RFNoC 4 Migration Guide

Contents

- 1 Abstract
- 2 Prerequisites
  - 2.1 Dependencies (Ubuntu 20.04 & 22.04)
  - 2.2 Vivado 2021.1 Design Edition
  - 2.3 UHD 4.6
  - 2.4 GNU Radio 3.8
  - 2.5 gr-ettus
- 3 RFNoC Block Development Environment
  - 3.1 Migrating a GNU Radio Out-of-Tree Module
    - 3.1.1 Creating a RFNoC Block with rfnocmodtool
    - 3.1.2 Building OOT module
    - 3.1.3 Running a testbench
    - 3.1.4 Building a FPGA image
  - 3.2 Migrating a Standalone UHD C++ Application
    - 3.2.1 Building rfnoc-example
    - 3.2.2 Running a testbench
    - 3.2.3 Building a FPGA image
- 4 Example RFNoC 3 to RFNoC 4 Block Migration
- 5 UHD Software Migration
  - 5.1 Noc Script XML Replaced by Block Description YAML
  - 5.2 RFNoC API Changes
  - 5.3 Block Properties
    - 5.3.1 RFNoC 3 Noc Script XML snippet
    - 5.3.2 RFNoC 4 Block Controller Class
- 6 FPGA Migration
  - 6.1 Noc Shell Changes
    - 6.1.1 Generating a Custom Noc Shell
    - 6.1.2 Changing Noc ID without using rfnoc_create_verilog
    - 6.1.3 Goodbye AXI Wrapper
  - 6.2 Settings Bus replaced by CtrlPort
  - 6.3 Testbench Infrastructure
  - 6.4 Building FPGA images using Image Core YAML Files
- 7 GNU Radio Software Migration
  - 7.1 RX & TX Streamer Blocks
  - 7.2 Setting RFNoC Block Properties Directly in GNU Radio

The UHD 4.0 release includes a major upgrade to the RFNoC framework called RFNoC 4. This article is a guide to aid users in migrating their existing RFNoC blocks from RFNoC 3 to RFNoC 4. The RFNoC Block Development Environment section provides guidance on how to setup an environment for developing out-of-tree RFNoC blocks in RFNoC 4. The UHD, FPGA, GNU Radio Migration sections provide general information on topics that most users will encounter when migrating their blocks. Finally, an equivalent RFNoC 3 and RFNoC 4 implementation of a digital gain RFNoC Block has been provided as a reference.


```
sudo apt-get install autoconf automake build-essential cmake cpufrequtils doxygen ethtool \
g++ git inetutils-tools libboost-all-dev libusb-1.0-0 libusb-1.0-0-dev libudev-dev \
libspdlog-dev doxygen swig python3-docutils python3-mako python3-numpy python3-requests \
python3-ruamel.yaml python3-setuptools cmake build-essential git g++ libgmp-dev swig \
python3-sphinx python3-lxml doxygen libfftw3-dev libsdl1.2-dev libgsl-dev libqwt-qt5-dev \
libqt5opengl5-dev python3-pyqt5 liblog4cpp5-dev libzmq3-dev python3-yaml python3-click \
python3-click-plugins python3-zmq python3-scipy python3-gi python3-gi-cairo \
gir1.2-gtk-3.0 libcodec2-dev libgsm1-dev pybind11-dev python3-matplotlib libsndfile1-dev \
python3-jsonschema python3-pygccxml libtinfo5 libncurses5
```

Please refer to the dependencies section on the FPGA build page in UHD's manual.

Note 1: Make sure to install AR76780 Patch for Vivado 2021.1

Note 2: The dependencies step above includes installing libtinfo5 libncurses5, which is a workaround for getting Vivado 2021.1 to run on Ubuntu 20.04 & 22.04

```
git clone --branch UHD-4.6 https://github.com/ettusresearch/uhd.git uhd
mkdir uhd/host/build; cd uhd/host/build
cmake ..
make
sudo make install
```

Note: If your design does not use GNU Radio, then installing GNU Radio and gr-ettus is not required

```
git clone --branch maint-3.8 --recursive https://github.com/gnuradio/gnuradio.git gnuradio
mkdir gnuradio/build; cd gnuradio/build;
cmake ..
make
sudo make install
```

You can also reference the GNU Radio Build Instructions for more detailed information.

```
git clone --branch maint-3.8-uhd4.0 https://github.com/ettusresearch/gr-ettus.git gr-ettus
mkdir gr-ettus/build; cd gr-ettus/build;
cmake -DENABLE_QT=True ..
make
sudo make install
```

Two options exist for developing RFNoC blocks depending on whether the your RFNoC block integrates with GNU Radio in an out-of-tree module or if it only uses UHD?s C++ API in a standalone application. The sections below outline how to setup the development environment for each scenario.

The tool rfnocmodtool automates the process of creating GNU Radio out-of-tree (OOT) modules that also have support for RFNoC blocks. This tool is part of gr-ettus and it has been ported to RFNoC 4.

Due to changes in almost every source file, it is recommended to use rfnocmodtool to generate a new RFNoC block from scratch and then update the generated ?skeleton? files.

The following steps show how to create an OOT module called *example* and RFNoC block called *gain* using rfnocmodtool. The naming is only for example purposes.

```
rfnocmodtool newmod
Name of the new module: example

cd rfnoc-tutorial
rfnocmodtool add
Enter name of block/code (without module name prefix): gain
Enter valid argument list, including default arguments: (leave blank)
Add Python QA code? [y/N] N
Add C++ QA code? [y/N] N
Block NoC ID (Hexadecimal): (Enter Noc ID of your block)
Skip Block Controllers Generation? [UHD block ctrl files] [y/N] N
Skip Block interface files Generation? [GRC block ctrl files] [y/N] N
```

Note: Noc IDs have been reduced from 64-bits in RFNoC 3 to 32-bits in RFNoC 4

The following are the relevant files that need to be updated when migrating your RFNoC Block.

```
rfnoc-example/
    grc/
        example_gain.block.yml           ? RFNoC Block GNU Radio Companion YAML file
    examples/
        gain.grc                         ? Example flowgraph using gain RFNoC Block
    include/tutorial/
        gain.h                           ? GNU Radio block C++ header
        gain_block_ctrl.hpp              ? RFNoC Block Controller C++ header
    lib/
        gain_impl.cc                     ? GNU Radio block C++ source
        gain_impl.h                      ? GNU Radio block C++ header
        gain_block_ctrl_impl.cpp         ? RFNoC Block Controller C++ source
    rfnoc/blocks/
        gain.yml                         ? RFNoC Block Description YAML file
    rfnoc/fpga/rfnoc_block_gain
        noc_shell_gain.v                 ? RFNoC Block Noc Shell Verilog Source
        rfnoc_block_gain.v               ? RFNoC Block Verilog Source
        rfnoc_block_gain_tb.v            ? RFNoC Block Testbench
    rfnoc/icores
        gain_x310_rfnoc_image_core.yml   ? Image Core YAML file with gain block


cd rfnoc-tutorial
mkdir build; cd build
cmake -DUHD_FPGA_DIR=(path to uhd/fpga directory) ..
make
sudo make install
```

CMake automatically creates makefile targets to run the generated testbench code for each added RFNoC block. For example, here is how to run the gain block testbench:

```
cd rfnoc-tutorial/build
make rfnoc_block_gain_tb
```

CMake automatically creates makefile targets to build FPGA images using the generated image core yaml files found in rfnoc/icore. Every RFNoC block created by rfnocmodtool automatically has an image core yaml file generated in that directory. For example, here is how to build an FPGA image using the image core yaml file generated for the gain block:

```
cd rfnoc-tutorial/build
make gain_x310_rfnoc_image_core
```

For applications that only use the UHD API, an example out-of-tree (UHD source tree) RFNoC block exists called rfnoc-example. It is located in the UHD source at uhd/host/examples/rfnoc-example. This directory can be copied outside of the UHD source tree and used a starting point to migrate your RFNoC block.

The following are the relevant files that need to be updated when migrating your RFNoC Block.

```
rfnoc-example/
    apps/
        init_gain_block.cpp         ? Example C++ application testing gain block
    blocks/
        gain.yml                    ? RFNoC Block Description YAML file
    fpga/rfnoc_block_gain
        noc_shell_gain.v            ? RFNoC Block Noc Shell Verilog Source
        rfnoc_block_gain.v          ? RFNoC Block Verilog Source
        rfnoc_block_gain_tb.v       ? RFNoC Block Testbench
    icores/
        x310_rfnoc_image_core.yml   ? Example Image Core YAML file
    include/rfnoc/example
        gain_block_control.hpp      ? RFNoC Block Controller C++ header
    lib/
        gain_block_control.cpp      ? RFNoC Block Controller C++ source


cd rfnoc-example
mkdir build; cd build
cmake ..
make
```

```
sudo make install
```

CMake automatically creates makefile targets to run RFNoC Block testbench simulations. For every RFNoC block subdirectory listed in the CMakeLists.txt file in the rfnoc-example/fpga directory, a target with the RFNoC block name appended with ?_tb? is added as a makefile target. For example, here is how to run the gain RFNoC block testbench:

```
cd rfnoc-example/build
make rfnoc_block_gain_tb
```

CMake automatically creates makefile targets to build a FPGA image for each image core yaml file listed in the CMakeLists.txt file in the rfnoc-example/icore directory. Each image core yaml file must be listed in the CMakeLists.txt. For example, here is how to build an FPGA image using the image core yaml file generated for the gain block:

```
cd rfnoc-tutorial/build
make gain_x310_rfnoc_image_core
```

This ZIP archive, File:migration example.zip, contains equivalent RFNoC 3 and RFNoC 4 versions of a digital gain RFNoC Block. The following sections will refer to files in this archive to show how the file structure changes when migrating from RFNoC 3 to RFNoC 4.

Migration reference files for this section from the Gain RFNoC Block example:

Description	RFNoC 3 Files	RFNoC 4 Files
Block Description	rfnoc/blocks/gain.xml	lib/gain_block_ctrl_impl.cpp include/example/gain_block_ctrl.hpp
Block Controller	rfnoc/blocks/gain.yml	lib/gain_block_ctrl_impl.cpp include/example/gain_block_ctrl.hpp

Note: Files are relative to the rfnoc-example directory in the respective rfnoc3 and rfnoc4 directories

RFNoC 3 used Noc Script XML, a domain specific language, to describe the configuration of a RFNoC block: the Noc ID, register names and addresses, args for writing to the registers, and the input/output ports.

RFNoC 4 replaces the Noc Script XML file with an easier to read and edit Block Description YAML file format. From a high level, the Block Description YAML file serves a similar function as the Noc Script XML file, with some similarities and key differences outlined in table below:

Item	Noc Script XML	Block Descript YAML	RFNoC 4 Notes
Block Name	`<name>gain</name>`	`module_name: gain`	
Noc ID	`<id>B160000000000000</id>`	`noc_id: 0xB16`	Noc ID are limited to 32-bits

Registers	`<registers>`
	`  <setreg>`
	`  <name>GAIN</name>`
	`  <address>128</address>`
	`  </setreg>`
	`</registers>`
		N/A	Registers must be defined in the Block Controller

Arguments	`<args>`
	`  <arg>`
	`  <name>gain</name>`
	`  <type>int</type>`
	`  ...`
	`  </arg>`
	`</args>`
		N/A	Args are implemented with properties in the Block Controller

Data Ports	`<ports>`
	`  <sink>`
	`  <name>in</name>`
	`  </sink>`
	`  <name>out</name>`
	`</ports>`
		`data:`
		`  fpga_iface: axis_pyld_ctxt`
		`  clk_domain: rfnoc_chdr`
		`  inputs:`
		`  in:`
		`  ...`
		`  outputs:`
		`  out:`
		`  ...`	

Control Ports	N/A	`control:`
		`  sw_iface: nocscript`
		`  fpga_iface: ctrlport`
		`  interface_direction: slave`
		`  ...`	

Clocking	N/A	`clocks:`
		`  - name: rfnoc_chdr`
		`  freq: "[]"`
		`  - name: rfnoc_ctrl`
		`  freq: "[]"`	

Note: For a more detailed description of the RFNoC 4 Block Description YAML syntax and the various options, see the RFNoC Specification.

Much of the user facing RFNoC software API has not changed or remains very similar between RFNoC 3 and RFNoC 4. The table below outlines some of the notable differences:

RFNoC 3	RFNoC 4	RFNoC 4 Notes
`usrp = uhd::device3::make(...)`	`graph = uhd::rfnoc::rfnoc_graph::make()`	No longer need to create a device3 object
`usrp->get_block_ctrl(...)`	`graph->get_block(...)`	Rename
N/A	`graph->enumerate_static_connections()`	Used to check static connections, for example the DDC and DUC blocks are usually statically connected to the radio block
`usrp->get_tx_streamer(...)`	`graph->create_tx_streamer(...)`	Rename
`usrp->get_rx_streamer(...)`	`graph->create_rx_streamer(...)`	Rename
N/A	`graph->commit()`	Commit graph and run initial checks
`sr_write(...)`	`regs().poke32(...)`	Address increments by 4
`sr_read32(...)`	`regs().peek32(...)`	Address increments by 4

`sr_read64(...)`	`regs().poke64(...)`	Address increments by 8
`set_arg(...)`	`set_property(...)`	Block args replaced with block properties concept
`get_arg(...)`	`get_property(...)`	Block args replaced with block properties concept

In RFNoC 3, RFNoC blocks can have arguments (also known as args) that are used to write user registers. This is implemented in the Noc Script XML in the <args> section.

RFNoC 4 expands and generalizes this concept with block properties: a high-level representation of the state of the block. Zero or more properties can be defined by the user in their RFNoC Block?s Block Controller C++ class. When read or written to, they can trigger a callback to a user defined resolver function. The RFNoC Specification provides more details on properties in the ?Block Properties? section.

The following shows an example of how to migrate a RFNoC 3 Noc Script XML ?arg? based register write to a RFNoC 4 property based implementation in the Block Controller:

```
<registers>
  <setreg>
    <name>GAIN</name>
    <address>128</address>
  </setreg>
</registers>

<args>
  <arg>
    <name>gain</name>
    <type>int</type>
    <value>1</value>
    <check>GE($gain, 0) AND LE($gain, 32767)</check>
    <check_message>Gain must be in the range [0, 32767]</check_message>
    <action>SR_WRITE("GAIN", $gain)</action>
  </arg>
</args>
```

```
// <registers>
//    <setreg>
//      <name>GAIN</name>
//      <address>128</address>
//    </setreg>
// </registers>
// Note: In RFNoC 4, register addresses can start at address 0 instead of address 128 as in RFNoC 3.
const uint32_t gain_block_ctrl::REG_GAIN_ADDR    = 0;
const uint32_t gain_block_ctrl::REG_GAIN_DEFAULT = 1;

class gain_block_ctrl_impl : public gain_block_ctrl
{
public:
    RFNOC_BLOCK_CONSTRUCTOR(gain_block_ctrl)
    {
        _register_props();
    }
private:
    void _register_props()
    {
        register_property(&_user_reg, [this]() {
            int user_reg = this->_user_reg.get();
            // <check>GE($gain, 0) AND LE($gain, 32767)</check>
            // <check_message>Gain must be in the range [0, 32767]</check_message>
            if (user_reg < 0 || user_reg > 32767) {
                throw uhd::value_error("Size value must be in [0,32767]");
            }
            // <action>SR_WRITE("GAIN", $gain)</action>
            this->regs().poke32(REG_USER_ADDR, user_reg);
        });
    }

// <name>gain</name>
// <type>int</type>
// <value>1</value>
property_t<int> _user_reg{"gain", REG_USER_DEFAULT, {res_source_info::USER}};
}
```

As the above shows, writing to a register can be replicated with a property and a resolver function. Of course, the resolver function can also be made much more sophisticated. For additional examples, see the in-tree block controllers in uhd/host/lib/rfnoc.

Migration reference files for this section from the Gain RFNoC Block example:

Description	RFNoC 3 Files	RFNoC 4 Files
Block Verilog Code	rfnoc/fpga-src/noc_block_gain.v	rfnoc/fpga/rfnoc_block_gain/rfnoc_block_gain.v
Block Noc Shell	N/A	rfnoc/fpga/rfnoc_block_gain/noc_shell_gain.v
Block Testbnech	rfnoc/testbench/noc_block_gain/noc_block_gain_tb.sv	rfnoc/fpga/rfnoc_block_gain/rfnoc_block_gain_tb.sv
Image Core	N/A	rfnoc/icores/gain_x310_rfnoc_image_core.yml

Note: Files are relative to the rfnoc-example directory in the respective rfnoc3 and rfnoc4 directories

RFNoC 4 replaces the highly parameterized RFNoC 3 Noc Shell with a per-block customized Noc Shell generated from the block?s Block Description YAML file. The Noc Shell generated via rfnocmodtool or the existing one in rfnoc-example is acceptable for most blocks that require one input and output data port.

Some blocks may need multiple data ports or other modifications. This requires editing the Block Description YAML file and then using the Python script rfnoc_create_verilog.py (found in uhd/host/utils/rfnoc_blocktool) to generate a new Noc Shell instance.

The argument ?-c? is used to provide the YAML file location. ?-d? provides the output destination directory.

Note: It is suggested to not set the destination directory to your existing RFNoC block code, as the script will automatically overwrite the existing code!

Example usage:

```
rfnoc_create_verilog.py -c ./rfnoc-example/rfnoc/blocks/gain.yml -d ./output/
```

In the generated Noc Shell Verilog code, a block?s Noc ID can be changed by updating the NOC_ID parameter on the *backend_iface* module. Make sure this matches the Noc ID in both the Block Description YAML file and Block Controller C++ code.

The RFNoC 3 version of Noc Shell outputs / accepts CHDR data packets consisting of a header, optional timestamp, and payload on a 64-bit AXI stream bus. Most designs then used a module called AXI Wrapper to handle the conversion between CHDR data packets and sample streams on a 32-bit AXI stream bus. AXI Wrapper also supported SIMPLE_MODE which for some use cases could transparently handle the header portion of the CHDR data packet. Otherwise, the user would need to set the header via m_axis_data_tuser.

In RFNoC 4, Noc Shell has absorbed AXI Wrapper?s functionality. Noc Shell outputs two AXI stream buses per input / output port: a payload and context bus. The payload bus is in most cases identical to AXI Wrapper?s output: a 32-bit stream of samples on an AXI Stream bus with packets delimited by tlast. The context AXI stream bus carries the header, optional timestamp, and optional metadata. If your block used AXI Wrapper?s SIMPLE_MODE, then you can loop the context bus back into Noc Shell. If not, you will need to modify the context bus data. Refer to the RFNoC Specification for the format and timing diagram of the context bus.

*Important Note: If your block used the AXI Rate Change module, Noc Shell has another data port mode to support this use case called **axis_data** that can be set in the Block Descriptor YAML file (see the fpga_iface entry). This mode causes the Noc Shell data ports to look more like AXI Wrapper?s and therefore makes them compatible with AXI Rate Change. See the DDC, DUC, or Keep One in N RFNoC Blocks for an example.*

CtrlPort replaces the Settings Bus in RFNoC 4. The CtrlPort bus is similar to the Settings Bus with a few key differences. The table below compares the signaling between the two bus formats and provides notes on any differences. Timing diagrams and additional information on the CtrlPort bus are also available in the RFNoC Specification.

Settings Bus (RFNoC 3)	CtrlPort (RFNoC 4)	RFNoC 4 Notes
set_stb	ctrlport_reg_wr	Write strobe
set_addr	ctrlport_req_addr	20-bits instead of 8-bits, increments by 4 instead of by 1, no reserved addresses (versus addresses 0-127 for Settings Bus)
set_data	ctrlport_req_data	Write data
N/A	ctrlport_req_rd	Read strobe equivalent of ctrlport_req_wr
rb_addr	N/A	CtrlPort uses ctrlport_req_addr for both **read and write** addresses
rb_data	ctrlport_resp_data	Read data, 32-bits instead of 64-bits
rb_stb	N/A	CtrlPort requires ack strobe for **reads and writes**

One additional difference when using CtrlPort is that there is not an equivalent Settings Register module. The bus is simple enough to setup a clocked process to handle reading from and writing to registers. See the Verilog example below:

```
// Note: Register addresses increment by 4
localparam REG_USER_ADDR    = 0; // Address for example user register
localparam REG_USER_DEFAULT = 0; // Default value for user register

reg [31:0] reg_user = REG_USER_DEFAULT;

always @(posedge ctrlport_clk) begin
  if (ctrlport_rst) begin
    reg_user = REG_USER_DEFAULT;
  end else begin
    // Default assignment
    m_ctrlport_resp_ack <= 0;

    // Read user register
    if (m_ctrlport_req_rd) begin // Read request
      case (m_ctrlport_req_addr)
        REG_USER_ADDR: begin
          m_ctrlport_resp_ack  <= 1;
          m_ctrlport_resp_data <= reg_user;
        end
      endcase
    end

    // Write user register
    if (m_ctrlport_req_wr) begin // Write requst
      case (m_ctrlport_req_addr)
        REG_USER_ADDR: begin
          m_ctrlport_resp_ack <= 1;
          reg_user            <= m_ctrlport_req_data[31:0];
        end
      endcase
    end
  end
end
```

*Important Note: For blocks that make heavy use of the Settings Bus and/or Settings Registers, there is a CtrlPort to Settings Bus bridge available called **ctrlport_to_settings_bus**. See the Keep One In N RFNoC Block for example code on how to interface with it.*

While RFNoC 4 does overhaul the RFNoC 3 testbench infrastructure API, most of the high level concepts remain the same. The table below outlines some of the commonly used RFNoC 3 functions / code and the RFNoC 4 equivalent.

Operation	RFNoC 3	RFNoC 4

Setup RFNoC	`RFNOC_SIM_INIT(...)`
	`RFNOC_ADD_BLOCK(...)`
	`RFNOC_CONNECT(...)`
		`RfnocBlockCtrlBfm #(...) blk_ctrl = new(...);`
		`blk_ctrl.connect_master_data_port(...)`
		`blk_ctrl.connect_slave_data_port(...)`
		Note: Instantiate one Block Controller BFM per RFNoC Block

Setup Test Cases	`TEST_CASE_START(...)` `TEST_CASE_DONE(...)`	`test.start_test(...)` `test.end_test()`

Register Read	`tb_streamer.read_reg(...)`	`blk_ctrl.reg_read(...)`
Register Write	`tb_streamer.write_reg(...)`	`blk_ctrl.reg_write(...)`
Send Data / Samples	`tb_streamer.send(...)`	`blk_ctrl.send_items(...)`
Receive Data / Samples	`tb_streamer.recv(...)`	`blk_ctrl.recv_items(...)`

RFNoC 4 replaces uhd_image_builder, the RFNoC 3 FPGA image building tool, with a new tool called rfnoc_image_builder. This tool produces a FPGA bitstream based on an Image Core YAML file that describes the device configuration (e.g. X310 with dual 10GigE) and included RFNoC blocks along with their connections (both static and dynamic), clocking, and I/O.

Both rfnocmodtool and the UHD in-tree example called rfnoc-example automatically setup make targets to handle running rfnoc_image_builder. If you want to use rfnoc_image_builder directly, more details can be found in the Getting Started with RFNoC in UHD 4.0.

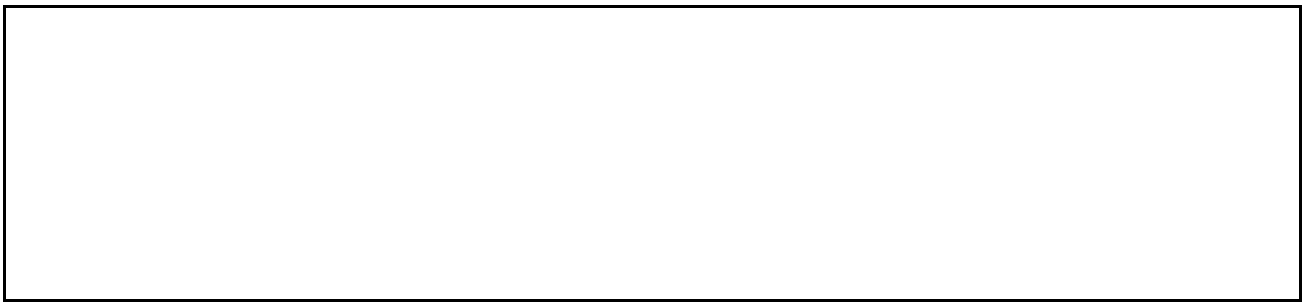
RFNoC 4 supports GNU Radio 3.8 only. Most of your RFNoC Block?s GNU Radio related changes will be due to API differences between GNU Radio 3.7 to 3.8. These changes are outside of the scope of this article. Instead, refer to GNU Radio 3.8 Migration Guide and GNU Radio Companion YAML sites for more information.

Migration reference files for this section from the Gain RFNoC Block example:

Description	RFNoC 3 Files	RFNoC 4 Files
GNU Radio Block	lib/gain_impl.cc lib/gain_impl.h include/example/gain.h	lib/gain_impl.cc lib/gain_impl.h include/example/gain.h
GRC Block Description	grc/gain.xml	grc/gain.yml
Example GRC Flowgraph	examples/gain.grc	examples/gain.grc

Note: Files are relative to the rfnoc-example directory in the respective rfnoc3 and rfnoc4 directories

When transition between a RFNoC block and a GNU Radio block or vice versa, you must insert either a RX stream or TX streamer block respectively. This differs from RFNoC 3, where a RFNoC block could be directly connected to a GNU Radio block.



The base class for RFNoC Block?s in GNU Radio have a set of functions that provide a shortcut to getting and setting properties without writing custom class methods. The table below lists the functions.

Property Type	Set Property	Get Property
Integer	set_int_property(...)	get_int_property(...)
Double	set_double_property(...)	get_double_property(...)
Bool	set_bool_property(...)	get_bool_property(...)
String	set_string_property(...)	get_string_property(...)

Example code for GNU Radio Companion YAML Block Description file

```
templates:
  imports: |-
    import example
  make: |-
    example.gain(
      self.rfnoc_graph,
      uhd.device_addr(${block_args}),
      ${device_select},
      ${instance_select})
    self.${id}.set_int_property('gain', ${gain})
  callbacks:
  - set_int_property('gain', ${gain})
```