Using the RFNoC Replay Block in UHD 4

Contents

AN-642b

This application note guides a user through basic use of the RFNoC Replay block in UHD 4.x and explains how to run the UHD Replay example. This example covers the USRP X410, X310/X300, N300/N310/N320 and E320 devices. For UHD 3.x, please refer to the UHD 3.x replay block Application Note.

Note: The FPGA size on E31x devices is limited, so it is not recommended for use with the Replay block. In order to fit the Replay block on the FPGA, use the E320 or a larger USRP instead.

An introduction to RFNoC with UHD 4.0 and above can be found here.

The Replay block is an RFNoC block that allows recording and playback of arbitrary data using DRAM on the USRP hardware as a buffer. To use the Replay block, it must be instantiated in the design and connected to the DRAM interface.

The replay block is a standard feature of UHD and RFNoC, and a custom compile of UHD is not required. By default, UHD ships the Replay block with the default images for the X410, X310/X300, N300/N310/N320 series of USRPs, so when using these images, the following examples can be run without any manual builds of UHD or FPGA images.

To follow this application note, you need a device with a replay block instantiated, and a UHD version recent enough to have the full support for the replay block (UHD 4.2 and beyond). To test for the replay block capabilities, connect and enable your USRP device, and run the following command:

```
uhd_usrp_probe --args <device args>
```

Insert the appropriate device args for your device, e.g.

```
uhd_usrp_probe --args type=x4xx,addr=192.168.30.2
```

Depending on your device, the output could look like this (truncated):

```
 _____________________________________________________
/
|       Device: X400-Series Device
|     _____________________________________________________
|    /
|    |       Mboard: ni-x4xx-<serial>
|    |   pid: 1040
|    |   rev: 4
|    |   rev_compat: 4
|    |   serial: <serial>
|    |   MPM Version: 4.0
|    |   FPGA Version: 7.6
|    |   FPGA git hash: <hash>.clean
|    |   RFNoC capable: Yes
|    |
|    |   Time sources:  internal, external, qsfp0, gpsdo
|    |   Clock sources: mboard, internal, external, nsync, gpsdo
|    |   Sensors: ...
|    |_____________________________________________________
|    /
|    |       RFNoC blocks on this device:
|    |
|    |   * 0/DDC#0
|    |   * 0/DDC#1
|    |   * 0/DUC#0
|    |   * 0/DUC#1
|    |   * 0/Radio#0
|    |   * 0/Radio#1
|    |   * 0/Replay#0
|    |_____________________________________________________
|    /
|    |       Static connections on this device:
|    |
|    |   * 0/SEP#0:0==>0/DUC#0:0
|    |   * 0/DUC#0:0==>0/Radio#0:0
|    |   * 0/Radio#0:0==>0/DDC#0:0
|    |   * 0/DDC#0:0==>0/SEP#0:0
|    |   * 0/SEP#1:0==>0/DUC#0:1
|    |   * 0/DUC#0:1==>0/Radio#0:1
|    |   * 0/Radio#0:1==>0/DDC#0:1
|    |   * 0/DDC#0:1==>0/SEP#1:0
|    |   * 0/SEP#2:0==>0/DUC#1:0
|    |   * 0/DUC#1:0==>0/Radio#1:0
|    |   * 0/Radio#1:0==>0/DDC#1:0
|    |   * 0/DDC#1:0==>0/SEP#2:0
```

```
|   |   * 0/SEP#3:0==>0/DUC#1:1
|   |   * 0/DUC#1:1==>0/Radio#1:1
|   |   * 0/Radio#1:1==>0/DDC#1:1
|   |   * 0/DDC#1:1==>0/SEP#3:0
|   |   * 0/SEP#4:0==>0/Replay#0:0
|   |   * 0/Replay#0:0==>0/SEP#4:0
|   |   * 0/SEP#5:0==>0/Replay#0:1
|   |   * 0/Replay#0:1==>0/SEP#5:0
|   |   * 0/SEP#6:0==>0/Replay#0:2
|   |   * 0/Replay#0:2==>0/SEP#6:0
|   |   * 0/SEP#7:0==>0/Replay#0:3
|   |   * 0/Replay#0:3==>0/SEP#7:0
```

The output tells us that this USRP has a Replay block instantiated (`0/Replay#0`). It has four static connections to stream endpoints, which also tells us that this is a four-port replay block.

If your device does not report a replay block, then you need to load an FPGA image which includes this block. See this section for instructions on how to do this before you proceed. If it does report a block, you can move to the next section.

The `rfnoc_replay_samples_from_file` example assumes that you have a file containing the samples you wish to replay. This could be generated in advance or recorded using `rx_samples_to_file` or another method. For this demonstration, we'll create a simple Python program (`sample_gen.py`) to generate some samples to use:

```
import math
import struct

SAMPLE_RATE = 200.0e6        # Sample rate in Hz
FREQUENCY   = 500.0e3        # Frequency of sinusoid to generate, in Hz
NUM_SAMPLES = 16000          # Number of samples to generate
AMPLITUDE   = 0.5            # Amplitude of the signal (from 0 to 1.0)
FILE_NAME   = 'samples.dat'

file = open(FILE_NAME, 'wb')

for i in range(NUM_SAMPLES):
    I = int((2**15-1) * AMPLITUDE * math.cos(i / (SAMPLE_RATE / FREQUENCY) * 2 * math.pi))
    Q = int((2**15-1) * AMPLITUDE * math.sin(i / (SAMPLE_RATE / FREQUENCY) * 2 * math.pi))
    file.write(struct.pack('<2h', I, Q))

file.close()
```

This program generates a file named `samples.dat` that contains 16000 samples (40 periods) of a 500 kHz tone sampled at a rate of 200 MHz. Each sample is saved in `sc16` format (signed complex with 16-bit real and 16-bit imaginary components). We can run the program by invoking python from the command line.

```
$ python ./sample_gen.py
```

To run the UHD Replay example, enter a command like the following (the path to the examples depends on your installation method, for a normal installation via apt-get, examples will be located in `/usr/lib/uhd/examples` or `/usr/local/lib/uhd/examples`).

```
$ cd /path/to/examples
$ ./replay_samples_from_file --args <device args> --freq 915e6 --gain 10 --file samples.dat --rate 200e6
```

This example would stream the samples from the file to the Replay block on the FPGA, where they are recorded into the USRP's on-board DRAM. Then, the Replay block will play the samples to the radio continuously with a base frequency of 915 MHz, creating a tone at 915.5 MHz. Press `Ctrl+C` to stop transmitting. Alternatively, use the `--nsamps` command line argument to transmit a certain number of samples before returning to the command line. Use the `--help` argument to see a full list of arguments.

The advantage of this example compared to directly streaming the file to the device is twofold:

- The initial upload to DRAM can happen at any link rate. Even when using 1 GbE, this example will work.
- Once uploaded, the host computer is basically idle. The FPGA will handle the streaming to the radio front-end.

The source code for `rfnoc_replay_samples_from_file` may be considered an example for best practices on how to use the replay block.

This block works like a record and playback buffer that uses DRAM on the USRP to store samples. Data can be streamed to the block, like to any other RFNoC block.

Refer the manual of the replay block controller for a comprehensive description of its features and API calls.

In the following, we shall use the C++ API to demonstrate the most important API calls. We will assume there is a replay block controller called `replay_ctrl` available in the current context, and it is of type `uhd::rfnoc::replay_block_control::sptr`.

Before streaming data to the replay block, it needs to be configured for recording:

```
replay_ctrl->record(buffer_start_byte_address, buffer_size_in_bytes, replay_chan);
```

This tells the Replay block that it should start recording any data it receives on port `port` into the DRAM at byte offset `buffer_start_byte_address` and should use up to `buffer_size_in_bytes` bytes. Once the buffer is filled, recording automatically stops. Care should be taken to configure the memory buffers so that they do not overlap if more than one Replay block or buffer is being used simultaneously.

Call this once for every port that you expect to stream data into.

Note: Care should be taken to not transfer more data to the Replay block than the size of the record buffer. Additional data is not accepted or dropped by the replay block, but flow control will cause data to back up in the RF network on the FPGA.

Note: The amount of memory available to the Replay block is limited by the size of the DRAM on the USRP and how the memory interface is configured on the USRP. By default, we expose the entire memory of the device, but it is possible to modify `axi_intercon_2x64_128_bd` if less memory should be made accessible.

Devices have the following amount of memory:

E310	512 MiB

E320 2 GiB
N3xx 2 GiB
X310 1 GiB
X410 4 GiB per bank (note: not all banks may be connected)
The currently available memory can be queried using the block controller and the `get_mem_size()` API call.

To restart recording from the same offset, the following API call can be used:

```
replay_ctrl->record_restart(replay_chan);
```

This resets the record pointer to point back to the beginning of the buffer and it resets the internal counters that track how much data has been recorded. If a previous recording has taken place then it is a good idea to ensure that stale data was not queued up in the RF network on the FPGA from a previous run. The `replay_samples_from_file` example does this by calling `record_restart()` then waiting to see if any new data shows up unexpectedly in the record buffer. If so, it restarts recording then waits again to see if data continues to appear.

You can determine when all data has been received by checking the status of the record fullness.

```
// Wait for recording to complete
while (replay_ctrl->get_record_fullness(replay_chan) < num_bytes_expected)
    std::this_thread::sleep_for(100ms);
```

Prior to playing back recorded data, it is necessary to configure the base address and size of the playback buffer. To play back previously recorded data, set the start address to the same address that was used for the record buffer and set the size of the playback buffer to the match the amount of data that was recorded. Note that the record and playback buffers do not need to be the same, allowing a single Replay block to both record and playback to different regions of memory simultaneously.

```
// Configure the Replay block to play back everything that was recorded
num_bytes_recorded = replay_ctrl->get_record_fullness(replay_chan);
replay_ctrl->config_play(buffer_start_byte_address, num_bytes_recorded, replay_chan);
```

To play back the data in the playback buffer, issue the appropriate UHD stream command. Playback automatically wraps around to the start of the buffer if more data is requested than the size of the playback buffer.

```
uhd::stream_cmd_t stream_cmd(uhd::stream_cmd_t::STREAM_MODE_START_CONTINUOUS);
stream_cmd.stream_now = true;
replay_ctrl->issue_stream_cmd(stream_cmd, replay_chan);
```

or

```
uhd::stream_cmd_t stream_cmd(uhd::stream_cmd_t::STREAM_MODE_NUM_SAMPS_AND_DONE);
stream_cmd.num_samps  = words_to_replay;
stream_cmd.stream_now = true;
replay_ctrl->issue_stream_cmd(stream_cmd, replay_chan);
```

`STREAM_MODE_START_CONTINUOUS` causes playback to continue indefinitely until explicitly stopped. `STREAM_MODE_NUM_SAMPS_AND_DONE` causes only the specified number of samples to be played once. Playback can be stopped by issuing

```
a stop command.

stream_cmd.stream_mode = uhd::stream_cmd_t::STREAM_MODE_STOP_CONTINUOUS;
replay_ctrl->issue_stream_cmd(stream_cmd);
```

This will stop playback at the end of the next DRAM read after the command is received (DRAM reads are not aborted mid-transaction). As a result, some data will continue to stream from the Replay block after the stop command is issued while waiting for the DRAM read to complete and for all the internal buffers to empty.

When using the C++ API, the `play()` API call is a useful shorthand for configuring playback and submitting the stream command at the same time:

```
replay_ctrl->play(buffer_start_byte_address, num_bytes_to_play, replay_chan, start_time, repeat);
```

In either case, the start time is optional. When given, the first sample to leave the block on playback is tagged with this timestamp.

There are two memory alignment values that need to be considered when dealing with the replay block. The first is the word size, which is the minimum number of bytes per DRAM transaction. For most configurations, the word size is 64 bits, which means that only even numbers of samples can be recorded or played back when using 16-bit complex samples (at 4 bytes per sample). Use the `get_word_size()` API call to identify the correct word size. The word size can go up to 512 bits.

The second alignment value is the memory's page boundaries. The start addresses for record and replay should fall onto a 4 kiB memory boundary to ensure correct alignment of data.

Before you begin, make sure you are using the `Bash` shell. See Reconfigure Default Shell in AN-315 for detailed instructions.

Note: Cloning the repository is only required when building custom FPGA images. If the replay block is already built into the USRP's bit file, this is not required. By default, UHD ships the Replay block with the default images for the X410, X310/X300, N300/N310/N320 series of USRPs.

If you do require access to the source code, e.g. to build an FPGA image with a custom replay block configuration, run the following command:

```
$ git clone https://github.com/EttusResearch/uhd.git
```

For the rest of the Application Note, we assume the repository was cloned into the location `~/src/uhd`.

In order to build the FPGA image for the intended USRP product, you will need to have the Xilinx development tools installed. The specific version required depends on the UHD version. Refer to the manual for the correct version for your UHD version, and the installation instructions for Vivado in order to install these tools. It is recommended that you use the default install location of `/opt/Xilinx/Vivado`.

Note: Most bitfiles already include the replay block! This section is only relevant if you want to build your own bitfile, or the bitfile you're using does not contain the replay block already.

In order to use the Replay block, it must be built into the FPGA image for the USRP you plan to use. This is currently a manual step. The instructions below are for the X310, but similar instructions apply to other RFNoC-capable devices.

To create a custom FPGA image with a replay, you need to create an image core file. The following lines are the relevant lines from the X310 default image core file:

```
  stream_endpoints:
    # ... all the other stream endpoints...
    ep4:                        # Stream endpoint name
      ctrl: False                       # Endpoint passes control traffic
      data: True                        # Endpoint passes data traffic
      buff_size: 4096                   # Ingress buffer size for data
    ep5:                        # Stream endpoint name
      ctrl: False                       # Endpoint passes control traffic
      data: True                        # Endpoint passes data traffic
      buff_size: 4096                   # Ingress buffer size for data

  noc_blocks:
    # ... all the other blocks...
    replay0:
      block_desc: 'replay.yml'
      parameters:
        NUM_PORTS: 2
        MEM_ADDR_W: 30

  connections:
    # ...connections for all the other blocks...
    # ep4 to replay0(0)
    - { srcblk: ep4,     srcport: out0,  dstblk: replay0, dstport: in_0 }
    # replay0(0) to ep4
    - { srcblk: replay0, srcport: out_0, dstblk: ep4,     dstport: in0  }
    # ep5 to replay0(1)
    - { srcblk: ep5,     srcport: out0,  dstblk: replay0, dstport: in_1 }
    # replay0(1) to ep5
    - { srcblk: replay0, srcport: out_1, dstblk: ep5,     dstport: in0  }
    # BSP Connections
    - { srcblk: replay0, srcport: axi_ram, dstblk: _device_, dstport: dram }
  clk_domains:
    # ...all other clock domains...
    - { srcblk: _device_, srcport: dram,  dstblk: replay0, dstport: mem  }
```

As you can see, the replay block requires configuration in up to four sections:

- For maximum flexibility, every port of the replay block will receive its own stream endpoint. This is not a requirement of the replay block, but allows its flexible use.
- Of course, the block needs to be declared in the `noc_blocks` section. The blocks contain two parameters, NUM_PORTS, and MEM_ADDR_W. The former is the number of ports this RFNoC block may have. The latter is the width of the memory address word, i.e., it is log2(memory_size).
- It must be connected to the stream endpoints in the `connections` section, as well as to the DRAM banks (BSP connection).
- Finally, the clock domain needs to be connected.

All devices have limits regarding the number of ports and the memory size. The following values may not be exceeded:

```
E310:
    NUM_PORTS: 2
    MEM_ADDR_W: 29
E320:
    NUM_PORTS: 4
    MEM_ADDR_W: 31
N3xx:
    NUM_PORTS: 4
    MEM_ADDR_W: 31
X310:
    NUM_PORTS: 2
    MEM_ADDR_W: 30
X410:
    NUM_PORTS: 4
    MEM_ADDR_W: 32
```

When the YAML image core file is complete, save it, e.g., as x310_replay_image_core.yml, and pass it to the image builder:

```
  rfnoc_image_builder -y x310_replay_image_core.yml [ --fpga-dir ~/usr/uhd/fpga ]
```

Note: Because the DRAM does not fit by default on the E31x devices, it is not included in the build by default. To include the DRAM on E31x builds, set the environment variable `DRAM=1` before building (e.g., `DRAM=1 rfnoc_image_builder ...`). The E320 is the recommended E-series device for use with the Replay block.

This will create a bitfile that contains the replay block. It can be loaded onto the device using the `uhd_image_loader` tool:

```
  uhd_image_loader --args type=x300,addr=<ip address> --fpga-path=/path/to/usrp_x310_fpga_HG.bit
```

When this is complete, the replay block is ready to use.

For reference, the following files implement the replay block:

- Verilog/HDL sources: https://github.com/EttusResearch/uhd/tree/master/fpga/usrp3/lib/rfnoc/blocks/rfnoc_block_replay
- C++ Block controller sources (header, block controller, Python bindings, unit tests):
  - https://github.com/EttusResearch/uhd/blob/master/host/include/uhd/rfnoc/replay_block_control.hpp
  - https://github.com/EttusResearch/uhd/blob/master/host/lib/rfnoc/replay_block_control.cpp
  - https://github.com/EttusResearch/uhd/blob/master/host/lib/rfnoc/replay_block_control_python.hpp
  - https://github.com/EttusResearch/uhd/blob/master/host/tests/rfnoc_block_tests/replay_block_test.cpp
- Example: https://github.com/EttusResearch/uhd/blob/master/host/examples/rfnoc_replay_samples_from_file.cpp