ID of the source block to connect to.
src_port	The port of the source block to connect to.

virtual std::vector<uhd::transport::adapter_id_t> rfnoc_graph::enumerate_adapters_to_dst (const block_id_t & *dst_blk*, size_t *dst_port*)

Enumerate all the possible host transport adapters that can be used to send to the specified block

If addr and second_addr were specified in device_args, the adapter_id_t associated with addr will come first in the vector, then second_addr.

Parameters

dst_blk	The block ID of the destination block to connect to.
dst_port	The port of the destination block to connect to.

virtual std::vector<graph_edge_t> rfnoc_graph::enumerate_static_connections () const

Enumerate all the possible static connections in the graph

Returns

A vector containing all the static edges in the graph.

template<typename T > std::vector<block_id_t> rfnoc_graph::find_blocks (const std::string & *block_id_hint*) const

Type-cast version of **find_blocks()**.

virtual std::vector<block_id_t> rfnoc_graph::find_blocks (const std::string & *block_id_hint*) const

Returns the block ids of all blocks that match the specified hint Uses **block_id_t::match()** internally. If no matching block is found, it returns an empty vector.

To access specialized block controller classes (i.e. derived from **noc_block_base**), use the templated version of this function, e.g.

```
// Assume DEV is an rfnoc_graph::sptr
auto null_blocks = DEV->find_blocks<null_noc_block>("NullSrcSink");
if (null_blocks.empty()) { cout << "No null blocks found!" << endl; }
```

Note

this access is not thread safe if performed during block enumeration

template<typename T > std::shared_ptr<T> rfnoc_graph::get_block (const block_id_t & *block_id*) const

Same as **get_block()**, but with a type cast.

If you have a block controller class that is derived from **noc_block_base**, use this function to access its specific methods. If the given block ID is not valid (i.e. such a block does not exist on this device) or if the type does not match, it will throw a uhd::lookup_error.

```
// Assume DEV is a device3::sptr
auto block_controller = get_block<my_noc_block>("0/MyBlock#0");
block_controller->my_own_block_method();
```

Note

this access is not thread safe if performed during block enumeration

virtual noc_block_base::sptr rfnoc_graph::get_block (const block_id_t & *block_id*) const

Returns a block controller class for an NoC block.

If the given block ID is not valid (i.e. such a block does not exist on this device), it will throw a uhd::lookup_error.

Parameters

block_id	Canonical block name (e.g. "0/FFT#1").

Note

this access is not thread safe if peformed during block enumeration

virtual std::shared_ptr<mb_controller> rfnoc_graph::get_mb_controller (const size_t *mb_index* = 0)

Return a reference to a motherboard controller.

virtual size_t rfnoc_graph::get_num_mboards () const
virtual uhd::property_tree::sptr rfnoc_graph::get_tree (void) const

Return a reference to the property tree.

template<typename T > bool rfnoc_graph::has_block (const block_id_t & *block_id*) const

Same as **has_block()**, but with a type check.

Returns

true if a block of type T with the specified id exists

Note

this access is not thread safe if performed during block enumeration

virtual bool rfnoc_graph::has_block (const block_id_t & *block_id*) const

Checks if a specific NoC block exists on the device.

Parameters

block_id	Canonical block name (e.g. "0/FFT#1").

Returns

true if a block with the specified id exists

Note

this access is not thread safe if performed during block enumeration

virtual bool rfnoc_graph::is_connectable (const block_id_t & *src_blk*, size_t *src_port*, const block_id_t & *dst_blk*, size_t *dst_port*)

Verify if two blocks/ports are connectable.

If this call returns true, then **connect()** can be called with the same arguments. It does not, however, check if the block was already connnected.

Returns

true if the two blocks are connectable

static sptr rfnoc_graph::make (const device_addr_t & *dev_addr*)[static]

Make a new USRP graph from the specified device address(es).

Parameters

dev_addr	the device address

Returns

A new **rfnoc_graph** object

Exceptions

uhd::key_error	no device found
uhd::index_error	fewer devices found than expected

virtual void rfnoc_graph::release ()

Release graph: Opposite of **commit()**

Calling this will disable property propagation until **commit()** has been called an equal number of times.

virtual bool rfnoc_graph::synchronize_devices (const uhd::time_spec_t & *time_spec*, const bool *quiet*)

Run any routines necessary to synchronize devices

The specific implementation of this call are device-specific. In all cases, it will set the time to a common value.

Any application that requires any kind of phase or time alignment (if supported by the hardware) must call this before operation.

Parameters

time_spec	The timestamp to be used to sync the devices. It will be an input to set_time_next_pps() on the motherboard controllers.
quiet	If true, there will be no errors or warnings printed if the synchronization happens. This call will always be called during

	initialization, but preconditions might not yet be met (e.g., the time and reference sources might still be internal), and will fail quietly in that case.

Returns

the success status of this call (true means devices are now synchronized)

3.4 Streamers

Streamers allow users to send data to or receive data from an RFNoC block. The API for streamers will be the same as what multi_usrp has today.

rx_streamer Class Reference

Public Types

- typedef std::shared_ptr< **rx_streamer** > **sptr**
- typedef ref_vector< void * > **buffs_type**

Public Member Functions

- size_t **get_num_channels** (void) const
- size_t **get_max_num_samps** (void) const
- size_t **recv** (const **buffs_type** &buffs, const size_t nsamps_per_buff, rx_metadata_t &metadata, const double timeout=0.1, const bool one_packet=false)
- void **issue_stream_cmd** (const stream_cmd_t &stream_cmd)

Detailed Description

The RX streamer is the host interface to receiving samples. (Unchanged in UHD)

Member Typedef Documentation

typedef ref_vector<void *> rx_streamer::buffs_type

Typedef for a pointer to a single, or a collection of recv buffers

typedef std::shared_ptr<rx_streamer> rx_streamer::sptr

A shared pointer to allow easy access to this class and for automatic memory management.

Member Function Documentation

size_t rx_streamer::get_max_num_samps (void) const

Get the max number of samples per buffer per packet

size_t rx_streamer::get_num_channels (void) const

Get the number of channels associated with this streamer

void rx_streamer::issue_stream_cmd (const stream_cmd_t & *stream_cmd*)

Issue a stream command to the usrp device. This tells the usrp to send samples into the host. See the documentation for stream_cmd_t for more info.

With multiple devices, the first stream command in a chain of commands should have a time spec in the near future and stream_now = false; to ensure that the packets can be aligned by their time specs.

Parameters:

stream_cmd	the stream command to issue

size_t rx_streamer::recv (const buffs_type & *buffs*, const size_t *nsamps_per_buff*, rx_metadata_t & *metadata*, const double *timeout* = `0.1`, const bool *one_packet* = `false`)

Receive buffers containing samples described by the metadata.

Receive handles fragmentation as follows: If the buffer has insufficient space to hold all samples that were received in a single packet over-the-wire, then the buffer will be completely filled, and the implementation will hold a pointer into the remaining portion of the packet. Subsequent calls will load from the remainder of the packet and will flag the metadata to show that this is a fragment. The next call to receive, after the remainder becomes exhausted, will perform an over-the-wire receive as usual. See the rx metadata fragment flags and offset fields for details.

This is a blocking call and will not return until the number of samples returned have been written into each buffer. Under a timeout condition, the number of samples returned may be less than the number of samples specified.

The one_packet option allows the user to guarantee that the call will return after a single packet has been processed. This may be useful to maintain packet boundaries in some cases.

Note on threading: **recv()** is *not* thread-safe, to avoid locking overhead. The application calling **recv()** is responsible for making sure that not more than one thread can call **recv()** at the same time.

Parameters:

buffs	a vector of writable memory to fill with samples
nsamps_per_buff	the size of each buffer in number of samples
metadata	data to fill describing the buffer
timeout	the timeout in seconds to wait for a packet
one_packet	return after the first packet is received

Returns:

the number of samples received or 0 on error

---

tx_streamer Class Reference

Public Types

- typedef std::shared_ptr< **tx_streamer** > **sptr**
- typedef ref_vector< const void * > **buffs_type**

Public Member Functions

- size_t **get_num_channels** (void) const
- size_t **get_max_num_samps** (void) const
- size_t **send** (const **buffs_type** &buffs, const size_t nsamps_per_buff, const tx_metadata_t &metadata, const double timeout=0.1)

- bool **recv_async_msg** (async_metadata_t &async_metadata, double timeout=0.1)

Detailed Description

The TX streamer is the host interface to transmitting samples. (Unchanged in UHD)

Member Typedef Documentation

typedef ref_vector<const void *> tx_streamer::buffs_type

Typedef for a pointer to a single, or a collection of send buffers

typedef std::shared_ptr<tx_streamer> tx_streamer::sptr

A shared pointer to allow easy access to this class and for automatic memory management.

Member Function Documentation

size_t tx_streamer::get_max_num_samps (void) const

Get the max number of samples per buffer per packet

size_t tx_streamer::get_num_channels (void) const

Get the number of channels associated with this streamer

bool tx_streamer::recv_async_msg (async_metadata_t & *async_metadata*, double *timeout* = `0.1`)

Receive and asynchronous message from this TX stream.

Parameters:

async_metadata	the metadata to be filled in
timeout	the timeout in seconds to wait for a message

Returns:

true when the async_metadata is valid, false for timeout

size_t tx_streamer::send (const buffs_type & *buffs*, const size_t *nsamps_per_buff*, const tx_metadata_t & *metadata*, const double *timeout* = `0.1`)

Send buffers containing samples described by the metadata.

Send handles fragmentation as follows: If the buffer has more items than the maximum per packet, the send method will fragment the samples across several packets. Send will respect the burst flags when fragmenting to ensure that start of burst can only be set on the first fragment and that end of burst can only be set on the final fragment.

This is a blocking call and will not return until the number of samples returned have been read out of each buffer. Under a timeout condition, the number of samples returned may be less than the number of samples specified.

Parameters:

buffs	a vector of read-only memory containing samples
nsamps_per_buff	the number of samples to send, per buffer
metadata	data describing the buffer's contents
timeout	the timeout in seconds to wait on a packet

Returns:

the number of samples sent

3.5 Motherboard Controllers

The motherboards are represented by a “motherboard controller”. This object can be used to configure, modify, or query properties that are specific to the motherboard itself, and are not tied to any particular block. A very common motherboard-level setting is the source for time and/or clock reference. The timekeepers are also controlled on the motherboard level and are controlled through the motherboard controller.

The indexing for motherboard controllers is consistent between block IDs and device arguments. For example, suppose an `rfnoc_graph` was created with the arguments `addr0=192.168.10.2,addr1=192.168.10.3`, in which case there will be two motherboards in this RFNoC graph. A call to `get_mb_controller(0)` will return the motherboard controller for the first motherboard (the one who’s IP address ends in 10.2 in this example), and the block ID 0/Radio#0 will correspond to the first radio on this same motherboard.

In RFNoC versions prior to UHD 4.0, the only API to interact with the graph were the block controllers. This left an API gap, and some API calls that affected the motherboard were attached to the radio block controllers instead. By splitting up block controllers and motherboard controllers, API calls are attached to the actual component they’re controlling.

However, block controllers may request access to the motherboard controllers themselves, which they sometimes require (e.g., to control or query clocks on the motherboard). This mechanism may also allow blocks, such as the radio blocks, to expose functionality that is technically tied to the motherboard.

3.6 uhd::multi_usrp API

An RFNoC capable USRP must work out of the box using the multi_usrp API. To do so, multi_usrp will expect a default image with Radios, DDCs, DUCs and buffering to implement the native USRP API. Internally, multi_usrp will build a graph to make the appropriate connections, much like a user application.

4 RFNoC Tools Overview

4.1 Basics

The RFNoC framework provides code generation tools to create blocks and to assemble an FPGA design using existing blocks. All tools have a command line interface (CLI) and graphical user interface (GUI). The block creation tool, called RFNoC ModTool, accepts basic parameters (Section 2.3.1.3), control-plane parameters (Section 2.3.2.3), data-plane parameters (Section 2.3.3.4), and other user options to generate Verilog and C++ code templates for a new block, units tests and the supporting metadata files for design assembly and for use by UHD. After a basic template for a block has been created, users can iteratively develop the FPGA and software implementation for the block, and then move to the next step of design assembly. The design assembly tool, called RFNoC Image Builder, accepts parameters and performs the steps described in Section 2.4.2 to build an FPGA image that instantiates blocks from the local block database, with connections specified statically at compile time. The generated image can then be deployed onto a USRP for UHD to automatically detect and target the blocks on the device.

User preferences can be communicated using files or GUI actions. The YAML format is used to describe all user-options and the XML format is used for generated files. In almost all cases, it should not be necessary to modify the XML files generated by the tools. The overview and interaction of the tools is shown in Figure 16 and is described in the following sections.

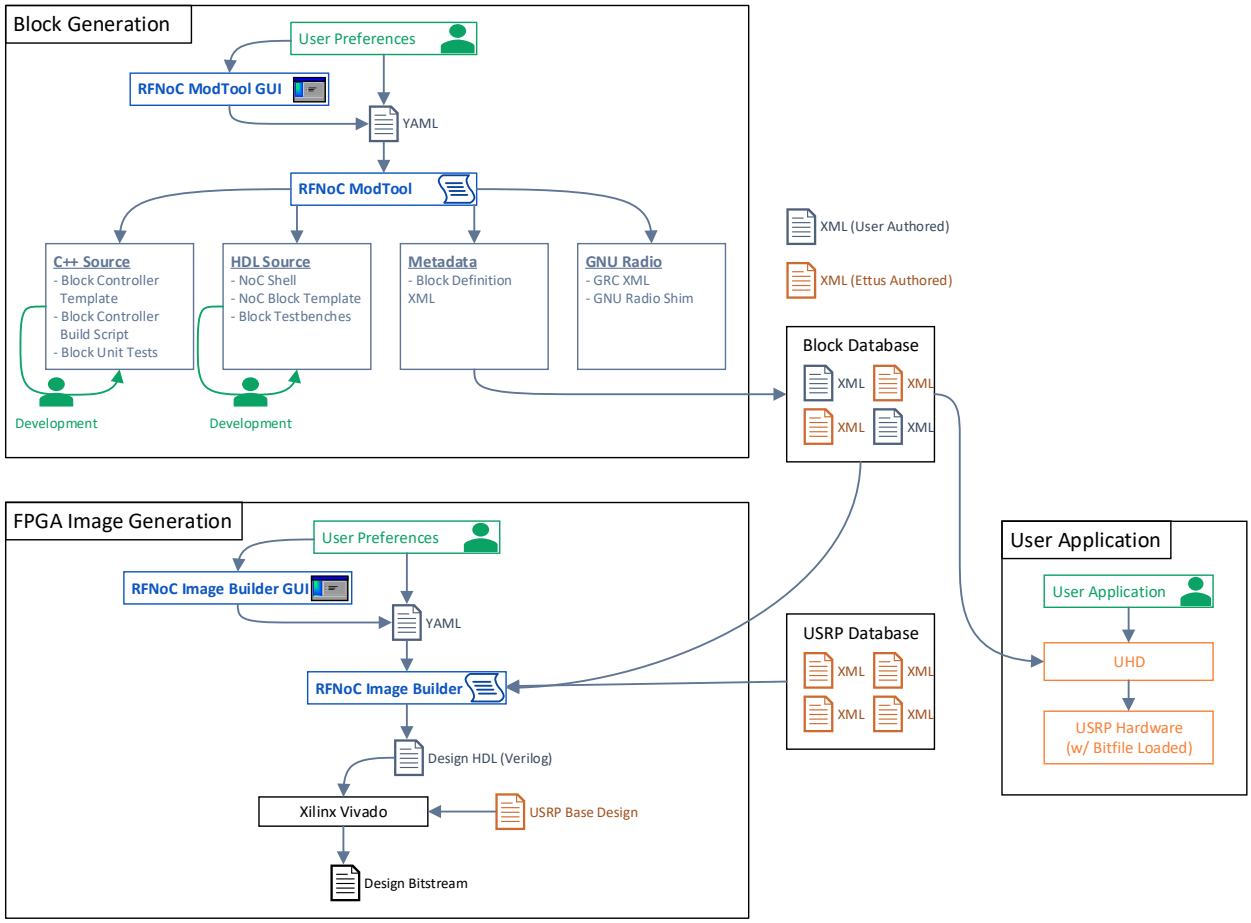

Figure 16: RFNoC Tool flow Overview

4.2 RFNoC ModTool

4.2.1 Overview

RFNoC ModTool should be used to generate a new RFNoC block which may have custom user-defined logic. The FPGA and software interfaces to the block are detailed in Section 2.3 and Section 3.2. The inputs to RFNoC ModTool are described above, and using these *User Preferences*, the tool will generate the following files:

- C++ Source
  - *Block Controller Template*: The block controller template contains boilerplate UHD code to communicate with the block in the FPGA. It will contain a basic register interface, and placeholders to define and implement block arguments and block properties.
  - *Block Controller Build Script*: The build script is a cmake project that can be used to build a dynamic library that UHD can call into to instantiate the custom block.
  - *Unit Tests*: A template to implement basic unit tests to validate the block controller.
- HDL Source

    - *NoC Shell*: A fully functional Verilog NoC Shell that has all the interfaces requested by the user.
    - *NoC Block Template*: A Verilog template for the NoC Block, which includes an instantiation for the NoC shell and a placeholder for users to insert their custom logic.
    - *Block Testbench Template*: A testbench template to allow users to write HDL unit tests for their block.
- GNU Radio Bindings (if GNU Radio is installed)
    - A GRC XML file to include the block into a GNU Radio flow graph
    - Any additional shim code to enable the block to function in GNU Radio
- Metadata
    - *Block Definition File*: This is an XML file that defines the interfaces and behavior of the block. The block definition file is used by UHD to discover capabilities of the block and to load the appropriate block controller class. It is also used by the RFNoC Image Builder (see below) to assemble an FPGA design using the block.

The user can interact with RFNoC ModTool using a GUI or using the CLI and specifying a YAML file with following format.

4.2.2 Input Format

The following is a description (and example) of the input YAML format.

```
---
# General parameters
# ----------------------------------------
schema: rfnoc_modtool_args      # Name of the schema used to validate this file
module_name: my_block           # Name of the RFNoC block
version: "1.0"                  # Format version of this file
rfnoc_version: "1.0"            # Version of RFNoC
chdr_width: 256                 # Bit width of the CHDR bus
noc_id: 0xDEADBEEF              # NoC ID for this block

# A list of all clocks needed by this block
# ----------------------------------------
# - rfnoc_chdr_clk and rfnoc_ctrl_clk are required clocks
# - All other clocks will be considered as user-defined clocks
clocks:
  - rfnoc_chdr:                     # Clock name prefix
      freq: 'range(100e6, 300e6)'   # Acceptable frequency range of this clock
  - rfnoc_ctrl:
      freq: 'range(10e6, 100e6)'
  - user0:
      freq: 'range(0, 1e9)'

# Options for the control interface
# ---------------------------------
```

```
control:
  sw_iface: nocscript           # Software controller implementation: {nocscript, c++}
  fpga_iface: axis_ctrl         # Type of FPGA interface: {axis_ctrl, ctrlport}
  interface_direction: slave    # Direction of control endpoint:
                                #   {slave, master_slave, remote_master_slave}
  fifo_depth: 32                # Number of 32-bit words in input buffer: [32, 4096]
  clk_domain: rfnoc_ctrl        # Clock domain for ctrl interface: {<Choose from "clocks">}
  ctrlport:                     # ctrlport specific options
    byte_mode: True             # Instantiate a byte enable: {True, False}
    timed: False                # Allow timed commands: {True, False}
    has_status: False           # Instantiate a status bus: {True, False}
  axis_ctrl:                    # axis_ctrl specific
    64_bit: False               # Instantiate a 64-bit bus instead of 32: {True, False}

# Options for the data interface
# ------------------------------
data:
  fpga_iface: axis_pyld_ctxt    # Type of FPGA interface:
                                #   {axis_chdr, axis_pyld_ctxt, axis_data}
  clk_domain: user0             # Clock domain for data interface: {<Choose from "clocks">}
  # A list of all input ports for this block:
  inputs:
    in0:                        # Port name
      context: True             # Is context port instantiated?: {True, False}
      num_ports: 2              # Optional number of ports (if not 1): [1, 64]
      item_width: 32            # Bit width of a sample
      nipc: 2                   # Number of samples per cycle (items per cycle)
      format: sc16              # Sample data format: {int16, sc8, sc16, ...}
      mdata_sig: ~              # Hash of the metadata signature: {~, MD5 sum}
      context_fifo_depth: 32    # Depth of context FIFO: Powers of 2 in [1, ∞)
      payload_fifo_depth: 32    # Depth of payload FIFO: Powers of 2 in [1, ∞)
    in1:
      context: True
      item_width: 16
      nipc: 4
      format: int16
      mdata_sig: 0412ffc5e7e1a19d8d23b4e288b3ced2
      context_fifo_depth: 32
      payload_fifo_depth: 32
  # A list of all output ports for this block:
  outputs:
    out0:
      context: True
      item_width: 32
      nipc: 2
      format: sc16
      mdata_sig: 0412ffc5e7e1a19d8d23b4e288b3ced4
```

```
      context_fifo_depth: 32
      payload_fifo_depth: 32
    out_1:
      context: True
      item_width: 16
      nipc: 4
      format: int16
      mdata_sig: ~
      context_fifo_depth: 32
      payload_fifo_depth: 32

# A list of all IO ports for this block
# -------------------------------------
io_port:
  timestamp:                  # Name of IO port
    type: time_keeper         # Descriptor for the IO signature of this port
    drive: listener           # Drive mode for port: {master, slave, listener, broadcaster}
  custom_xy:
    type: my_iface_sic
    drive: slave

# A list of all registers in the block
# ------------------------------------
registers:
  - user_reg_0:                # Register name
      offset: 0x0000           # Byte offset of the register in the block memory space
  - user_reg_1:
      offset: 0x0004

# A list of all user properties for the block
# (Edge properties not supported in nocscript)
# --------------------------------------------
properties:
  - user_arg_0:                                 # Name of argument
    type: uint32_t                             # C++ data-type of argument
    nocscript: 'REG_WRITE(user_reg_0, $val)'  # NoC script code to execute when set
  - user_arg_1:
    type: string
    nocscript: 'REG_WRITE(user_reg_1, $val)'
  - user_arg_2:
    type: int32_t
    nocscript: ''
...
```

4.3 RFNoC Image Builder

4.3.1 Overview

RFNoC Image Builder should be used to generate an FPGA design and a bitstream using blocks provided by Ettus Research or created by the user. RFNoC Image Builder will generate Verilog to instantiate blocks requested by the user, connect them and integrate all components with the USRP board support package, to create a full design that can be synthesized and built into a bitstream. The code and the bitstream is the only output of this tool.

The user can interact with RFNoC Image Builder using a GUI or using the CLI and specifying a YAML file with following format.

4.3.2 Input Format

The following is an example of the input YAML format.

```
---
# General parameters
# ----------------------------------------
schema: rfnoc_imagebuilder_args         # Identifier for the schema used to validate this file
version: "1.0"                          # File version
rfnoc_version: "1.0"                    # RFNoC protocol version
chdr_width: 64                          # Bit width of the CHDR bus for this image
device: 'x310'                          # USRP device to build for
default_target: 'X310_HG'               # Default FPGA image type to build


# A list of all stream endpoints in design
# ----------------------------------------
stream_endpoints:
  ep0:                                  # Stream endpoint name
    ctrl: True                          # Endpoint passes control traffic
    data: True                          # Endpoint passes data traffic
    num_data_i: 1                       # Number of data input ports
    num_data_o: 2                       # Number of data output ports
    buff_size: 32768                    # Ingress buffer size for data
  ep1:
    ctrl: False
    data: True
    num_data_i: 1
    num_data_o: 1
    buff_size: 32768

# A list of all NoC blocks in design
# ----------------------------------
noc_blocks:
  blk0:                             # NoC block name
```



```
    block_desc: 'blk0_desc.yml'     # Block device descriptor file
    parameters:                     # Optional list of module parameters
      MEM_DEPTH: 64                 # Block-specific module parameters to use
      MASKS: '{8'hE0, 8'h1F}'
  blk1:
     block_desc: 'blk1_desc.yml'

# A list of all static connections in design
# ------------------------------------------
# Format: A list of connection maps (list of key-value pairs) with the following keys
#         - srcblk  = Source block to connect
#         - srcport = Port on the source block to connect
#         - dstblk  = Destination block to connect
#         - dstport = Port on the destination block to connect
connections:
    - {srcblk: blk0,     srcport: out_0,        dstblk: blk1, dstport: din          }
    - {srcblk: blk1,     srcport: dout,         dstblk: ep0,  dstport: in0          }
    - {srcblk: ep1,      srcport: out0,         dstblk: blk0, dstport: in_1         }
    - {srcblk: blk0,     srcport: user_iface_0, dstblk: blk1, dstport: user_iface_0 }
    - {srcblk: _device_, srcport: time_keeper,  dstblk: blk0, dstport: timestamp    }

# A list of all clock domain connections in design
# ------------------------------------------------
# Format: A list of connection maps (list of key-value pairs) with the following keys
#         - srcblk  = Source block to connect (Always "_device"_)
#         - srcport = Clock domain on the source block to connect
#         - dstblk  = Destination block to connect
#         - dstport = Clock domain on the destination block to connect
clk_domains:
    - {srcblk: _device_, srcport: radio,    dstblk: blk1, dstport: user0  }
...
```

5 Index

5.1 Figures

5.2 Tables

