Using the RFNoC Replay Block

Contents

AN-642

This application note guides a user through basic use of the RFNoC Replay block in UHD 3.x and explains how to run the UHD Replay example. This example covers use on the X300/X310 and N310 products. UHD 4.0 and later is covered in Using the RFNoC Replay Block in UHD 4.

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. It can take the place of the DMA FIFO(s) or be used concert with the DMA FIFO(s). In this note we will be replacing the DMA FIFO block with the Replay block and running a UHD example that records data to DRAM from a file then plays it back over the radio continuously.

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

Your system must be configured for RFNoC development to compile and use the RFNoC examples. Here we briefly explain how to setup a system to build and run the RFNoC Replay example.

Note: Refer to Application Note AN-823 Getting Started with RFNoC Development for a more detailed overview of RFNoC development.

To begin, use the following `git clone` command to download the needed UHD repository. The `--recursive` option causes the latest compatible FPGA code to also be cloned into the `fpga-src` subfolder.

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

Then checkout the appropriate version of UHD that you intend to use. Replay block support was added in UHD 3.14. The latest UHD 3.x version is recommended.

```
$ git checkout UHD-3.15.LTS
$ git submodule update --recursive
```

If you have not already done so, follow the steps in Application Note **AN-445** under the heading Update and Install dependencies.

Note: Refer to Application Note AN-445 for detailed instructions on building and installing UHD from the source code. However, RFNoC must be enabled when running CMake in order to run the RFNoC examples. The instructions below summarize the basic steps required to build and install UHD so that you can run the Replay example.

To build and install UHD, begin by opening a terminal in the UHD repository that you cloned, then create a `build` folder within the `host` host folder of the repository.

```
$ cd uhd/host
$ mkdir build
$ cd build
```

Run CMake with RFNoC enabled to create the Makefiles.

```
$ cmake -DENABLE_RFNOC=ON ../
```

Run Make to build UHD with RFNoC support.

```
$ make
```

Install UHD, using the default install prefix, which will install UHD under the `/usr/local/lib` folder. You need to run this as root due to the permissions on that folder.

```
$ sudo make install
```

Update the system's shared library cache.

```
$ sudo ldconfig
```

Make sure that the `LD_LIBRARY_PATH` environment variable is defined and includes the folder under which UHD was installed. Most commonly, you can add the line below to the end of your `$HOME/.bashrc` file.

Note: the `LD_LIBRARY_PATH` location may vary depending on your Linux distribution.

```
$ export LD_LIBRARY_PATH=/usr/local/lib
```

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 branch and state of the FPGA code. The UHD-3.13 branches require Vivado 17.4. Refer to 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` to ensure compatibility with the FPGA build flow.

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 the N310.

First, we must modify the Verilog code to include the Replay Block. To do this, modify the file `fpga-src/top/x300/x300_core.v` and change localparam `USE_REPLAY` from 0 to 1. This causes the FPGA code to instantiate `noc_block_replay` instead of `noc_block_axi_dma_fifo`. Note that the DMA FIFO will not be included in this example and therefore cannot be used.

Note: If using the N310, modify the file `fpga-src/top/n3xx/n3xx_core.v` and make the same change. Other products that support RFNoC can also use the replay block. However, in other products, the noc_block_replay instance would need to be manually instantiated in the code following the examples given in the `x300_core.v` and `n3xx_core.v` files.

After making the required code change, you are ready to rebuild the FPGA image. Begin by setting up the environment to use the FPGA build tools.

```
$ cd uhd/fpga-src/usrp3/top/x300
$ source ./setup.sh
```

Run make to build the desired FPGA image. For example, to build the X310 HG image, use the following command:

```
$ make X310_HG
```

Once compilation is complete, download the image to your USRP product. For example, if the X310 HG image were connected to SFP port 0 (1 Gigabit Ethernet) using the default IP address, then you would run the following command.

```
$ uhd_image_loader --args="type=x300,addr=192.168.10.2" --fpga-path=./build-X310_HG/x300.bit
```

After the download has completed, power cycle the X310 to load the new bitstream. Confirm that the Replay block appears in the system by running `uhd_usrp_probe`.

```
$ uhd_usrp_probe --args="addr=192.168.10.2"
```

You should see the `Replay` block listed among the RFNoC blocks on the device.

```
|   |     _____________________________________________________
|   |    /
|   |   |       RFNoC blocks on this device:
|   |   |
|   |   |   * Replay_0
|   |   |   * Radio_0
|   |   |   * Radio_1
|   |   |   * DDC_0
|   |   |   * DDC_1
|   |   |   * DUC_0
|   |   |   * DUC_1
```

In this section we will compile the replay_from_file UHD example. Begin by creating a CMake file for the Replay example using `uhd/host/examples/init_usrp/CMakeLists.txt` as an example.

```
$ cd uhd/host/examples
$ mkdir replay_samples_from_file
$ cd replay_samples_from_file
$ cp ../init_usrp/CMakeLists.txt ./
```

Edit `CMakeLists.txt` and change the `init_usrp` references to `replay_samples_from_file` and `init_usrp.cpp` to `../replay_samples_from_file.cpp`.

```
### Make the executable ##########################################################
add_executable(init_usrp init_usrp.cpp)

SET(CMAKE_BUILD_TYPE "Release")
MESSAGE(STATUS "*********************************************************************************")
MESSAGE(STATUS "* NOTE: When building your own app, you probably need all kinds of different  ")
MESSAGE(STATUS "* compiler flags. This is just an example, so it's unlikely these settings    ")
MESSAGE(STATUS "* exactly match what you require. Make sure to double-check compiler and     ")
MESSAGE(STATUS "* linker flags to make sure your specific requirements are included.         ")
MESSAGE(STATUS "*********************************************************************************")

# Shared library case: All we need to do is link against the library, and
# anything else we need (in this case, some Boost libraries):
if(NOT UHD_USE_STATIC_LIBS)
    message(STATUS "Linking against shared UHD library.")
    target_link_libraries(init_usrp ${UHD_LIBRARIES} ${Boost_LIBRARIES})
# Shared library case: All we need to do is link against the library, and
# anything else we need (in this case, some Boost libraries):
else(NOT UHD_USE_STATIC_LIBS)
    message(STATUS "Linking against static UHD library.")
    target_link_libraries(init_usrp
        # We could use ${UHD_LIBRARIES}, but linking requires some extra flags,
        # so we use this convenience variable provided to us
        ${UHD_STATIC_LIB_LINK_FLAG}
        # Also, when linking statically, we need to pull in all the deps for
        # UHD as well, because the dependencies don't get resolved automatically
        ${UHD_STATIC_LIB_DEPS}
    )
endif(NOT UHD_USE_STATIC_LIBS)

### Once it's built... ###########################################################
# Here, you would have commands to install your program.
# We will skip these in this example.
```

You can now invoke CMake and run Make to build the example.

```
$ mkdir build
$ cd build
$ cmake ../
$ make
```

The `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
```

Note: The `replay_samples_from_file` example does not perform rate conversion (i.e., the the DUC is not used), so the rate specified must match the native sample rate of your device (i.e., 200 Msps for the X300/X310 or 125 Msps for the N310). The samples file should contain `sc16` data samples and

should be a multiple of 2 samples (8 bytes) in size, since the Replay block records and plays back in multiples of 8 bytes. For example, for the N310 you could change `SAMPLE_RATE` in the Python program to 125.0e6, which would result in 64 periods of the 500 kHz tone.

To run the UHD Replay example, you could enter a command like the following.

```
$ ./replay_samples_from_file --freq 915e6 --gain 10 --file samples.dat
```

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 would cause Replay block to play back 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.

The Replay block is contained in `noc_block_replay.v`. This block works like a record and playback buffer that uses DRAM on the USRP to store samples. It connects to the RFNoC crossbar and to the DRAM in the same way that the `noc_block_axi_dma_fifo` block does. Data can be streamed to the block, like to any other RFNoC block. Playback is analogous to the way the `noc_block_radio_core` works when we ask it to receive radio samples.

One key difference is that the Replay block works only with 64-bit samples. Therefore, all addresses, buffer sizes, and transfers should be a multiple of 8 bytes. For example, when using `sc16` samples (4 bytes each) everything should be a multiple of two samples to ensure we are always working with multiples of 8 bytes.

Note: Refer to the example source code in `replay_samples_from_file.cpp` for a more detailed example of how to use the Replay block.

Prior to streaming data to the Replay block for recording, it is necessary to configure the base address and size of the record buffer.

```
// Configure the record buffer
replay_ctrl->config_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 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. The memory addresses and sizes should be 8-byte aligned.

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. For example, the `axi_intercon_2x64_128_bd` IP used by the X310 is configured to give each connected device an address space of 32 MiB. As a result, the Replay block will be limited to this amount of memory. The `axi_intercon_2x64_128_bd` file must be modified if more than 32 MiB needs to be buffered.

To begin recording data to the Replay block, the record logic should be initialized:

```
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);
```

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 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.num_samps  = words_to_replay;
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. The `num_samps` parameter is a 28-bit value, limiting this mode of playback to 2^{28} words at a time. 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. The `replay_samples_from_file` example determines when playback streaming has stopped by reading the 64-bit `SR_READBACK_REG_GLOBAL_PARAMS` register and waiting for the packet count to stop increasing.