RFSoC DFE DUC-DDC Mixer v1.0

LogiCORE IP Product Guide

Vivado Design Suite

PG393 (Early Access Draft) December 15, 2021

Table of Contents

Chapter 1

Introduction

The Xilinx® Zynq UltraScale+ RFSoC DFE Adaptable SoCs contain a variety of dedicated custom signal processing primitives, which address the challenges of high-speed, low-power digital front-end design.

The RFSoC DFE DUC-DDC Mixer LogiCORE™ IP is a high-performance configurable digital down-conversion and digital up-conversion block, with a multi-carrier mixing and extraction stage. It supports mixed sample rates and multiple up/down conversion factors.

Features

- Supports up to eight component carriers across up to eight antennas.
- Interpolation and decimation factors up to 16x.
- CORDIC-based dual-modulus NCOs for frequency-error-free operation.
- 16-bit or 18-bit I+Q complex data input and output, AXI4-Stream interfaces.
- Flexible time-division multiplexing for multiple component carriers and antennas.
- Latency-compensated sideband channel (TUSER) for sample-accurate framing.
- Triggered parameter-change operations and buffer flushing.
- Power-down and dynamic power saving modes available.
- AXI4-Lite memory-mapped control and status interface with software API provided.

IP Facts

LogiCORE™ IP Facts Table	
Core Specifics	
Supported Device Family[1]	Zynq® UltraScale+™ RFSoC DFE
Supported User Interfaces	AXI4-Stream
Resources	N/A
Provided with Core	
Design Files	Encrypted RTL
Example Design	Verilog (simulation only)
Test Bench	Provided with the example design
Constraints File	Not Provided
Simulation Model	Encrypted Verilog C Model
Supported S/W Driver[2]	Standalone and Linux
Tested Design Flows[3]	
Design Entry	Vivado® Design Suite
Simulation	For supported simulators, see the Xilinx Design Tools: Release Notes Guide.
Synthesis	Vivado® Synthesis
Support	
All Vivado IP Change Logs	Master Vivado IP Change Logs: 72775
Xilinx Support web page	

Notes:

1. For a complete list of supported devices, see the Vivado® IP catalog.
2. Standalone driver details can be found in <install_directory>/Vitis/<release>/data/embeddedsw/doc/xilinx_drivers_api_toc.htm.
3. For the supported versions of third-party tools, see the Xilinx Design Tools: Release Notes Guide.

Chapter 2

Overview

Navigating Content by Design Process

Xilinx® documentation is organized around a set of standard design processes to help you find relevant content for your current development task. All Versal® ACAP design process Design Hubs and the Design Flow Assistant materials can be found on the Xilinx.com website. This document covers the following design processes:

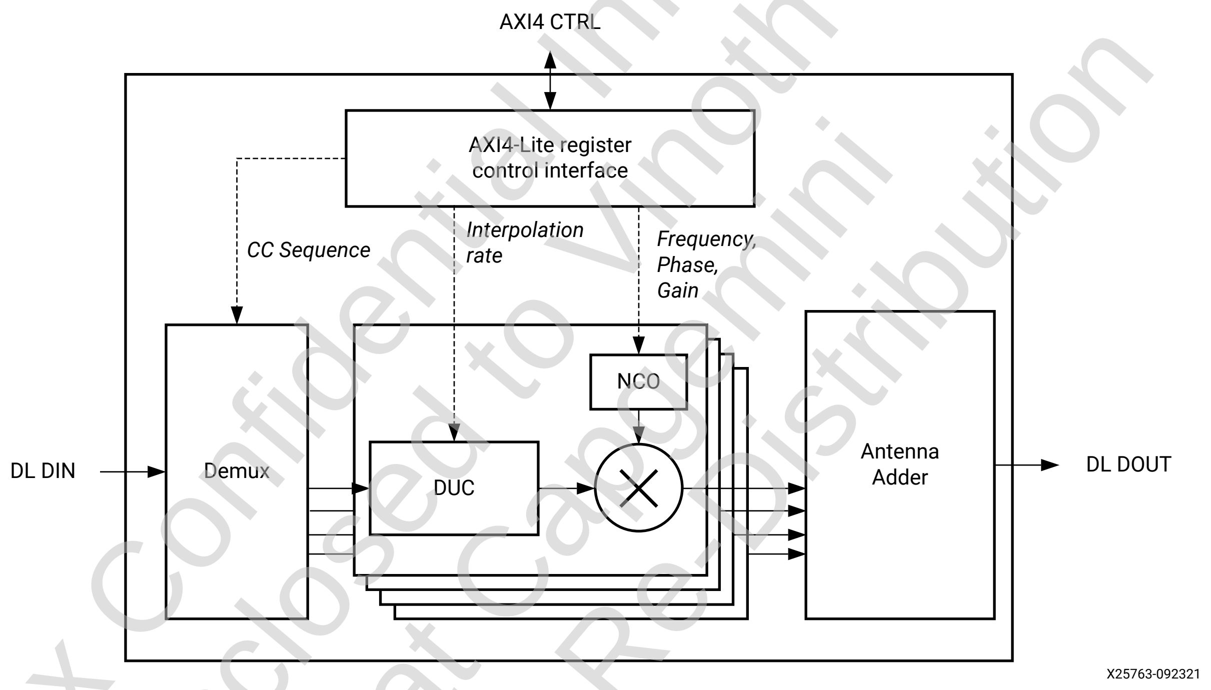
- **Embedded Software Development:** Creating the software platform from the hardware platform and developing the application code using the embedded CPU. Also covers XRT and Graph APIs. Topics in this document that apply to this design process include:
  - Configuring and Controlling the Core
  - Chapter 6: Example Design
  - Appendix B: API Reference
- **Hardware, IP, and Platform Development:** Creating the PL IP blocks for the hardware platform, creating PL kernels, functional simulation, and evaluating the Vivado® timing, resource use, and power closure. Also involves developing the hardware platform for system integration. Topics in this document that apply to this design process include:
  - Port Descriptions
  - Clocking
  - Resets
  - Interfacing to the Core
  - Chapter 5: Design Flow Steps

Core Overview

The RFSoC DFE DUC-DDC Mixer core provides up-conversion and multi-carrier mixing for transmitter applications, and carrier extraction and down-conversion for receiver applications. Up to eight parallel antenna streams can be supported in the uplink and downlink directions.

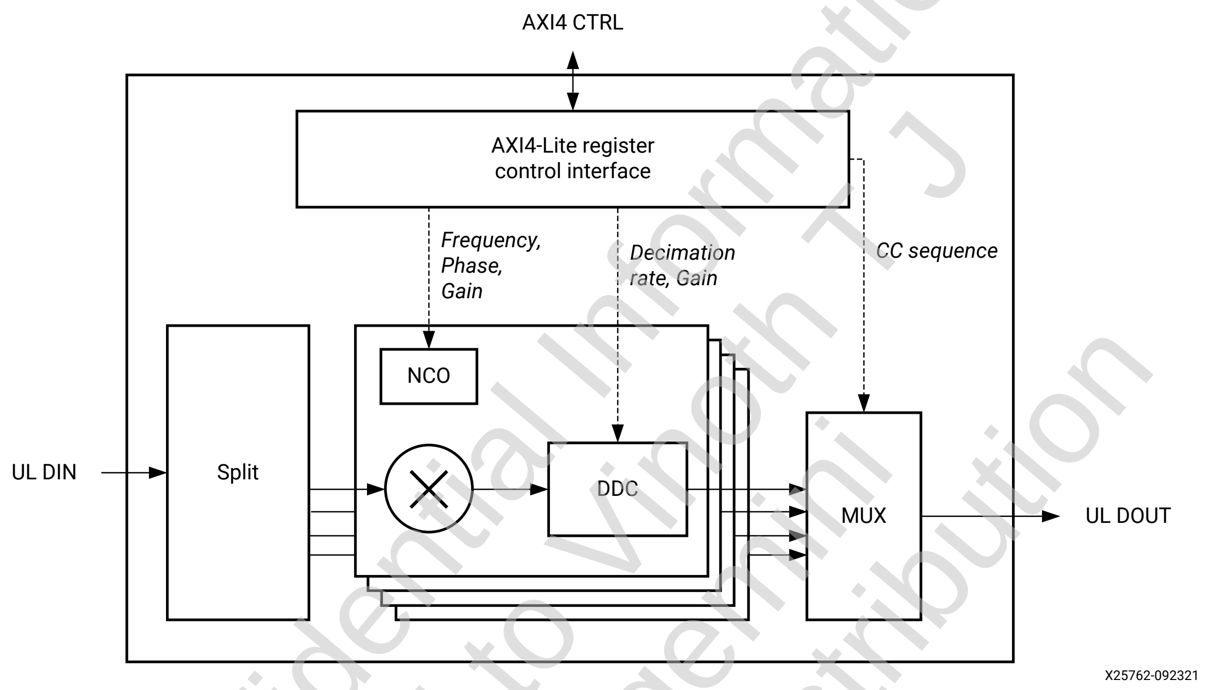
When configured in downlink mode, the up-converter (DUC) performs a sample rate change from the baseband rate of each component carrier to the common sample rate of the combined transmit signal, using a configurable chain of symmetric half-band 2x interpolation filters. Interpolation factors of 1x, 2x, 4x, 8x, and 16x are possible. The up-converted streams are then mixed to the required carrier frequencies using highly accurate CORDIC-based dual-modulus NCOs. Finally the carriers are combined into a single transmit signal. An overview of the core functional structure in downlink mode is shown in the following figure.

Figure 1: **Core Structure in Downlink Mode**



In the receive direction, the CORDIC-based NCOs are used to perform carrier extraction. Data for the individual component carriers is then passed to the down-converter (DDC), which performs a sample rate change from the sample rate of the received signal to the baseband rate of each component carrier, using a configurable chain of symmetric half-band 2x decimation filters. Decimation factors of 1x, 2x, 4x, 8x and 16x are possible. An overview of the core functional structure in uplink mode is shown in the following figure.

Figure 2: **Core Structure in Uplink Mode**



Standard AXI4-Stream interfaces are used for data input and output. The width of the data buses can be configured as either 16-bit or 18-bit according to the application.

Note: For 16-bit output, additional rounding is performed.

Each sample comprises I and Q components for 32 or 36 bits per complex sample. According to a multiplexing sequence, samples from multiple component carriers and antennas are interleaved on the input and output data buses, which are programmable dynamically.

Each AXI4-Stream interface has associated `TUSER` and `TLAST` signals for user-defined framing/timing control purposes. The core provides a delay-matching feature to align this out-of-band information throughout the signal processing chain.

The core has an AXI4-Lite memory-mapped interface for configuration, control and status readback. An optional IRQ output is also available to enable the processor to interrupt error conditions. A software driver is included, which provides all the API functions required to set up and operate the RFSoC DFE DUC-DDC Mixer core.

The API can instruct changes to the carrier frequencies, mixer parameters, gain factors and carrier interleaving sequence, for enabling/disabling of carriers and antennas, and for entry into and exit from low power modes. These configuration updates can all be scheduled to take place according to a chosen trigger event.

In all cases where triggering of control changes are possible, this can be either immediate (on register write) or based on arbitrary transitions on selected bits of the TUSER input bus. TUSER-based triggering allows updates to be synchronized precisely to a specific sample, for example, to coincide with a radio slot boundary or other periodic events. This is recommended in most cases and is required when updating NCO settings and component carrier sequencing patterns to ensure that phase continuity is maintained.

Applications

The RFSoC DFE DUC-DDC Mixer core can implement the up-conversion, down-conversion, and carrier extraction/mixing stages of a communication system, including those based on the 3GPP LTE and the 3GPP 5G standard. The core is a component of the Xilinx® RFSoC DFE targeted reference design.

Licensing and Ordering

This Xilinx® LogiCORE™ IP module is provided at no additional cost in Vivado® for customers with a valid license for the RFSoC DFE family of Adaptable SoCs.

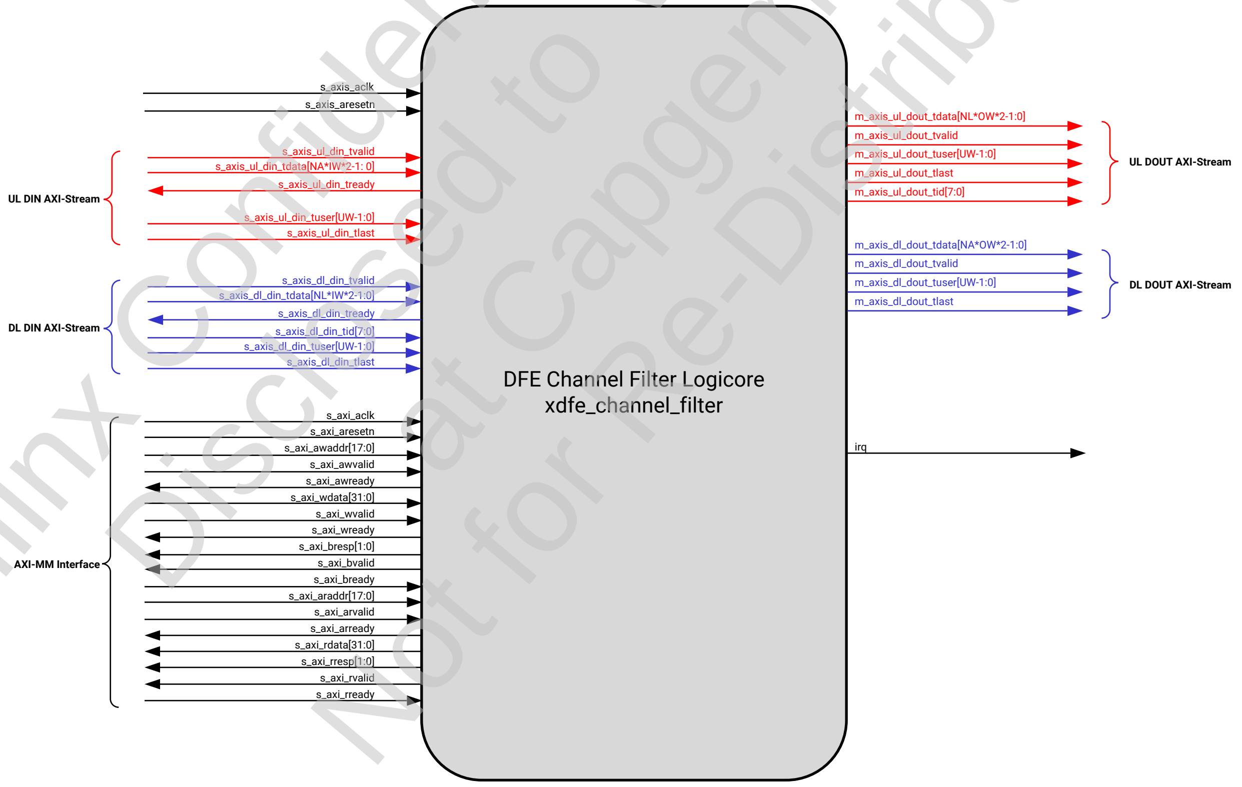
Information about other Xilinx® LogiCORE™ IP modules is available at the Xilinx Intellectual Property page. For information about pricing and availability of other Xilinx LogiCORE IP modules and tools, contact your local Xilinx sales representative.

Product Specification

Port Descriptions

The core interfaces are shown in the following figure.

Figure 3: **Core Ports**



Either the downlink interfaces highlighted in blue or the uplink interfaces highlighted in red can be present, but not both.

The widths of the data buses within the UL DIN and DL DOUT AXI4-Stream interfaces are determined by the input and output sample widths, abbreviated to IW and OW, and the number of antennas denoted as NA. The width is given by NA×IW×2 or NA×OW×2. One complex sample pair per antenna is transferred on every CPS clock cycles, where CPS is the number of mixer clocks per cycle that was chosen when the core was customized. CPS can be 1, 2 or 4. .

The widths of the data buses within the DL DIN and UL DOUT AXI4-Stream interfaces are determined by the input or output sample width, the number of antennas, and the antenna interleaving factor that was chosen when the core was customized. These interfaces carry multiplexed data for multiple component carriers and antennas. The width is given by NL×IW×2 or NL×OW×2, where NL is the number of antenna lanes. NL is calculated as the number of antennas divided by the antenna interleaving factor. For example, with eight antennas and an antenna interleave factor of 2, the number of antenna lanes NL is equal to 4.

The width of the latency-compensated `TUSER` sideband channel, denoted by UW, can be chosen at customization time.

Downlink Data Input Interface Ports

Port Name	I/O	Clock	Description
s_axis_dl_din_tdata[2*`IW`*`NL`-1:0]	I	s_axis_aclk	Input sample data. Width is determined by the input sample width `IW` (16 for 16-bit samples and 24 for 18-bit samples) and the number of antenna lanes `NL`.
s_axis_dl_din_tvalid	I	s_axis_aclk	*Valid* handshake signal for the data input channel. The upstream logic uses this to signal that it can provide data.
s_axis_dl_din_tready	O	s_axis_aclk	*Ready* handshake signal for the data input channel. The core uses this to signal that it can accept data.
s_axis_dl_din_tlast	I	s_axis_aclk	*Last* framing signal for the data input channel. The core does not rely on this signal for operation, but passes it through to the data output interface after applying latency compensation.
s_axis_dl_din_tid[7:0]	I	s_axis_aclk	Transaction ID associated with the sample(s) on s_axis_dl_din_tdata. The lower four bits [3:0] indicate the antenna interleaving phase and the upper four bits [7:4] indicate the component carrier ID.
s_axis_dl_din_tuser[`UW`-1:0]	I	s_axis_aclk	User-defined framing information. The width of the field is defined when configuring the core. Used for triggering configuration updates within the core. Also passed through to the data output interface after applying latency compensation.

Downlink Data Output Interface Ports

Port Name	I/O	Clock	Description
m_axis_dl_dout_tdata[2*`OW`*`NA`-1:0]	O	s_axis_aclk	Output sample data. Width is determined by the output sample width `OW` (16 for 16-bit samples and 24 for 18-bit samples) and the number of antennas `NA`.
m_axis_dl_dout_tvalid	O	s_axis_aclk	*Valid* handshake signal for the data output channel. The downstream logic uses this to identify cycles with valid sample data.
m_axis_dl_dout_tlast	O	s_axis_aclk	*Last* framing signal for the data input channel. This is a latency-compensated version of s_axis_dl_din_tlast.
m_axis_dl_dout_tuser[`UW`-1:0]	O	s_axis_aclk	User-defined framing information. The width of the field is defined when configuring the core. This is a latency-compensated version of s_axis_dl_din_tuser.

Uplink Data Input Interface Ports

Port Name	I/O	Clock	Description
s_axis_ul_din_tdata[2*`IW`*`NA`-1:0]	I	s_axis_aclk	Input sample data. Width is determined by the input sample width `IW` (16 for 16-bit samples and 24 for 18-bit samples) and the number of antennas `NA`.
s_axis_ul_din_tvalid	I	s_axis_aclk	*Valid* handshake signal for the data input channel. The upstream logic uses this to signal that it can provide data.
s_axis_ul_din_tready	O	s_axis_aclk	*Ready* handshake signal for the data input channel. The core uses this to signal that it can accept data.
s_axis_ul_din_tlast	I	s_axis_aclk	*Last* framing signal for the data input channel. The core does not rely on this signal for operation, but passes it through to the data output interface after applying latency compensation.
s_axis_ul_din_tuser[`UW`-1:0]	I	s_axis_aclk	User-defined framing information. The width of the field is defined when configuring the core. Used for triggering configuration updates within the core. Also passed through to the data output interface after applying latency compensation.

Uplink Data Output Interface Ports

Port Name	I/O	Clock	Description
m_axis_ul_dout_tdata[2*`OW`*`NL`-1:0]	O	s_axis_aclk	Output sample data. Width is determined by the output sample width `OW` (16 for 16-bit samples and 24 for 18-bit samples) and the number of antenna lanes `NL`.

Port Name	I/O	Clock	Description
m_axis_ul_dout_tvalid	O	s_axis_aclk	*Valid* handshake signal for the data output channel. The downstream logic uses this to identify cycles with valid sample data present.
m_axis_ul_dout_tlast	O	s_axis_aclk	*Last* framing signal for the data input channel. This is a latency-compensated version of s_axis_ul_din_tlast.
m_axis_ul_dout_tid[7:0]	O	s_axis_aclk	Transaction ID associated with the sample(s) on m_axis_ul_dout_tdata. The lower four bits [3:0] indicate the antenna interleaving phase and the upper four bits [7:4] indicate the component carrier ID.
m_axis_ul_dout_tuser[UW-1:0]	O	s_axis_aclk	User-defined framing information. The width of the field is defined when configuring the core. This is a latency-compensated version of s_axis_ul_din_tuser.

Memory Mapped AXI4-Lite Interface Ports

Port Name	I/O	Clock	Description
s_axi_ctrl_arready	O	s_axi_aclk	Indicates that the core is ready for a read address on s_axi_araddr.
s_axi_ctrl_arvalid	I	s_axi_aclk	Indicates that the bus logic is providing a read address on s_axi_araddr.
s_axi_ctrl_araddr[17:0]	I	s_axi_aclk	Read address. Accepted when s_axi_ctrl_arready and s_axi_ctrl_arvalid are high on the same clock cycle.
s_axi_ctrl_awready	O	s_axi_aclk	Indicates that the core is ready for a write address on s_axi_awaddr.
s_axi_ctrl_awvalid	I	s_axi_aclk	Indicates that the bus logic is providing a write address on s_axi_awaddr.
s_axi_ctrl_awaddr[17:0]	I	s_axi_aclk	Write address. Accepted when s_axi_ctrl_awvalid and s_axi_awready are asserted on the same clock cycle.
s_axi_ctrl_bready	I	s_axi_aclk	Indicates that the bus logic is ready to receive a write transaction response.
s_axi_ctrl_bvalid	O	s_axi_aclk	Indicates that the core has completed a write transaction and the response on s_axi_ctrl_bresp is valid.
s_axi_ctrl_bresp[1:0]	O	s_axi_aclk	Write transaction response (00 = OK, 1x = ERROR).
s_axi_ctrl_rready	I	s_axi_aclk	Indicates that the bus logic is ready to receive read data.
s_axi_ctrl_rvalid	O	s_axi_aclk	Indicates that the core has completed a read transaction and that the data on s_axi_ctrl_rdata and response on s_axi_ctrl_rresp are valid.
s_axi_ctrl_rresp[1:0]	O	s_axi_aclk	Read transaction response (00 = OK, 1x = ERROR).
s_axi_ctrl_rdata[31:0]	O	s_axi_aclk	Read data.
s_axi_ctrl_wready	O	s_axi_aclk	Indicates that the core is ready to receive write data on s_axi_ctrl_wdata.
s_axi_ctrl_wvalid	I	s_axi_aclk	Indicates that the bus logic is providing write data on s_axi_ctrl_wdata.

Port Name	I/O	Clock	Description
s_axi_ctrl_wdata[31:0]	I	s_axi_aclk	Write data.
irq	O	s_axi_aclk	Interrupt request to the processor. (Not part of the AXI4-Lite standard set of signals.)

Clock and Reset Ports

Port Name	I/O	Clock	Description
s_axis_aclk	I	-	Clock for AXI4-Stream interfaces and internal operation of block.
s_axis_aresetn	I	s_axis_aclk	Active-Low synchronous reset for AXI4-Stream interfaces and internal operation of block.
s_axi_aclk	I	-	Clock for memory mapped AXI interface.
s_axi_aresetn	I	s_axi_aclk	Active-Low synchronous reset for memory mapped AXI interface.

Chapter 4

Designing with the Core

This section includes guidelines and additional information to facilitate designing with the core.

General Design Guidelines

Registering Signals

To simplify timing and increase system performance in a programmable device design, keep all inputs and outputs registered between the user application and the core. This means that all inputs and outputs from the user application should come from, or connect to, a flip-flop. While registering signals might not be possible for all paths, it simplifies timing analysis and makes it easier for the Xilinx® tools to place and route the design.

Make Only Allowed Modifications

You should not modify the core. Any modifications can have adverse effects on system timing and protocol compliance. Supported user configurations of the core can only be made by selecting the options in the customization IP dialog box when the core is generated.

Clocking

The RFSoC DFE DUC-DDC Mixer core has two clocks, `s_axis_aclk` and `s_axi_aclk`. The internal datapath logic (including DFE DUC-DDC mixer primitives) and the input, output, and mixer I/O interfaces all operate synchronously to `s_axis_aclk`. A typical frequency for this clock in a 5G wireless system would be 491.52 MHz.

The memory mapped AXI4-Lite interface operates synchronously to `s_axi_aclk`. No timing relationship between this clock and `s_axis_aclk` is assumed. This allows the core to be interfaced to a microprocessor bus which can run at a lower frequency than the sample-processing frequency (for example, 250 MHz).

Resets

The RFSoC DFE DUC-DDC Mixer core has two resets, `s_axis_aresetn` and `s_axi_aresetn`, corresponding to the two clock domains described above. These signals are active-Low and are synchronous to the associated clocks.

`s_axis_aresetn` is used to reset the core to its default state by asserting the reset signal for at least two active clock cycles. Resetting the core clears all state information including the contents of the filter delay pipeline, the phase accumulators and NCO outputs, and all carrier and antenna configuration information. The core is ready for operation on the first clock cycle following the deassertion of `s_axis_aresetn`.

`s_axi_aresetn` is part of the AXI4-Lite bus infrastructure and resets the memory mapped AXI4-Lite interface and other peripherals connected to the bus.

RECOMMENDED: *It is strongly recommended that both reset signals remain asserted at system start up until both `s_axis_aclk` and `s_axi_aclk` are stable. Attempting to operate the AXI4-Lite interface without the `s_axis_aclk` present, or to operate the DUC-DDC mixer datapath without the `s_axi_aclk` present, may lead to unexpected behavior and system instability.*

Interfacing to the Core

Data Format

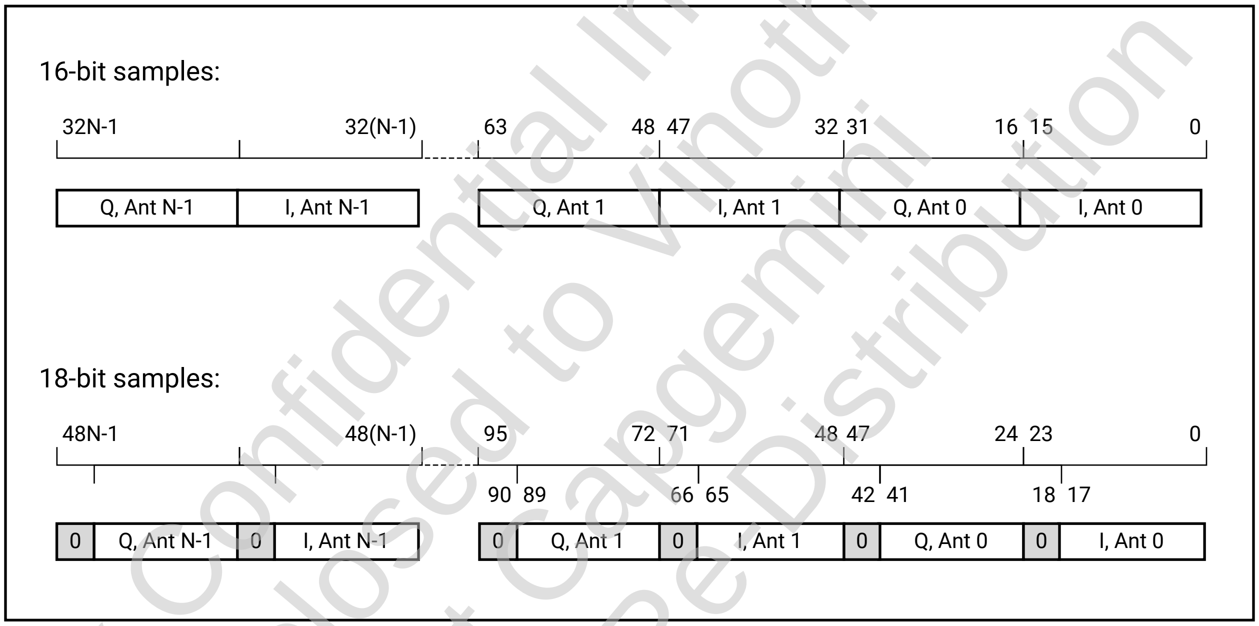
The TDATA buses associated with the AXI4-Stream interfaces all use the same format for transferring sample data. The in-phase (I) part of each sample is placed at the lowest-numbered bit position, followed by the quadrature (Q) part. Each part of the sample has its least-significant bit aligned to an 8-bit boundary, with zero-padding added above the most significant bit if the sample width is not a multiple of 8 bits.

For the downlink data output interface (`m_axis_dl_dout_tdata`) and uplink data input interface (`s_axis_ul_din_tdata`), the data bus width scales with the number of antennas supported. This number is configurable when the core is generated. If the number of antennas is greater than one, the samples for the multiple antennas are concatenated from lowest numbered (RHS) to highest numbered (LHS) position within the bus.

For the downlink data input interface (`s_axis_dl_din_tdata`) and uplink data output interface (`m_axis_ul_dout_tdata`), the number of antenna lanes is defined as the total number of supported antennas divided by the antenna interleave factor. These parameters are configurable when the core is generated. If the number of antenna lanes is greater than one, the samples for the multiple antennas being transferred on each clock cycle are concatenated in ascending order from lowest numbered (RHS) to highest numbered (LHS) position within the bus.

The following figure shows the data format for both 16-bit and 18-bit sample widths.

Figure 4: **Data Format for 16-bit and 18-bit Sample Widths**



The sample data is treated as two's complement fixed-point data. The position of the binary point is arbitrary.

Latency Compensation

The `TUSER` and `TLAST` signals on the AXI4-Stream output interfaces are generated from the equivalent signals on the corresponding input interface, after latency compensation. The core delays these signals according to the datapath pipeline depth. This datapath latency is the same for all component carriers regardless of their sample rate.

Additional delay can be added to the `TUSER` and `TLAST` signals using the API, if required. For example, this can be used to set their latency to match the group delay of a specific component carrier.

The group delay of a component carrier depends on the interpolation or decimation rate selected. Each increase in rate requires an additional half-band filter and therefore increases the group delay. The group delay is given below in terms of samples at the downlink output or uplink input.

Table 1: **Group Delay**

Interpolation/ Decimation Rate	x1	x2	x4	x8	x16
Group Delay	0	23	55	115	235

In a downlink configuration with several component carriers operating at different rates, a delay offset may be required for one or more component carriers at the downlink input in order for all component carriers to be aligned at the downlink output. The number of samples that each component carrier must be delayed for alignment can be calculated using the formula below. This delay is relative to the component carrier(s) with the highest interpolation rate and therefore the longest group delay (which appears as group_delay$_{cc\_max}$ in the equation).

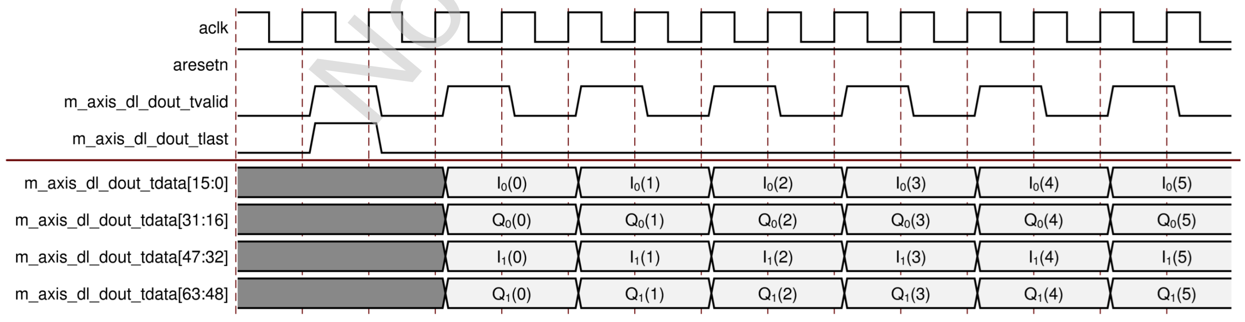
$$sample\_delay_{cc\_n} = \frac{group\_delay_{cc\_max} - group\_delay_{cc\_n}}{rate_{cc\_n}}$$

Flow Control

The RFSoC DFE DUC-DDC Mixer core does not support an arbitrary sample-by-sample flow control handshake. There is no way to stall data on any of the input or output data interfaces. The `TVALID` signals of the AXI4-Streams are expected to remain High constantly when the core is operational. The only exception is when the number of mixer clocks per sample is greater than 1.

When the core is configured for operation with two or four mixer clocks per sample, `TVALID` is used to qualify each sample on the downlink output or uplink input interface. This is shown in the following diagram for a downlink configuration with two mixer clocks per sample.

Figure 5: **Downlink Data Output Interface with Two Mixer Clocks Per Sample**


aclk
aresetn
m_axis_dl_dout_tvalid
m_axis_dl_dout_tlast
m_axis_dl_dout_tdata[15:0]
m_axis_dl_dout_tdata[31:16]
m_axis_dl_dout_tdata[47:32]
m_axis_dl_dout_tdata[63:48]
I0(0) I0(1) I0(2) I0(3) I0(4) I0(5)
Q0(0) Q0(1) Q0(2) Q0(3) Q0(4) Q0(5)
I1(0) I1(1) I1(2) I1(3) I1(4) I1(5)
Q1(0) Q1(1) Q1(2) Q1(3) Q1(4) Q1(5)

On the data input interfaces, `TVALID` must be asserted to indicate that data is present on `TDATA`. The core will not deassert `TREADY` during normal operation.

On the data output interfaces, `TVALID` is asserted by the core to indicate that data is present on `TDATA` on the current cycle. No `TREADY` signal is provided on these interfaces. The output `TVALID` signal will follow the input `TVALID` signal with a delay equal to the processing latency of the datapath.

When operating below the maximum capacity, for example, because not all component carriers are active, the `TVALID` signals on the uplink output interface and the downlink input interface will be high even on cycles in which no valid data is being transferred. The `TID` signals on these interfaces are used to identify which bus cycles carry data samples belonging to active component carriers.

Component Carrier Sequencing

The RFSoC DFE DUC-DDC Mixer core provides a high degree of flexibility for defining the time-division multiplexing (TDM) of the samples of multiple component carriers on the downlink data input and uplink data output interfaces. The desired component carrier sequence can be set up using software API calls, and activated at a chosen point in time using the TUSER-based triggering mechanism.

***Note*:** Component-carrier sequencing does not apply to the downlink data output or uplink data input interfaces.

This section makes use of the following parameters to define the sequencing mechanism.

- F_c, the datapath clock frequency
- F_s, the aggregate carrier sample rate
- $F_{cc,n}$, the sample rate of carrier n
- N_{AC}, the number of active carriers
- AILV, the antenna interleaving factor
- L, the length of the CC sequence

The aggregate carrier sample rate F_s available at the downlink input or uplink output is given by the AXI4-Stream clock frequency F_c divided by the antenna interleave factor AILV which can take a value of 1, 2, 4, or 8. This sample rate can be allocated to a single carrier or divided up between multiple component carriers as described below.

Each component carrier to be processed by the core must have a sample rate $F_{cc,n}$ that is equal to F_c, $F_c/2$, $F_c/4$, $F_c/8$ or $F_c/16$. The sum of $F_{cc,n}$ over all active carriers (n=0..N_{AC}-1) must be less than or equal to F_s and N_{AC}×AILV must be less than or equal to 16.

The data belonging to each component carrier is identified by a 4-bit value on `s_axis_dl_din_tid[7:4]` or `m_axis_ul_dout_tid[7:4]` known as the component carrier identifier (CCID). The CCID sequence determines the order in which carriers are multiplexed on the data bus. The sequence length L may be set to a value of 1, 2, 4, 8, or 16 when the core is activated via the API, where L×AILV is less than or equal to 16.

The number of times the CCID for active carrier n appears in the CCID sequence must be equal to $F_{cc,n} \times L \div F_s$. The ordering of entries within the CCID sequence is not important.

In a simple single-carrier configuration where one carrier uses the entire aggregate sample rate, no TDM is required for component carrier multiplexing. The component carrier sequence may have a length of one. Or, if the sequence length is greater than one, every element of the sequence must have the same value, which is the CCID assigned to that carrier.

When a single component carrier is present but its sample rate is less than F_s, some of the sample slots on the input and output interfaces will be unused. Which cycles contain data for the active component carrier and which are inactive is indicated by means of the CCID field. There is no CCID value specifically assigned to stand for an unused slot. Any CCID value that is not allocated to an active carrier can be used for this purpose.

For example, consider a configuration with F_s = 491.52 MSPS aggregate carrier sample rate and a single component carrier with a 122.88 MSPS sample rate, assigned to CCID 0. If the CCID sequence length has been set to 8, then a valid CCID sequence would be {0, 15, 15, 15, 0, 15, 15, 15}. Here the dummy number 15 has been chosen to represent all the unused slots in the sequence. Other sequences are possible, provided that the number 0 (the CCID of the active carrier) appears in exactly 25% of the slots.

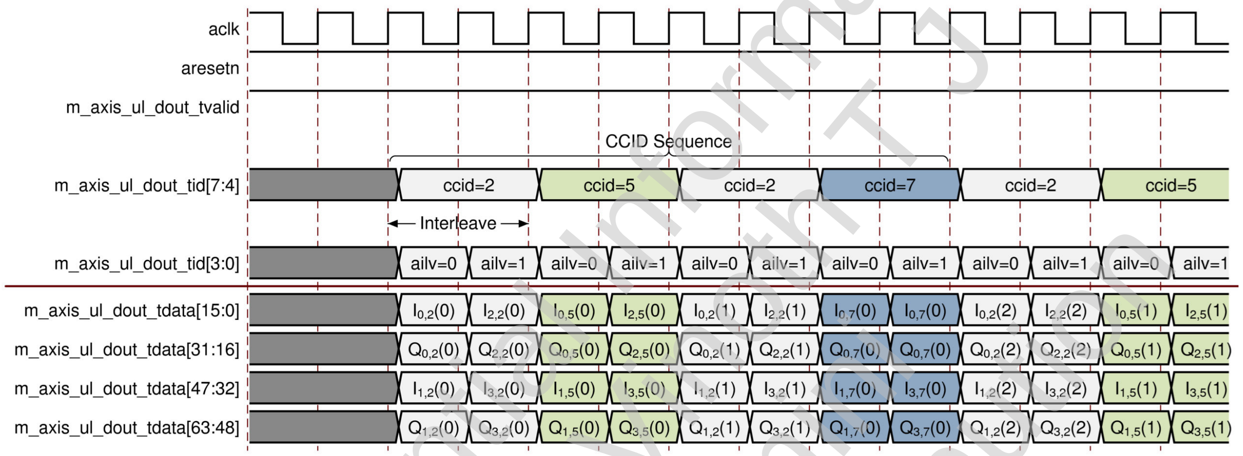
If two carriers are multiplexed, each carrier having half of the aggregate carrier sample rate, the TDM sequence alternates between the carriers and has a minimum length of two. For example, if a second 122.88 MSPS carrier with CCID 1 is added to previous example, then a valid CCID sequence would be {0, 1, 15, 15, 0, 1, 15, 15}. Another would be {0, 15, 1, 15, 0, 15, 1, 15}. The core supports an arbitrary interleaving of active carriers and unused slots within the sequence, although regular patterns with consistent spacing between samples belonging to the same carrier are most likely to be useful in practical systems.

Extending this multiplexing scheme to carriers with different sample rates is straightforward. For example, if the sample rate of the carrier with CCID 0 in the previous example is increased to 245.76 MSPS, then a valid CCID sequence would be {0, 1, 0, 15, 0, 1, 0, 15}. The value 0 appears in 50% of the slots because CCID 0 occupies 50% of the aggregate carrier bandwidth. The remaining 50% is split between CCID 1 (25%) and the unused samples (25%) denoted by dummy CCID 15.

When the antenna interleave factor is greater than 1, antenna data interleaving takes place at the level below the component carrier sequencing. All data samples for all antennas are transferred for the current carrier in the sequence before moving on to the next carrier.

The following diagram shows an example of the component carrier sequencing and how it interacts with the interleaving of samples for multiple antennas.

Figure 6: **Timing for Component Carrier Sequencing**


aclk
aresetn
m_axis_ul_dout_tvalid
CCID Sequence
m_axis_ul_dout_tid[7:4]
ccid=2
ccid=5
ccid=2
ccid=7
ccid=2
ccid=5
Interleave
m_axis_ul_dout_tid[3:0]
ailv=0
ailv=1
m_axis_ul_dout_tdata[15:0]
m_axis_ul_dout_tdata[31:16]
m_axis_ul_dout_tdata[47:32]
m_axis_ul_dout_tdata[63:48]
I0,2(0) I2,2(0) I0,5(0) I2,5(0) I0,2(1) I2,2(1) I0,7(0) I0,7(0) I0,2(2) I2,2(2) I0,5(1) I2,5(1)
Q0,2(0) Q2,2(0) Q0,5(0) Q2,5(0) Q0,2(1) Q2,2(1) Q0,7(0) Q0,7(0) Q0,2(2) Q2,2(2) Q0,5(1) Q2,5(1)
I1,2(0) I3,2(0) I1,5(0) I3,5(0) I1,2(1) I3,2(1) I1,7(0) I3,7(0) I1,2(2) I3,2(2) I1,5(1) I3,5(1)
Q1,2(0) Q3,2(0) Q1,5(0) Q3,5(0) Q1,2(1) Q3,2(1) Q1,7(0) Q3,7(0) Q1,2(2) Q3,2(2) Q1,5(1) Q3,5(1)

Here the number of antennas is four, the antenna interleave factor is 2, and the sample width is 16 bits. There are three component carriers configured; they are numbered 2, 5, and 7. Carriers 5 and 7 each have half the sample rate of carrier 2. The resulting CCID sequence has a minimum length of four and can be represented as {2, 5, 2, 7}. The I and Q parts of each sample in the diagram carry a pair of subscripts `a`, `c`, where `a` is the antenna number and `c` is the component carrier ID. The number in parentheses after the sample is the time index of that sample within the carrier sample stream.

On the downlink input data interface, the sequence of values on `TID[7:4]` is expected to match the programmed CC sequence. Where the antenna interleave factor AILV is greater than one, the value on TID[3:0] (shown as `ailv` in the diagram) is expected to count from 0 to AILV-1 within each element of the CCID sequence. The core monitors `TID` and reports a CC sequence error if a mismatch is detected on either of these sub-fields.

The clock cycle on which the component carrier interleaving sequence begins is set when the core is activated. This sequence start position cannot subsequently be changed unless the core is deactivated, brought into and out of reset, and reactivated. Activation is performed using one of the triggering mechanisms described in the following section. To ensure that the start position can be controlled precisely, a single-shot TUSER trigger type should be used for the activation process.

Configuring and Controlling the Core

A full software API is provided to facilitate the configuration, control and monitoring of the RFSoC DFE DUC-DDC Mixer core operation. The API is a set of data structures and functions, defined in the C programming language, which provide software applications with an abstracted view of the RFSoC DFE DUC-DDC Mixer core.

Triggering Configuration Changes

All changes to the configuration of the RFSoC DFE DUC-DDC Mixer core that affect the processing of data are controlled by a triggering mechanism. This includes activating and deactivating the core, adding and removing component carriers from the TDM sequence, changes to NCO settings and antenna gain, and entry into and exit from low power mode.

First, an `XDfeMix_TriggerCfg` structure is used to define the triggers. This is passed to the `XDfeMix_SetTriggersCfg` function to make the new trigger setup current. Then the trigger is made active through a further software API call, according to the particular configuration change that is being made. When the trigger condition is met, the requested change is put into effect by the hardware.

A variable `TriggerCfg` of type `XDfeMix_TriggerCfg` has three fields. `TriggerCfg.Activate` defines the trigger to be used for activating and deactivating the core. `TriggerCfg.CCUpdate` defines the trigger to be used for component carrier, antenna gain and NCO updates. `TriggerCfg.LowPower` defines the trigger to be used for entry into and exit from low power mode.

There are three types of triggers:

- **Immediate Triggers:**

  The simplest type of trigger is the immediate trigger. To define an immediate trigger, set `TriggerCfg.<trigger>.Mode` to 0. When an API call that uses the corresponding trigger is subsequently made, the trigger condition is considered to be met as soon as the API performs a write to the hardware register controlling the configuration parameter in question. The configuration change therefore takes effect immediately.

  Immediate triggers are primarily used for development and debugging. Because the timing of a register write across the AXI interconnect cannot be guaranteed, and the AXI4-Lite control interface operates in a different clock domain from the AXI4-Stream data interface, this mechanism is not suitable for use in situations where sample-accurate alignment of configuration changes is required.

- **Single-shot Triggers:**

For activating and deactivating the core and for updating the component carrier configuration, a single-shot trigger should be used. To define a single-shot trigger, set `TriggerCfg.<trigger>.Mode` to 1. It is also necessary to define the source for the trigger by setting `TriggerCfg.<trigger>.TUSERBit` to indicate which bit of `s_axis_dl_din_tuser` or `s_axis_ul_din_tuser` should provide the trigger event. The type of trigger event the hardware should respond to is determined by `TriggerCfg.<trigger>.TuserEdgeLevel`. For this field, a value of 0 defines an active-Low trigger, 1 defines an active-High trigger, 2 defines a falling-edge trigger and 3 defines a rising-edge trigger. When an API call that uses the corresponding trigger is subsequently made, the trigger condition is considered to be met as soon as the specified event occurs on the chosen bit of `s_axis_din_tuser` or `s_axis_ul_din_tuser`. The delay between the trigger event and the configuration change taking effect depends on the nature of the configuration change, as described in the following sections. Once the trigger has occurred, it is deactivated until the next API call enables it again.

- **Continuous Trigger:**

  Once activated, a continuous trigger causes the corresponding event to occur repeatedly without further software intervention. To define a continuous trigger, set `TriggerCfg.<trigger>.Mode` to 2 and define the `TUSERBit` and `TuserEdgeLevel` fields as described above for a single-shot trigger. When an API call that uses the corresponding trigger is subsequently made, the trigger condition is considered to be met as soon as the specified event occurs on the chosen bit of `s_axis_din_tuser`. The delay between the trigger event and the configuration change taking effect depends on the nature of the configuration change, as described in the following sections. The trigger remains enabled so that subsequent activity on the same TUSER bit can cause the requested configuration change to re-occur.

  Continuous triggers are primarily useful for entry into and exit from low power mode, for example to periodically enable and disable uplink and downlink processing in a TDD system.

  It is not possible to use a continuous trigger for component carrier, antenna gain or NCO updates.

Initialization

The RFSoC DFE DUC-DDC Mixer software API uses the libmetal abstraction layer to provide a consistent interface and use model independent of any operating system in use.

Each instance of the RFSoC DFE DUC-DDC Mixer core in the design requires a corresponding instance of an `XDfeMix` API object in the software. The API object is created and bound to the core using code such as the following.

```
/* Declare an instance pointer for the API object */
XDfeMix *InstancePtr = NULL;

/* Initialize an instance of the DUC-DDC/Mixer driver */
InstancePtr = XDfeMix_InstanceInit(XDFEMIX_NODE1_NAME);
```

The argument to `XDfeMix_InstanceInit` is the device ID. It controls how the API locates and binds to the core instance in hardware. The symbol definitions required vary from platform to platform and will typically be generated as part of the Vitis platform generation process. When using embedded Linux, the device ID comes from the device tree. When using a bare-metal implementation, the device ID comes from the `xparameters.h` header file.

Once an API object has been created successfully, the RFSoC DFE DUC-DDC Mixer core should be put into reset to ensure that data processing is halted and the IP is returned to its default state prior to configuration. This can be done by the software API as follows.

```
/* Reset the DUC-DDC/Mixer core */
XDfeMix_Reset(InstancePtr);
```

In order to ensure that the expected versions of the IP core and software components are in use, the API provides a function to query the version numbers. An example of the usage of these functions is shown below. This also provides a useful initial check that the hardware registers can be read successfully by the processor.

```
/* Declare version number objects */
XDfeMix_Version SwVersion;
XDfeMix_Version HwVersion;

/* Get software and hardware version numbers */
XDfeMix_GetVersions(InstancePtr, &SwVersion, &HwVersion);

/* Print the version numbers to the console*/
printf("SW Version: Major %d, Minor %d\n", SwVersion.Major,
SwVersion.Minor);
printf("HW Version: Major %d, Minor %d, Revision %d, Patch %d\n",
HwVersion.Major, HwVersion.Minor, HwVersion.Revision,
HwVersion.Patch);
```

After the core has been put into reset, call the `XDfeMix_Configure` function to complete the low-level configuration of the core and release the reset. This function checks the configuration of the IP core instance against the information in the device tree (when embedded Linux is used) or `xparameters.h` file (when a bare metal system is used).

```
/* Declare a configuration object */
XDfeMix_Cfg Cfg;

/* Configure the IP instance */
XDfeMix_Configure(InstancePtr, &Cfg);
```

Once the RFSoC DFE DUC-DDC Mixer core is configured, it must be initialized. The `XDfeMix_Initialize` function prepares the core for use by setting up the features and parameters that must remain fixed while the core is operating. The only feature in this category is the component carrier ID sequence length. The function is used as follows.

```
/* Declare and populate the initialization parameters object */
XDfeMix_Init Init;
Init.Sequence.Length = 16;    // CCID sequence length (must be a power of 2)

/* Initialize the IP core */
XDfeMix_Initialize(InstancePtr, &Init);
```

After initialization is complete, the core is ready to be activated. Activation brings the core into an operational state in which the core can process data.

The start position of the component carrier sequence is set at the point of activation. To ensure that this start position can be accurately synchronized with other logic in the system that is providing or consuming the TDM data stream of the RFSoC DFE DUC-DDC Mixer core downlink input or uplink output, the activation process should use a single-shot trigger based on a bit within the `s_axis_dl_din_tuser` or `s_axis_ul_din_tuser` bus.

The core activation function takes a Boolean argument which determines whether the low power mode should be enabled or disabled.

An example of the core activation procedure is shown below.

```
/* Declare and populate a trigger configuration object */
XDfeMix_TriggerCfg TriggerCfg;
TriggerCfg.Activate.Mode = 1;            // Single-shot trigger
TriggerCfg.Activate.TUSERBit = 0;        // on TUSER bit 0
TriggerCfg.Activate.TuserEdgeLevel = 1;  // when high

/* Set the trigger configuration */
XDfeMix_SetTriggersCfg(InstancePtr, &TriggerCfg);

/* Activate the IP core (disabling the low power mode) */
XDfeMix_Activate(InstancePtr, false);
```

The call to `XDfeMix_Activate` will return immediately. The core will be activated when the trigger condition is met.

Latency Compensation Setup

The RFSoC DFE DUC-DDC Mixer core generates the `TUSER` and `TLAST` signals on the output interface by delaying the signals received on the input interface. The default behavior is to delay `TUSER` and `TLAST` by a number of clock cycles equal to the latency of the data processing pipeline. This datapath delay can be queried from software using the following code.

```
u32 dataDelay;

dataDelay = XDfeMix_GetTDataDelay(InstancePtr, 0);
```

The second argument to `XDfeMix_GetTDataDelay` is the number of additional clock cycles to add to the datapath delay. When a value of 0 is specified, the function returns the number of clock cycles of datapath delay with no adjustment applied.

The API also provides control over the amount of additional delay applied to the `TUSER` and `TLAST` signals at the output. This extra delay can be set to any number of clock cycles from 0 to 1023 as required, for example to match the group delay of a particular component carrier (see Latency Compensation). The current setting can also be queried. The following code illustrates the required function calls.

```
XDfeMix_SetTUserDelay(InstancePtr, 15);    // Additional 15 clock-cycle delay
currentDelay = XDfeMix_GetTUserDelay(InstancePtr);
```

***Note*:** The TUSER delay can only be set prior to core activation because changes to the delay while the core is active could lead to undefined behavior on `m_axis_dl_dout_tuser` or `m_axis_ul_dout_tuser`.

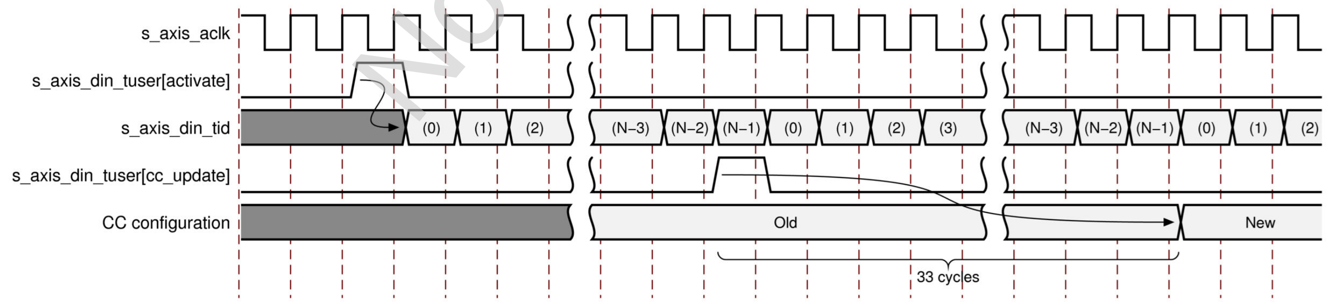
Carrier Setup and CCID Sequencing

The API provides functions to dynamically add, update, and remove component carriers, and to manage the CCID sequence.

Updates to the component carrier configuration are typically performed using a single-shot trigger based on a bit of the TUSER bus, similar to that used for the activation process. This allows the timing of the update to be aligned precisely to a particular clock cycle. The TUSER bit transition used for a component carrier update trigger should usually be aligned to the cycle before the start of the recurring CCID sequence pattern. The spacing between trigger events (activation or CC update) on the TUSER bus must always be an exact multiple of 16 clock cycles.

Updates to the component carrier configuration take effect (32×CPS + 1) cycles after the trigger condition occurs, regardless of the CCID sequence length. The relative timing of these events is illustrated in the diagram below in the case where CPS = 1.

Figure 7: **Component Carrier Update Sequence**



The functions described in this section operate on one component carrier at a time, requiring a separate trigger for each update. To add, remove or update multiple component carriers using a single trigger, see Making Simultaneous Updates.

After the core has been activated, the `XDfeMix_AddCC` function can be used to start processing for a new component carrier. This function takes the carrier configuration structure, CCID value, NCO configuration, and a bitmap representing the position of this component carrier's samples within the CCID sequence. It then adds the new component carrier to any existing carriers that are already active, and writes the new setup into the core's configuration registers. Finally it sets up the CC update trigger before returning. The new carrier is activated when the trigger condition occurs as described above.

Before calling `XDfeMix_AddCC`, the CC update trigger must be configured using `XDfeMix_SetTriggersCfg`. The CC update trigger can be configured at the same time as the Activate trigger during the initialization phase, if desired. If the same trigger configuration is to be used multiple times by successive component carrier add, update, or remove function calls, it only needs to be configured once before the first call.

It is also necessary to clear the event status before adding, removing, or updating a CC, in order for the software to be able to detect that the CC update trigger condition was met. This is achieved by means of the `XDfeMix_ClearEventStatus` function.

The following code sequence illustrates how to set up a new component carrier using the steps above.

```
/* Declare and populate trigger, carrier & NCO configuration and CCID
bitmap */
XDfeMix_TriggerCfg TriggerCfg;
XDfeMix_CarrierCfg CarrierCfg;
XDfeMix_Status status = { 0 };
XDfeMix_NCO NCO;
u32 BitSequence;
u32 CCID;
u32 result;
double FreqMhz;
double NcoFreqMhz;
double FrequencyControlWord;

CarrierCfg.DUCDDCCfg.NCOIdx = 0; // Use first NCO
CarrierCfg.DUCDDCCfg.CCGain = 3; // 0dB output gain for this CC (downlink
only)
NCO.NCOGain = 0;                 // 0dB NCO gain for this CC

FreqMhz = 40.0;                  // Mix to 40 MHz
NcoFreqMhz = 491.52;             // System clock is 491.52 MHz
FrequencyControlWord = floor((FreqMhz / NcoFreqMhz) * 0x80000000);
NCO.FrequencyCfg.FrequencyControlWord = FrequencyControlWord;

BitSequence = 0xffff;        // CC will occupy every timeslot
CCID = 0;                    // CCID number chosen for this carrier

TriggerCfg.CCUpdate.Mode = 1;              // Single-shot trigger
TriggerCfg.CCUpdate.TUSERBit = 0;          // on bit 0
TriggerCfg.CCUpdate.TuserEdgeLevel = 1;    // when high
```

```
/* Set up the CC update trigger */
XDfeMix_SetTriggersCfg(InstancePtr, &TriggerCfg);

/* Clear the event status */
status.CCUpdate = 1;
XDfeMix_ClearEventStatus(InstancePtr, &status);

/* Add the component carrier */
result = XDfeMix_AddCC(InstancePtr, CCID, BitSequence, &CarrierCfg, &NCO);

if (result != XST_SUCCESS)
  printf("Add CC #%d failed with error code %d!\n", CCID, result);
else
{
  printf("Successfully scheduled CC #%d for addition.\n", CCID);

  status.CCUpdate = 0;

  /* (Demonstration only!) Poll for successful CC update trigger */
  while (status.CCUpdate == 0)
  {
    XDfeMix_GetEventStatus(InstancePtr, &status);
    usleep(10000);
  }
  printf("Addition of CC #%d was triggered successfully.\n", CCID);
}
```

The polling loop at the end of this example is not realistic, but illustrates how the API can determine whether the CC update trigger condition has occurred.

When in downlink mode, valid settings for `CarrierCfg.DUCDDCCfg.CCGain` are 0 for -18dB gain, 1 for -12dB, 2 for -6dB, and 3 for 0dB. The gain is applied at the output of the mixer. This gain setting is ignored when the core is in uplink mode.

The sample rate of each CC is inferred by `XDfeMix_AddCC` from its `BitSequence` argument, and used to configure the appropriate interpolation or decimation parameters.

Adding multiple component carriers can be done by following the sequence above several times. Note that a separate call to `XDfeMix_AddCC` is required for each component carrier and a separate trigger is required for each update when using this method. See Making Simultaneous Updates for an alternative set of API calls which allow multiple carriers to be added using a single trigger.

When defining multiple CCs, ensure that the bit sequence values do not overlap. For example, to create one 122.88 MSPS CC and three 30.72 MSPS CCs, a valid set of bit sequences could be:

```
u32 BitSeq0_122p88 = 0x5555;
u32 BitSeq1_30p72  = 0x0202;
u32 BitSeq2_30p72  = 0x0808;
u32 BitSeq3_30p72  = 0x2020;

assert (BitSeq0_122p88 & BitSeq1_30p72 & BitSeq2_30p72 & BitSeq3_30p72 ==
0);
```

The examples in this section assume a CCID sequence length of 16. For shorter CCID sequences, a smaller bitmap (8-bit, 4-bit etc.) can be used. The pattern specified by the bitmap must match the pattern supplied to the core on `s_axis_din_tid`. If it does not match, the core records a CC sequence error. This condition can be detected using the following code:

```
XDfeMix_Status status = { 0 };
XDfeMix_GetEventStatus(InstancePtr, &status);
if (status.CCSequenceError != 0)
  printf("A CC sequence error occured!\n");
```

The `XDfeMix_UpdateCC` function is used in a similar way to `XDfeMix_AddCC`. The sequence of function calls required, including event clearing and triggering setup, is identical for both functions. The main difference between the two is that `XDfeMix_UpdateCC` does not alter the CCID sequence and therefore does not take a bit sequence as an argument. It operates on an existing active component carrier, overwriting its configuration with the new values specified in the `XDfeMix_CarrierCfg` structure. This can be used to alter the gain of an existing CC.

The function `XDfeMix_RemoveCC` takes only a CCID number as an argument. It deactivates the specified CC and removes it from the CCID sequence when the trigger condition occurs.

An active component carrier can be moved seamlessly from one NCO to another without having to disturb data processing for the carrier by removing and re-adding it. This is done using the `XDfeMix_MoveCC` function as shown in the following code.

```
s32 CCID     = 5;   // Move CCID #5
u32 from_NCO = 0;   // from NCO #0
u32 to_NCO   = 2;   // to NCO #2

XDfeMix_MoveCC(InstancePtr, CCID, 0, from_NCO, to_NCO);
```

The third argument to `XDfeMix_MoveCC` should be treated as a dummy argument with a value of 0. After the CC update trigger has occurred, the specified CC will be mixed using the new NCO and the old one will be available for allocation to another CC.

As for `XDfeMix_AddCC`, the `XDfeMix_UpdateCC`, `XDfeMix_MoveCC`, and `XDfeMix_RemoveCC` functions act only on a single carrier and require one trigger event per function call.

NCO Configuration

The RFSoC DFE DUC-DDC Mixer core uses a dual-modulus CORDIC-based NCO to provide direct digital synthesis for a wide range of frequencies with zero frequency error. The NCOs can be configured via the API.

A simple N-bit single-modulus phase accumulator produces a ramp signal with an increment known as the Frequency Control Word (FCW), which is calculated as follows:

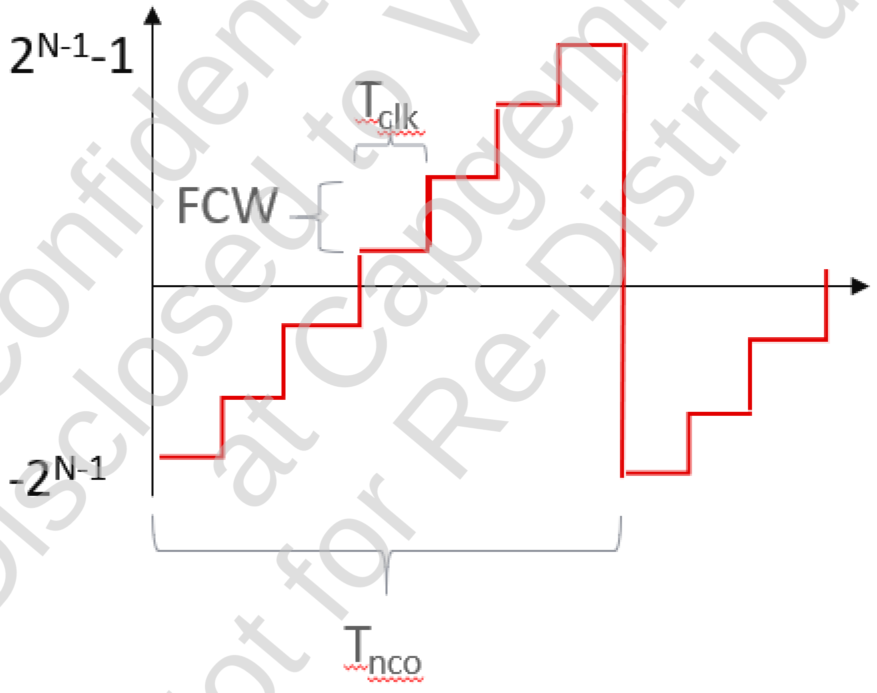
$$FCW_{ideal} = \frac{2^{N_{bits}} f_{NCO}}{f_{clk}}$$

$$FCW = \lfloor FCW_{ideal} \rfloor$$

In the RFSoC DFE DUC-DDC Mixer core, N_{bits} is equal to 32. The output of the 32-bit phase accumulator is rounded to 18 bits. An 18-bit phase offset is then applied, and the result is fed to a CORDIC block which converts it into a complex sinusoid with the corresponding frequency and phase.

The phase ramp will have frequency of exactly f_{NCO} when FCW_{ideal} is an integer and no rounding is required ($FCW \equiv FCW_{ideal}$). This situation is shown in the figure below, where T_{clk} is $1/f_{clk}$ and T_{nco} is $1/f_{NCO}$

Figure 8: **Simple Single-modulus NCO Phase Accumulator**



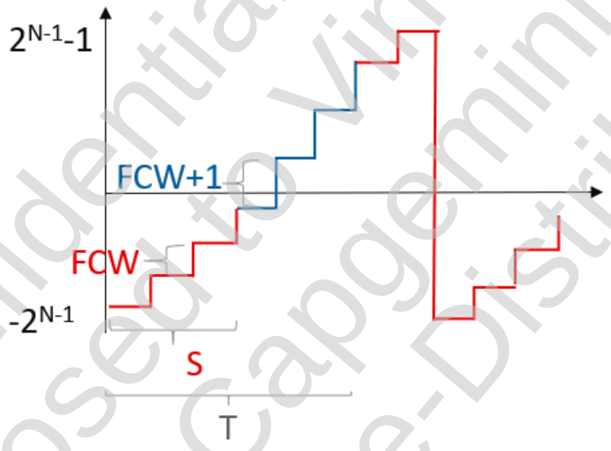
However, when FCW_{ideal} is not an integer, a frequency error results due to the quantization error in the value of FCW. For applications such as wireless communications with high accuracy and long run time requirements, the dual modulus NCO can prevent this frequency error from occurring.

In a dual-modulus phase accumulator, the phase increment on any given clock cycle may take the value of FCW or FCW+1. The ratio of cycle which use FCW to cycles which use FCW+1 is equal to the fractional part of FCW_{ideal}. Provided that the fractional part is a rational quantity, the quantization error of FCW is exactly canceled out by the quantization error of FCW+1 over the dual-modulus cycle and the effective FCW thus achieved is equal to FCW_{ideal}. This eliminates the frequency error.

$$FCW_{effective} = \frac{S(FCW) + (T - S)(FCW + 1)}{T}$$

The scheme is illustrated in the figure below.

Figure 9: **Dual-modulus NCO Parameters**



The dual-modulus cycle is encoded in two integers T and S. In a period of T clock cycles, the phase will increment by FCW for S clock cycles, and FCW+1 for (T-S) clock cycles. The ratio S/T is computed as follows:

$$\frac{S}{T} = \lfloor FCW_{ideal} \rfloor + 1 - FCW_{ideal}$$

The core allows S, T and FCW to be specified to 32-bit precision. If the dual-modulus functionality is not required (because the FCW calculation is exact), S and T should be set to 0.

In the previous section (Carrier Setup and CCID Sequencing), the source code example for the `XDfeMix_AddCC` function showed a simple computation for the frequency control word based on f_{nco}=40 MHz and f_{clk} = 491.52 MHz. In this case, the value of FCW_{ideal} is not an integer, but 349525333⅓. Since the fractional part is rational (S/T = ⅔), the exact frequency of 40 MHz can be obtained by configuring the NCO as follows.

```
/* Declare NCO configuration structure */
XDfeMix_NCO NCOCfg;

NCOCfg.FrequencyCfg.FrequencyControlWord = 349525333; // Rounded down from
FCW_ideal
NCOCfg.FrequencyCfg.SingleModCount = 2; // The value of S
NCOCfg.FrequencyCfg.DualModCount = 1; // The value of (T-S)
```

It is also possible to set a phase offset for the NCO, as an unsigned 18-bit number representing a fraction of 2π.

```
NCOCfg.FrequencyCfg.PhaseOffset.PhaseOffset = 0x10000; // π/2 phase offset
```

The NCO hardware also incorporates a gain setting which can be modified as follows.

```
NCOCfg.NCOGain = 1; // 0 = 0dB, 1 = -3dB, 2 = -6dB, 3 = -9dB.
```

The `XDfeMix_NCO` structure is used by all functions that manipulate the NCO parameters. It also contains a sub-structure of type `XDfeMix_Phase` which allows the phase accumulator state to be captured and restored.

Antenna Gain

In the downlink operating mode, the API allows an optional gain adjustment setting (either 0 dB or -6 dB) to be applied on a per-antenna basis. This gain adjustment is not available in uplink mode.

The `XDfeMix_UpdateAntenna` function operates in the same way as the `XDfeMix_UpdateCC` function. An example code sequence is shown below. It assumes that the CC update trigger has already been configured as shown in previous examples. See Making Simultaneous Updatesfor details of how to set the gain for multiple antennas using a single trigger.

```
u32 AntID;
u32 Gain;

/* Clear the event status */
XDfeMix_ClearEventStatus(InstancePtr);

/* Set 0dB gain for antenna 0 */
AntID = 0;
Gain  = 1; // 1 = 0dB
XDfeMix_SetAntennaGain(&InstancePtr, AntID, Gain);

/* ... not shown - poll for CC update and/or wait for trigger to occur ...
```

```
*/

/* Clear the event status */
XDfeMix_ClearEventStatus(InstancePtr);

/* Set -6dB gain for antenna 1 */
AntID = 1;
Gain  = 0; // 0 = -6dB
XDfeMix_SetAntennaGain(&InstancePtr, AntID, Gain);
```

***Note*:** Only one antenna's gain can be modified per trigger event when using this function.

Making Simultaneous Updates

The API provides additional functions and configuration structures to allow updates for multiple component carriers and antennas to be triggered simultaneously. All details of the trigger setup (including the triggering latency) are the same as when performing individual updates as described in the previous section.

The overall sequence of events for making multiple simultaneous updates is as follows. First an `XDfeMix_CCCfg` structure is declared to hold the details of the core's configuration. This can be populated either with the configuration currently in effect or with an empty configuration setup. In both cases, the current state of the core is unaffected and all data processing continues uninterrupted.

The API provides functions to add, remove or update component carriers within this configuration, and to alter antenna parameters. When all changes to the configuration structure are complete, it is scheduled to be put into effect by a final API call.

The initial configuration structure is declared and populated with code similar to the following.

```
/* Declare configuration structure */
XDfeMix_CCCfg CCConfig;

/* Retrieve the current configuration parameters */
XDfeMix_GetCurrentCCCfg(InstancePtr, &CCConfig);
```

To start from a blank configuration instead of the current configuration, use the function `XDfeMix_GetEmptyCCCfg` instead of `XDfeMix_GetCurrentCCCfg`. The arguments and syntax are identical for the two calls. An empty configuration is initialized to 0, and therefore has no active component carriers.

To add a component carrier to the configuration, use the `XDfeMix_AddCCtoCCCfg` function. Its arguments and usage are almost identical to `XDfeMix_AddCC` but it operates on an `XDfeMix_CCCfg` structure rather than directly on the core itself. To remove a component carrier from the configuration, use `XDfeMix_RemoveCCfromCCCfg`.

```
XDfeMix_CarrierCfg CarrierCfg;
XDfeMix_NCO NCO;
u32 BitSequence;
u32 CCID;
u32 result;
double FreqMhz;
double NcoFreqMhz;
double FrequencyControlWord;

CarrierCfg.DUCDDCCfg.NCOIdx = 1; // Use NCO 1
CarrierCfg.DUCDDCCfg.CCGain = 2; // -6dB output gain for this CC
NCO.NCOGain = 1;                 // -3dB NCO gain for this CC

FreqMhz = 76.8;                  // Mix to 76.8 MHz
NcoFreqMhz = 491.52;             // System clock is 491.52 MHz
FrequencyControlWord = floor((FreqMhz / NcoFreqMhz) * 0x80000000);
NCO.FrequencyCfg.FrequencyControlWord = FrequencyControlWord;

BitSequence = 0x5555;        // CC will occupy every other timeslot,
starting with the first
CCID = 0;                    // CCID number chosen for the new carrier

/* Add the component carrier to the configuration */
result = XDfeMix_AddCCtoCCCfg(InstancePtr, &CCConfig, CCID, BitSequence,
&CarrierCfg, &NCO);

/* Remove CCID 9 from the configuration */
XDfeMix_RemoveCCfromCCCfg(InstancePtr, &CCConfig, 9);
```

The CC parameters for an existing component carrier can be updated within the configuration using `XDfeMix_UpdateCCinCCCfg`. It is also possible to retrieve the parameters for an existing component carrier using `XDfeMix_GetCarrierCfgAndNCO`. The code below shows how the these functions can be used.

```
/* Retrieve the carrier and NCO configuration for CCID 3 */
XDfeMix_GetCarrierCfgAndNCO(InstancePtr, &CCConfig, 3, &BitSequence,
&CarrierCfg, &NCO);

/* Update the gain for CCID 3 */
CarrierCfg.DUCDDCCfg.CCGain = 3;  // 0dB output gain
XDfeMix_UpdateCCinCCCfg(InstancePtr, &CCConfig, 3, &CarrierCfg);
```

RECOMMENDED: *To modify the NCO used by a carrier, use the `XDfeMix_MoveCC` function instead of altering `CarrierCfg.DUCDDCCfg.NCOIdx` in a CC update.*

The API provides the function `XDfeMix_SetAntennaCfgInCCCfg` to set the gain for multiple antennas within the configuration structure. This function is used as follows.

```
/* Define an antenna configuration structure */
XDfeMix_AntennaCfg AntConfig;

/* Populate the structure */
for (int ant = 0; ant < NUM_ANTS; ant++) {
  AntConfig.Gain[ant] = 1; // Set 0dB gain for all antennas
}

/* Apply the antenna configuration */
XDfeMix_SetAntennaCfgInCCCfg(InstancePtr, &CCConfig, &AntConfig);
```

When all the require changes have been made and the configuration structure is ready to be applied, call the `XDfeMix_SetNextCCCfgAndTrigger` function to write the new configuration to the core.

```
/* Apply the changes */
result = XDfeMix_SetNextCCCfgAndTrigger(InstancePtr, &CCConfig);

if (result == XST_SUCCESS)
  printf("CC configuration scheduled for update\n");
else
  printf("Set CC configuration failed with error code %d!\n", result);
```

The new component carrier and antenna configuration is activated when the specified trigger condition occurs.

Status, Events, and Interrupts

The control logic within the RFSoC DFE DUC-DDC Mixer core performs continuous monitoring for error events and records them in status registers. The API provides this error information, along with other status signals generated by the core. A complete set of interrupt enable, disable, masking and clearing functions is provided to support designs where the optional IRQ signal is connected to the control microprocessor.

Status information includes:

- **Arithmetic overflow in DUC/DDC:** The core records whether overflows have occurred on the real or imaginary part of the filter datapath, and at which stage of the half-band cascade the overflow was seen. The antenna and CCID where overflow occurred are recorded. If overflow occurs on more than one antenna or CCID before the event status is interrogated, only the lowest-numbered antenna or CCID is recorded.
- **Arithmetic overflow in Mixer:** The core records the lowest-numbered CCID and antenna on which overflow occurred within the mixer datapath.
- **Arithmetic overflow in Antenna Adder:** The core records which antenna adder stage recorded an overflow, and the lowest-numbered antenna on which overflow occurred within the antenna adder datapath.

- **CC sequence errors:** When in downlink mode, if the input TID sequence does not match the configuration programmed into the core via the API, an error is flagged.
- **CC updated triggered:** When a component carrier's configuration is scheduled to be updated based on a user-defined trigger, this status information confirms whether the trigger condition was met.

The use of the `XDfeMix_GetEventStatus` and `XDfeMix_ClearEventStatus` functions to check the CC sequence error and CC update trigger status was described in the previous sections. The same mechanism is used to obtain information about overflow events. Details of the exact location of the overflow can be retrieved using the functions `XDfeMix_GetMixerStatus` and `XDfeMix_GetDUCDDCStatus`, as shown by the following code.

```
/* Declare a status structure */
XDfeMix_Status status;

/* Query the core status */
XDfeMix_GetEventStatus(InstancePtr, &status);

if (status.DUCDDCOverflow != 0) {
  XDfeMix_DUCDDCStatus DUCDDCstatus;
  XDfeMix_GetDUCDDCStatus(InstancePtr, 0, &DUCDDCstatus); // second
argument is reserved

  printf("DUC/DDC Overflow: real stage %d, imag stage %d, CCID #%d, antenna
%d\n",
    DUCDDCstatus.RealOverflowStage, DUCDDCstatus.ImagOverflowStage,
    DUCDDCstatus.FirstCCIDOverflowing,
DUCDDCstatus.FirstAntennaOverflowing);

} else if (status.MixerOverflow != 0) {
  XDfeMix_MixerStatus Mixerstatus;
  XDfeMix_GetMixerStatus(InstancePtr, 0, &Mixerstatus); // second argument
is reserved

  printf("Mixer/Adder Overflow: adder stage %d, adder antenna %d, CCID #%d,
antenna %d\n",
    Mixerstatus.AdderStage, Mixerstatus.AdderAntenna,
    Mixerstatus.MixCCID, Mixerstatus.MixAntenna);
}
```

An interrupt masking function is provided to support designs where the optional IRQ signal is connected to the control microprocessor, or to other logic such as an integrated logic analyzer core. The following code demonstrates how to choose which of the events will generate an interrupt on the IRQ output of the core.

```
/* Declare an interrupt mask structure */
XDfeMix_InterruptMask IrqMask;

/* Populate the structure */
IrqMask.DUCDDCOverflow   = 0; // Enable (don't mask) DUC/DDC overflow
interrupts
IrqMask.MixerOverflow    = 0; // Enable (don't mask) Mixer overflow
interrupts
IrqMask.CCUpdate         = 1; // Disable (mask) CC update interrupts
IrqMask.CCSequenceError  = 0; // Enable (don't mask) CC sequence error
interrupts

/* Set up the interrupt mask */
XDfeMix_SetInterruptMask(InstancePtr, &IrqMask);
```

Whenever an event occurs, the corresponding bit will be set in the event status register and an interrupt will be generated by driving IRQ high unless the event has been masked as shown above. If the same event then re-occurs before `XDfeMix_ClearEventStatus` has been called, the status bit and the IRQ state will remain high. The IRQ line will not be driven low again until the event has been cleared.

Low Power Mode

For certain use cases such as time-division duplex (TDD) operation, the RFSoC DFE DUC-DDC Mixer core can be placed periodically into a low-power state. In this state the DUC/DDC and Mixer datapath functions are disabled and the data output from the core will not change. The control logic remains operational and the core continues to track the CCID sequence and produce TUSER and TLAST signals on the output. Internally, the filter delay pipelines and NCO phase accumulator state are frozen and do not change while the core is in low power mode.

Entry into and exit from low power mode are expected to be timed using triggers similar to those used for CC updates. The most likely scenario is a continuous trigger that recurs automatically without any need for software intervention between events. The following code shows an example of how to set up a periodic low-power transition.

```
/* Declare and populate trigger configuration */
XDfeMix_TriggerCfg TriggerCfg;

TriggerCfg.LowPower.Mode = 2;             // Continuous trigger
TriggerCfg.LowPower.TUSERBit = 7;         // on bit 7
TriggerCfg.LowPower.TuserEdgeLevel = 0;   // when low

/* Put the trigger into effect */
XDfeMix_SetTriggersCfg(InstancePtr, &TriggerCfg);
```

Following the call to `XDfeMix_SetTriggersCfg`, the core will enter the low power state whenever bit 7 of `s_axis_dl_din_tuser` (or `s_axis_ul_din_tuser`) is 0 and will return to normal operation whenever the bit returns to 1.

To achieve the lowest possible power consumption from the core, it must be deactivated using the `XDfeMix_Deactivate` function. This places all the logic into a quiescent state and disables the clock to the underlying primitives. To reactivate the core, use the `XDfeMix_Activate` function following the procedure described in the Initialization section.

Design Flow Steps

This section describes customizing and generating the core, constraining the core, and the simulation, synthesis, and implementation steps that are specific to this IP core. More detailed information about the standard Vivado® design flows and the IP integrator can be found in the following Vivado Design Suite user guides:

- *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
- *Vivado Design Suite User Guide: Designing with IP* (UG896)
- *Vivado Design Suite User Guide: Getting Started* (UG910)
- *Vivado Design Suite User Guide: Logic Simulation* (UG900)

Customizing and Generating the Core

This section includes information about using Xilinx® tools to customize and generate the core in the Vivado® Design Suite.

If you are customizing and generating the core in the Vivado IP integrator, see the *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994) for detailed information. IP integrator might auto-compute certain configuration values when validating or generating the design. To check whether the values do change, see the description of the parameter in this chapter. To view the parameter value, run the `validate_bd_design` command in the Tcl console.

You can customize the IP for use in your design by specifying values for the various parameters associated with the IP core using the following steps:

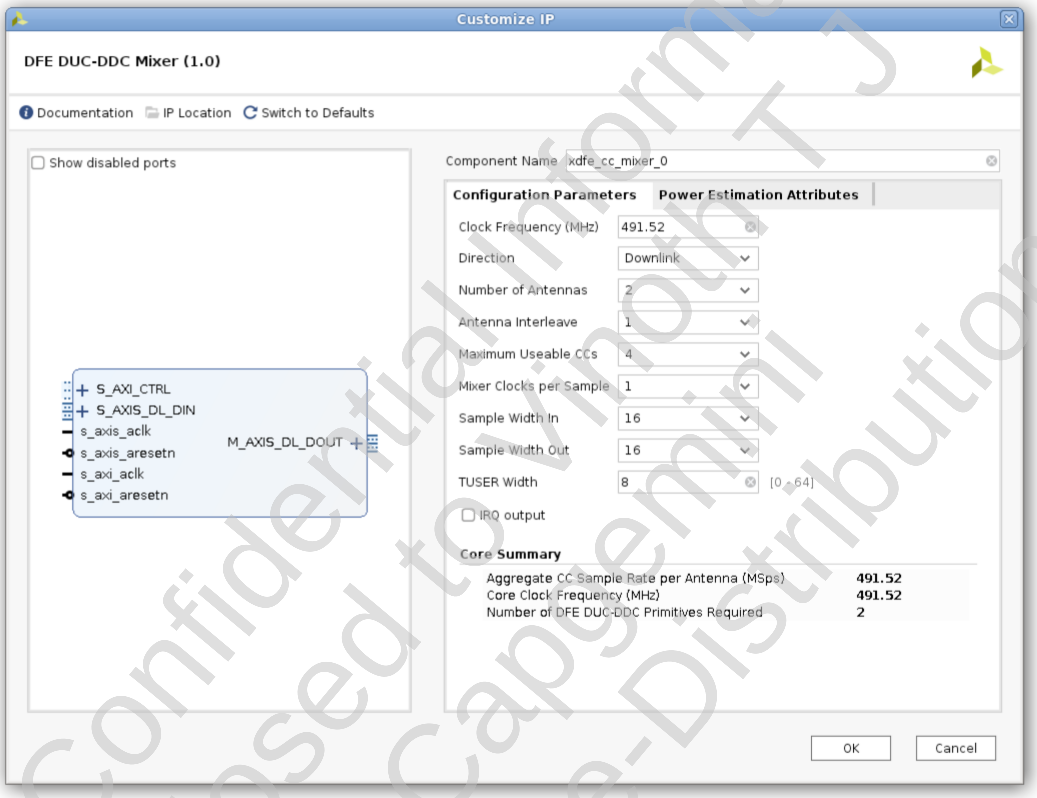
1. Select the IP from the IP catalog.
2. Double-click the selected IP or select the Customize IP command from the toolbar or right-click menu.

For details, see the *Vivado Design Suite User Guide: Designing with IP* (UG896) and the *Vivado Design Suite User Guide: Getting Started* (UG910).

Figures in this chapter are illustrations of the Vivado IDE. The layout depicted here might vary from the current version.

Configuration Tab

Figure 10: **Configuration Tab**



Component Name: xdfe_cc_mixer_0

Clock Frequency (MHz)	491.52
Direction	Downlink
Number of Antennas	2
Antenna Interleave	1
Maximum Useable CCs	4
Mixer Clocks per Sample	1
Sample Width In	16
Sample Width Out	16
TUSER Width	8 [0 - 64]

☐ IRQ output

Core Summary

Aggregate CC Sample Rate per Antenna (MSps)	491.52
Core Clock Frequency (MHz)	491.52
Number of DFE DUC-DDC Primitives Required	2

- **Clock Frequency**: The frequency of `s_axis_aclk`, at which the datapath will operate, determines the total sample rate per antenna.
- **Mode:** The core can be configured in either downlink (transmit) mode or uplink (receive) mode.
- **Number of Antennas:** The core can provide filtering for 1, 2, 4, or 8 antennas in parallel.
- **Antenna Interleave:** This factor specifies the number of clock cycles across which the samples for each antenna are transferred on the downlink input and uplink output interfaces.

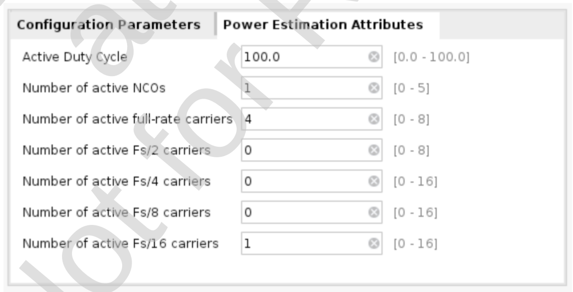
  For example, if the number of antennas is eight, an antenna interleave of 1 will transfer that data for all eight antennas in parallel in a single clock cycle. At the other extreme, an antenna interleave of 8 will transfer the data for a single antenna on each clock cycle, and it takes eight cycles in total to transfer the data for all antennas. The antenna interleave cannot be larger than the number of antennas.

- **Mixer Clocks Per Sample:** The mixer stage can be configured to operate at the same rate as the DUC-DDC stage by setting this parameter to 1 (the default). It can also be configured to operate at half the rate or a quarter of the rate of the DUC-DDC stage by setting this parameter to 2 or 4 respectively. This is useful if the total instantaneous bandwidth of the system is less than the core clock frequency, and power/area savings are required.
- **Maximum Useable CCs:** This parameter sets the maximum number of carriers that can be processed simultaneously by the core. Valid values are 2, 4, and 8, depending on the number of antennas and the antenna interleave.
- **Sample Width In:** The number of bits used to represent the I and Q components of the input data samples. Switchable between 16 (the default) and 18 bits.
- **Sample Width Out:** The number of bits used to represent the I and Q components of the output data samples. Switchable between 16 (the default) and 18 bits.
- **TUSER Width:** The number of bits in the TUSER buses associated with each AXI4-Stream interface. Valid values are 0 (for no TUSER bus) up to 64 bits. The default is 8 bits.
- **IRQ Output:** Determines whether the core should include an IRQ output port to connect to the interrupt controller of the microprocessor. By default, no IRQ output port is provided.

Power Estimation Tab

The RFSoC DFE DUC-DDC Mixer core supports power estimation. The Power Estimation Attributes tab in the IP customization dialog, shown below, allows power estimation parameters to be specified. The values set for these fields have no effect on the functionality of the core.

Figure 11: **Power Estimation Attributes Tab**



Active Duty Cycle	100.0	[0.0 - 100.0]
Number of active NCOs	1	[0 - 5]
Number of active full-rate carriers	4	[0 - 8]
Number of active Fs/2 carriers	0	[0 - 8]
Number of active Fs/4 carriers	0	[0 - 16]
Number of active Fs/8 carriers	0	[0 - 16]
Number of active Fs/16 carriers	1	[0 - 16]

- **Active Duty Cycle:** The duty cycle of the core as a percentage, where 100% means the core is always operational and 0% means the core is always deactivated.
- **Number of active NCOs:** The number of NCOs per hardware primitive that are active. The same setting is applied to all underlying DFE_DUC_DDC primitives within the core.
- **Number of active carriers:** For each possible rate from $F_S/16$ up to F_S, the number of carriers of the given rate that are active per hardware primitive. The same setting is applied to all underlying DFE_DUC_DDC primitives within the core.

User Parameters

The following table shows the relationship between the fields in the Vivado® IDE and the user parameters (which can be viewed in the Tcl Console).

Table 2: **User Parameters**

Vivado IDE Parameter	User Parameter	Default Value
Clock Frequency (MHz)	CLOCK_FREQUENCY	491.52
Mode	MODE	downlink
Number of antennas	NUM_ANTENNA	1
Antenna interleave	ANTENNA_INTERLEAVE	1
Maximum useable CCs	MAX_USEABLE_CCIDS	4
Mixer clocks per sample	MIXER_CPS	1
Sample width in	SAMPLE_WIDTH_IN	16
Sample width out	SAMPLE_WIDTH_OUT	16
TUSER width	TUSER_WIDTH	8
IRQ output	HAS_IRQ	false

Output Generation

For details, see the *Vivado Design Suite User Guide: Designing with IP* (UG896).

Constraining the Core

Required Constraints

This section is not applicable for this IP core.

Device, Package, and Speed Grade Selections

This section is not applicable for this IP core.

Clock Frequencies

This section is not applicable for this IP core.

Clock Management

This section is not applicable for this IP core.

Clock Placement

This section is not applicable for this IP core.

Banking

This section is not applicable for this IP core.

Transceiver Placement

This section is not applicable for this IP core.

I/O Standard and Placement

This section is not applicable for this IP core.

Simulation

For comprehensive information about Vivado® simulation components, as well as information about using supported third-party tools, see the *Vivado Design Suite User Guide: Logic Simulation* (UG900).

Synthesis and Implementation

For details about synthesis and implementation, see the *Vivado Design Suite User Guide: Designing with IP* (UG896).

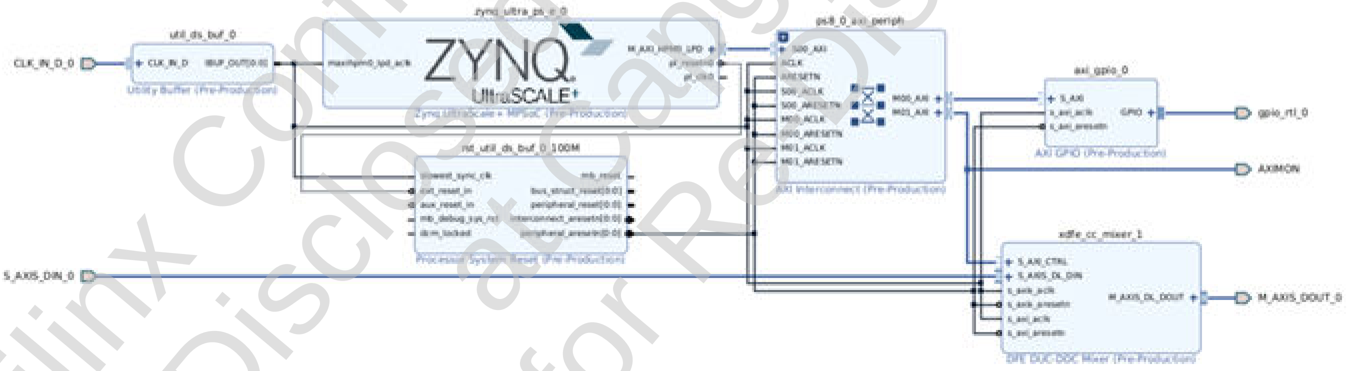
Example Design

This chapter contains information about the RFSoC DFE DUC-DDC Mixer example design provided in the Vivado® Design Suite. Note that the example design is provided for simulation purposes only. Synthesis, implementation, and bitstream generation are not supported.

Overview

The following figure shows the structure of the example design as viewed in the Vivado® IP integrator.

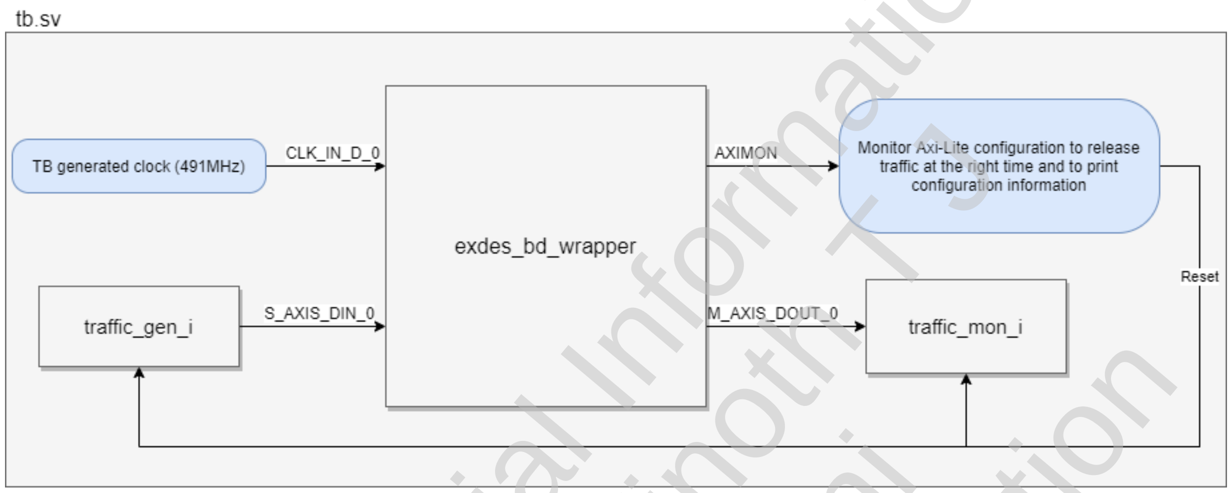
Figure 12: **Example Design**



The example design block diagram contains an instance of the RFSoC DFE DUC-DDC Mixer core along with a Zynq® processing system and associated AXI infrastructure to run the example software application which configures and controls the core.

A test bench is provided with the example design to provide clocks and resets, drive stimulus data into the core, and check the output data. The test bench structure is illustrated below.

Figure 13: **DFE DUC-DDC/Mixer Example Design Test Bench**


tb.sv
TB generated clock (491MHz)
CLK_IN_D_0
exdes_bd_wrapper
AXIMON
Monitor Axi-Lite configuration to release traffic at the right time and to print configuration information
Reset
traffic_gen_i
S_AXIS_DIN_0
M_AXIS_DOUT_0
traffic_mon_i

The test bench consists of a traffic generator and traffic monitor for the AXI4-Stream interfaces, along with an AXI4-Lite monitor component which monitors register writes to the core and ensures that the test bench provides appropriate stimulus at the correct time.

Running the Example Design

After the example design project has been created, the output products and HDL wrapper will be automatically generated. Some example design parameters are inherited from the core instance from which the example design was generated. The example design supports uplink and downlink configurations, with an antenna interleave of 1 or 2. The number of antennas will always be set to 2. Other configuration parameters chosen when the core was originally customized will be ignored.

The QEMU emulator software requires an ELF binary executable file to be specified for operation. The RFSoC DFE DUC-DDC Mixer example design includes an example software program which makes use of the core's software driver and demonstrates the use of the API functions to configure and control the core. This program will read the configuration of the core and adapt its operation based on the parameters. The example software is provided both as a precompiled ELF file for use with QEMU.

When the example design is first generated, QEMU is launched automatically. In this case, skip to step 3 in the sequence below. Otherwise, after re-opening the project or when restarting the simulation, launch QEMU manually using the scripts provided following steps 1 and 2 below.

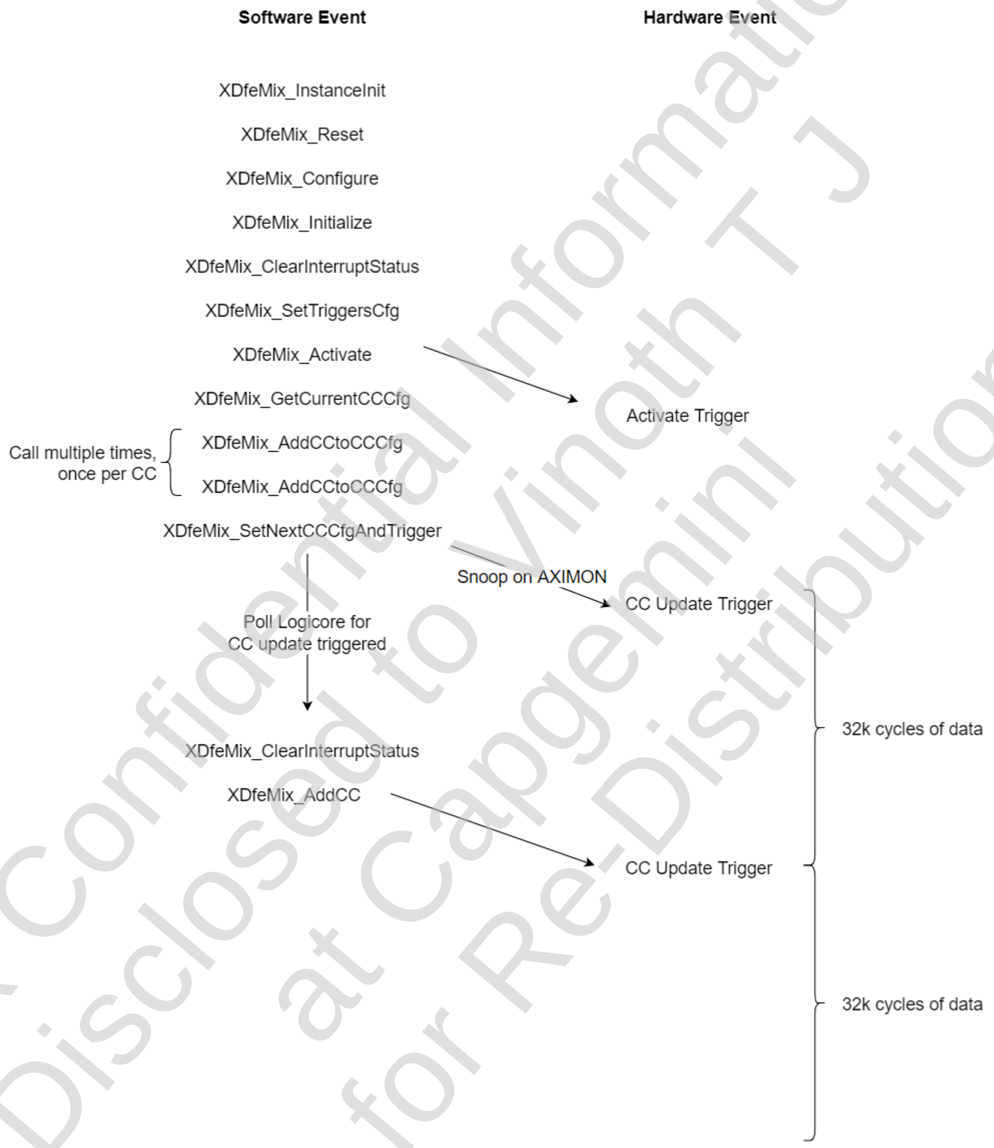
1. Ensure the current working directory of your shell is the `imports` directory within the example design project directory using `cd <proj_dir>/imports`.

2. To launch QEMU using the default configuration-agnostic ELF file that ships with the example design, execute the command `./start_qemu.sh -elf elf_file.txt`.
3. To run the simulation, select **SIMULATION → Run Simulation → Run Behavioral Simulation** from the Vivado Flow Navigator panel.
4. Click **Run All** to run the simulation. The sequence of events in the example design simulation is described below.

***Note*:** To re-run the simulation, the simulation window must be closed to end the current QEMU processes. QEMU must then be re-started and the simulation run as described above. It is not possible to use the Restart operation within the simulator directly. Should the simulation terminate unnaturally, the QEMU process may not automatically terminate. In this case, manually terminate QEMU before attempting to start a new simulation.

Once the simulation starts, the processor will configure and activate the core, and set up the first and second component carriers in the sequence. The simulation environment monitors the AXI4-Lite interface and determines the right time to start the stimulus by waiting for the CC trigger configuration to be written. After the flow of stimulus data has started, the processor will add a third component carrier to the CC sequence. The software API functions used and their relationship to events in the hardware are illustrated in the following figure.

Figure 14: **DFE DUC-DDC/Mixer Example Design Sequence of Operations**



The default ELF file provided with the example design is agnostic to the core parameters. These parameters are read dynamically by the program during simulation. Depending on whether the core is customized for uplink or downlink operation, a different set of values will be used for the component carrier configuration:

- In downlink mode, NCOs 0,1 and 2 run at 122.88 MHz, 276.48 MHz, and 368.64 MHz, respectively.
- In uplink mode, NCOs 0, 1 and 2 run at 368.64 MHz, 215.04 MHz, and 122.88 MHz, respectively.

The uplink and downlink frequencies are complementary, such that the downlink output after carrier combining can be fed into the uplink version of the design to recover the original carrier data.

The CCID sequence programming is the same in both the downlink and the uplink. CCID 0 always occupies 1/2 of the CCID slots. CCID 1 occupies 1/8 of the slots. When CCID 2 is added, it occupies 1/4 of the slots. The diagrams below show the CC sequences for the first 32k clock cycles and the second 32k clock cycles.

Figure 15: **CCID Sequence for the First 32k Cycles**

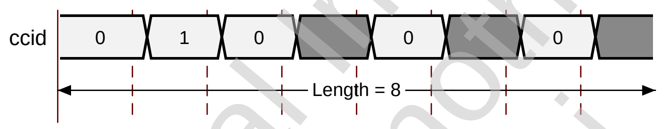
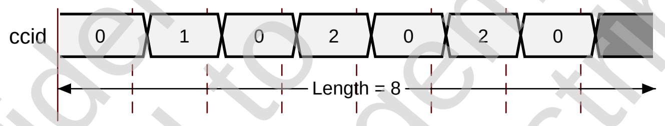


Figure 16: **CC sequence #2**



Example Design Data Files

The example design project includes a number of data files containing stimulus data for the RFSoC DFE DUC-DDC Mixer core. Which text file the test bench reads depends on the core parameters. Data and TID text files are named by convention `<dl/ul>_a2_ilv<1/2>_din_<data/tid>*.txt`. The data and control samples in these files are driven onto the core inputs by the stimulus generator component.

The files contain vectors that exercise the core datapath for 64k clock cycles, with a sequence reconfiguration happening mid-way through. The first 32k data cycles correspond to the initial sequence configuration with two active CCs, and the second 32k data cycles correspond to the final sequence configuration with three active CCs.

The data corresponding to Antenna 0 is generated by filtering a white noise signal with an 80% bandpass. The data corresponding to Antenna 1 are constant per CCID, as an aid to visualization of the data flow through the system.

The output `TDATA` and `TID` signals are monitored and compared to a golden reference which is generated using the core's C model. The golden reference data is provided in the form of text files named `gold_<ul/dl>_<data/tid>_*.txt`. The test bench counts any errors in the core outputs and logs these to the console, using the results of the comparison to decide whether the test passed or failed.

Modifying the Example Program

The `run_vitis.tcl` script is provided to create a Vitis™ project and generate an ELF file for use with QEMU. The C source code provided for the Vitis project is the same as that used to generate the default, configuration-agnostic ELF file that is delivered with the example design. The script should be run through the Vivado Tcl console using the command `source run_vitis.tcl`.

Vitis console outputs are displayed in the terminal window from which Vivado is launched. The Vitis project is created in the `vitis_workspace` directory within the example design project's root directory. Once this script has run and the Vitis project has been generated, it is recommended to use the Vitis GUI for subsequent C code updates.

To launch QEMU using a custom ELF file generated in this way, use the ELF file `../vitis_workspace/<core_name>_app/Debug/<core_name>_ex_app.elf` on the command line in place of the default ELF file name when launching QEMU.

Note: Modifying the example software source code may require corresponding changes in the test bench and in the stimulus and golden reference data files, if the CC parameters or sequence are altered.

C Model

The RFSoC DFE DUC-DDC Mixer bit-accurate C model is a self-contained, linkable, shared library that models the functionality of this core. The model performs up-conversion, down-conversion, NCO mixing and carrier combining using the same internal fixed-point arithmetic as the IP core and produces bit-accurate results. It is suitable for inclusion in a larger framework for system-level simulation or core-specific verification.

Features

- Bit accurate with the RFSoC DFE DUC-DDC Mixer core.
- Supports 64-bit Linux and 64-bit Windows platforms.
- Transaction-based interface.

The DFE DUC-DDC mixer bit-accurate C model does not model flow control, latency, or other timing-related aspects of the RFSoC DFE DUC-DDC Mixer IP core's operation.

Overview

The model consists of a set of C functions that reside in a shared library. Example C code demonstrates how these functions form the interface to the C model. Full details of this interface are given in the C Model Interface section. An example piece of source code showing how to call the model is provided. The model is also available as a MATLAB® software MEX function for seamless MATLAB software integration.

The model is bit accurate but not cycle-accurate; it produces exactly the same output data as the core on a sample-by-sample basis. However, it does not model the core latency or its interface signals. Bit-accurate behavior can be guaranteed while dynamically changing the carrier configuration, provided that the switch-over point is matched to the same sample boundary in the model as in the hardware.

The flushing of hardware sample buffers is not modeled. Buffer flushing will cause the hardware and the model to mismatch temporarily until sufficient new samples are provided to clear the effect of the flush operation from the processing pipeline.

Unpacking and Model Contents

Unzip the DFE DUC-DDC Mixer C model zip file for your platform. This produces the directory structure and files shown in the following table.

Table 3: **C Model Zip File Contents: Linux**

File	Description
xdfe_cc_mixer_v1_0_bitacc_cmodel.h	Header file which defines the DFE DUC-DDC Mixer C model API.
xip_vector_array_bitacc_cmodel.h	Header file with vector array type definition.
xip_common_bitacc_cmodel.h	Header file with common definitions.
libIP_xdfe_cc_mixer_v1_0_bitacc_cmodel.so	Model shared object library.
run_bitacc_cmodel.c	Example program for calling the C model.
xdfe_cc_mixer_v1_0_bitacc_mex.cpp	MATLAB MEX function source.
@xdfe_cc_mixer_v1_0_bitacc	MATLAB MEX function class directory.
gmp.h	GNU Multiple-precision library header.
libgmp.so.11	GNU Multiple-precision library.
libgmpxx.so.4	GNU Multiple-precision library for C++.

Table 4: **C Model Zip File Contents: Windows**

File	Description
xdfe_cc_mixer_v1_0_bitacc_cmodel.h	Header file which defines the DFE DUC-DDC Mixer C model API.
xip_vector_array_bitacc_cmodel.h	Header file with vector array type definition.
xip_common_bitacc_cmodel.h	Header file with common definitions.
libIp_xdfe_cc_mixer_v1_0_bitacc_cmodel.dll	Model dynamically-linked library.
libIp_xdfe_cc_mixer_v1_0_bitacc_cmodel.lib	Model statically-linked library.
run_bitacc_cmodel.c	Example program for calling the C model.
xdfe_cc_mixer_v1_0_bitacc_mex.cpp	MATLAB MEX function source.
@xdfe_cc_mixer_v1_0_bitacc	MATLAB MEX function class directory.
gmp.h	GNU Multiple-precision library header.
libgmp.dll	GNU Multiple-precision dynamically-linked library.
libgmp.lib	GNU Multiple-precision statically-linked library.

Installation

Linux

For Linux, follow these steps:

- Unzip the contents of the ZIP file.
- Ensure that the directory containing the files `libIp_dfe_cc_mixer_v1_0_bitacc_cmodel.so`, `libgmp.so.11`, and `libgmpxx.so.4` is in your $LD_LIBRARY_PATH environment variable.

Windows

For Windows, follow these steps:

- Unzip the contents of the ZIP file.
- Ensure that the directory containing the files `libIp_dfe_cc_mixer_v1_0_bitacc_cmodel.dll` and `libgmp.dll` is:
  - Either in your %PATH% environment variable
  - Or, the directory from which you run your executable that calls the DUC-DDC mixer C model.

C Model Interface

The API of the C model is defined in the header file `xdfe_cc_mixer_v1_0_bitacc_cmodel.h`. This interface consists of data structures and functions as described in the following sections.

To use the C model:

- Create a model configuration structure and initialize the values within it according to your application.
- Use this model configuration structure to create an instance of the model.
- Create an antenna configuration structure, which contains an array of CC configuration structures.
- Initialize the CC configuration structures for each component carrier in your system.
- Initialize arrays for input and output samples. Allocate memory and populate the input array.
- Call the 'do'-function of the model to process blocks of data for all component carriers, passing in a pointer to the antenna configuration structure.
- When finished, destroy the model instance and deallocate memory as necessary.

The C model supports up to eight NCOs each represented in the antenna configuration structure however the C model does not support component carrier to NCO abstraction. Therefore the configuration of the component carrier is attached to the NCO in the antenna configuration structure. Inputs or outputs for a component carrier should be passed or taken from the NCO the component carrier would be assigned to.

When configuring the NCO for each carrier, the frequency control word and single/dual modulus cycle count parameters must be calculated to ensure the NCO behavior is modeled correctly. A helper function `xdfe_cc_mixer_v1_0_calc_freq_config` is provided to calculate the correct NCO configuration values for the desired frequency.

One call to `xdfe_cc_mixer_v1_0_do` will process an arbitrary number of samples for all component carriers on one antenna. When in uplink mode the minimum number of samples that can be processed is equal to the configured highest rate. For each additional antenna a separate instance of the model will need to be created. The `xdfe_cc_mixer_v1_0_set_ant_cfg` function loads a specific antenna configuration structure into the model. Optionally this can also be done by passing an antenna configuration structure when`xdfe_cc_mixer_v1_0_do` is called. If the antenna configuration structure is not passed in then the model will used the last antenna configuration it received.

The DUC-DDC Mixer C model can be configured in uplink or downlink mode. To perform both uplink and downlink data processing simultaneously, create two instances of the model where one is configured for uplink operation and the other is configured for downlink operation.

An example C++ file called `run_bitacc_cmodel.c` is included in the ZIP file. This demonstrates how to call the DUC-DDC Mixer C model. Refer to this file for examples of using the following interface described.

Constants

The following constants are defined by the API:

Table 5: **Constants**

Name	Value
Model Configuration	
XDFE_CC_MIXER_V1_0_UPLINK	0
XDFE_CC_MIXER_V1_0_DOWNLINK	1
XDFE_CC_MIXER_V1_0_UTILISE_MIXER_UP	0
XDFE_CC_MIXER_V1_0_BYPASS_MIXER_UP	1
XDFE_CC_MIXER_V1_0_UTILISE_DDC_DN	0
XDFE_CC_MIXER_V1_0_BYPASS_DCC_DN	1
Supported Sample Widths	
XDFE_CC_MIXER_V1_0_SAMPLE_WIDTH_16B	16
XDFE_CC_MIXER_V1_0_SAMPLE_WIDTH_18B	18
Maximum Component Carriers Per Antenna	
XDFE_CC_MIXER_V1_0_MAX_NCO	8
NCO Sin/Cos Scaling	
XDFE_CC_MIXER_V1_0_NCO_GAIN_0DB	0
XDFE_CC_MIXER_V1_0_NCO_GAIN_MINUS3DB	1

Table 5: **Constants** *(cont'd)*

Name	Value
XDFE_CC_MIXER_V1_0_NCO_GAIN_MINUS6DB	2
XDFE_CC_MIXER_V1_0_NCO_GAIN_MINUS9DB	3
DUC-DDC Mixer Interpolation/Decimation Rates	
XDFE_CC_MIXER_V1_0_RATE_0X	0
XDFE_CC_MIXER_V1_0_RATE_1X	1
XDFE_CC_MIXER_V1_0_RATE_2X	2
XDFE_CC_MIXER_V1_0_RATE_4X	4
XDFE_CC_MIXER_V1_0_RATE_8X	8
XDFE_CC_MIXER_V1_0_RATE_16X	16
Component Carrier Scaling	
XDFE_CC_MIXER_V1_0_CC_GAIN_MINUS18DB	0
XDFE_CC_MIXER_V1_0_CC_GAIN_MINUS12DB	1
XDFE_CC_MIXER_V1_0_CC_GAIN_MINUS6DB	2
XDFE_CC_MIXER_V1_0_CC_GAIN_0DB	3
Antenna Scaling	
XDFE_CC_MIXER_V1_0_ANT_GAIN_MINUS6DB	0
XDFE_CC_MIXER_V1_0_ANT_GAIN_0DB	1

Data Types

The following types are defined by the API:

Table 6: **Data Type Definitions**

Name	Description
Standard types	
xdfe_cc_mixer_v1_0	Model object type (opaque to user)
xip_bit	Single bit type
xip_uint	Unsigned integer type
xip_int	Integer type
xip_status	Result code from API functions; XIP_STATUS_OK, XIP_STATUS_ERROR
xip_msg_handler	Message handler function signature
xip_real	Real scalar type
Structures	
xdfe_cc_mixer_v1_0_config	Model configuration parameters, see xdfe_cc_mixer_v1_0_config structure section
xdfe_cc_mixer_v1_0_frequency	NCO frequency configuration, see xdfe_cc_mixer_v1_0_frequency structure section
xdfe_cc_mixer_v1_0_phase	NCO phase accumulator state, see xdfe_cc_mixer_v1_0_phase structure section

Table 6: **Data Type Definitions** *(cont'd)*

Name	Description
xdfe_cc_mixer_v1_0_nco	NCO configuration, see xdfe_cc_mixer_v1_0_nco structure section
xdfe_cc_mixer_v1_0_cc_cfg	Component carrier configuration, see xdfe_cc_mixer_v1_0_cc_cfg structure section
xdfe_cc_mixer_v1_0_ant_cfg	Antenna configuration, see xdfe_cc_mixer_v1_0_ant_cfg structure section
xdfe_cc_mixer_v1_0_cc_status	Component carrier status, see xdfe_cc_mixer_v1_0_cc_status structure section
xdfe_cc_mixer_v1_0_status	Status, see xdfe_cc_mixer_v1_0_status structure section
Dynamic Arrays	
xip_vector_array_complex	Dynamic vector of arrays (of a standard type)

Functions

The C model provides the following functions:

xdfe_cc_mixer_v1_0_get_version

Gets the version of library.

Table 7: **Returns**

Type	Description
const char*	Textual representation of library version

xdfe_cc_mixer_v1_0_default_config

Gets default configuration structures.

Table 8: **Arguments**

Argument Name	Type	Description
config	xdfe_cc_mixer_v1_0_config*	

Table 9: **Returns**

Type	Description
xip_status	Exit code

xdfe_cc_mixer_v1_0_default_ant_cfg

Gets default antenna configuration structures.

Table 10: **Arguments**

Argument Name	Type	Description
ant_cfg	xdfe_cc_mixer_v1_0_ant_cfg*	

Table 11: **Returns**

Type	Description
xip_status	Exit code

xdfe_cc_mixer_v1_0_create

Gets default antenna configuration structures.

Table 12: **Arguments**

Argument Name	Type	Description
config	const xdfe_cc_mixer_v1_0_config*	Model configuration structure
msg_handler	xip_msg_handler	Callback function for errors and warnings
msg_handle	void*	Optional argument to be passed back to callback function

Table 13: **Returns**

Type	Description
xdfe_cc_mixer_v1_0*	Model instance handle

xdfe_cc_mixer_v1_0_reset

Resets an instance of the core.

Table 14: **Arguments**

Argument Name	Type	Description
model	xdfe_cc_mixer_v1_0*	Model instance handle

Table 15: **Returns**

Type	Description
xip_status	Exit code

xdfe_cc_mixer_v1_0_calc_freq_config

Calculates NCO frequency configuration

Table 16: **Arguments**

Argument Name	Type	Description
freq_nco	xip_real	Desired NCO frequency
freq_clk	xip_real	NCO clock frequency

Table 17: **Returns**

Type	Description
xdfe_cc_mixer_v1_0_frequency	NCO Frequency configuration

xdfe_cc_mixer_v1_0_set_ant_cfg

Sets antenna configuration.

Table 18: **Arguments**

Argument Name	Type	Description
model	xdfe_cc_mixer_v1_0*	Model instance handle
ant_cfg	const xdfe_cc_mixer_v1_0_ant_cfg*	Antenna configuration

Table 19: **Returns**

Type	Description
xip_status	Exit code

xdfe_cc_mixer_v1_0_update_ant_cfg

Updates an ant_cfg struct to reflect state of the mixer.

Table 20: **Arguments**

Argument Name	Type	Description
model	xdfe_cc_mixer_v1_0*	Model instance handle
ant_cfg	xdfe_cc_mixer_v1_0_ant_cfg*	Antenna configuration

Table 21: **Returns**

Type	Description
xip_status	Exit code

xdfe_cc_mixer_v1_0_cc_move_nco_update

Updates the relevant NCO to facilitate an NCO move

Table 22: **Arguments**

Argument Name	Type	Description
model	xdfe_cc_mixer_v1_0*	Model instance handle
from_nco_id	xip_uint	NCO ID to align phase from
to_nco_id	xip_uint	NCO ID to align phase to
ant_cfg	xdfe_cc_mixer_v1_0_ant_cfg*	ant cfg stuct to update. Optional argument, set to 0 (NULL). No ant_cfg returned when undefined.

Table 23: **Returns**

Type	Description
xip_status	Exit code

xdfe_cc_mixer_v1_0_get_status

Gets status, overflow cleared on read.

Table 24: **Arguments**

Argument Name	Type	Description
model	xdfe_cc_mixer_v1_0*	Model instance handle
status	xdfe_cc_mixer_v1_0_status*	Status output

Table 25: **Returns**

Type	Description
xip_status	Exit code

xdfe_cc_mixer_v1_0_do

Mix and rate change the supplied data.

Table 26: **Arguments**

Argument Name	Type	Description
model	xdfe_cc_mixer_v1_0*	Model instance handle
ant_cfg	const xdfe_cc_mixer_v1_0_ant_cfg*	Antenna configuration. Optional argument, set to 0 (NULL). Use current configuration when undefined.
max_nco	xip_uint	Maximum NCOs of the model. Valid values are 2, 4, and 8. In downlink, this is the number of input vectors expected. In uplink, this is the number of output vectors generated.
din	const xip_vector_array_complex*	Input data

Table 26: **Arguments** *(cont'd)*

Argument Name	Type	Description
dout	xip_vector_array_complex*	Output data
status	xdfe_cc_mixer_v1_0_status*	Overflow, or underflow, and status. Optional argument, set to 0 (NULL). Not read when undefined.

Table 27: **Returns**

Type	Description
xip_status	Exit code

xdfe_cc_mixer_v1_0_destroy

Destroys model instance and free any resources allocated.

Table 28: **Arguments**

Argument Name	Type	Description
model	xdfe_cc_mixer_v1_0*	Model instance handle

Table 29: **Returns**

Type	Description
xip_status	Exit code

Structures

The following structures are defined by the API:

xdfe_cc_mixer_v1_0_config

Field Name	Type	Description
name	const char*	Instance name
mode	xip_bit	Specify uplink or downlink data path
sample_width_in	xip_uint	Input sample width
sample_width_out	xip_uint	Output sample width

xdfe_cc_mixer_v1_0_frequency

Field Name	Type	Description
freq_control_word	xip_uint	Frequency control word (FCW)

Field Name	Type	Description
single_mod_count	xip_uint	Single modulus cycle count (S)
dual_mod_count	xip_uint	Dual modulus cycle count (T-S)
phase_offset	xip_uint	Phase offset

xdfe_cc_mixer_v1_0_phase

Field Name	Type	Description
phase_acc	xip_uint	Phase accumulator state
dual_mod_count	xip_uint	Phase accumulator dual modulus count
dual_mod_sel	xip_uint	Phase accumulator dual modulus selection value

xdfe_cc_mixer_v1_0_nco

Field Name	Type	Description
frequency	xdfe_cc_mixer_v1_0_frequency	Frequency configuration
phase	xdfe_cc_mixer_v1_0_phase*	Phase accumulator state. Set to NULL when not to be updated
gain	xip_uint	Specifies the scaling of NCO sin/cosine output

xdfe_cc_mixer_v1_0_cc_cfg

Field Name	Type	Description
nco	xdfe_cc_mixer_v1_0_nco	NCO configuration
rate	xip_uint	DUC-DDC Mixer rate
gain	xip_uint	CC gain specification

xdfe_cc_mixer_v1_0_ant_cfg

Field Name	Type	Description
gain	xip_uint	Antenna gain specification
cc	xdfe_cc_mixer_v1_0_cc_cfg[XDFE_CC_MIXER_V1_0_MAX_CC]	Array of CC configurations

xdfe_cc_mixer_v1_0_cc_status

Field Name	Type	Description
cm_ouflow	xip_int	Sample number of first overflow, and saturation, to occur in the complex multiplier

Field Name	Type	Description
ducddc_stg1_ouflow	xip_int	Sample number of first overflow, and saturation, to occur in the DUC-DDC Mixer stage 1
ducddc_stg2_ouflow	xip_int	Sample number of first overflow, and saturation, to occur in the DUC-DDC Mixer stage 2
ducddc_stg3_ouflow	xip_int	Sample number of first overflow, and saturation, to occur in the DUC-DDC Mixer stage 3
ducddc_stg4_ouflow	xip_int	Sample number of first overflow, and saturation, to occur in the DUC-DDC Mixer stage 4
nco_phase	xdfe_cc_mixer_v1_0_phase	NCO accumulator state

xdfe_cc_mixer_v1_0_status

Field Name	Type	Description
ant_add1_ouflow	xip_int	Sample number of first overflow, and saturation, to occur in the first adder stage
ant_add2_ouflow	xip_int	Sample number of first overflow, and saturation, to occur in the second adder stage
cc	xdfe_cc_mixer_v1_0_cc_status[XDFE_CC_MIXER_V1_0_MAX_CC]	Component carrier status

Compiling and Linking

Place the header files (`xdfe_cc_mixer_v1_0_bitacc_cmodel.h`, `xip_common_bitacc_cmodel.h`, `xip_vector_array_bitacc_cmodel.h`, and `gmp.h`) with other header files in your project.

Compilation varies from platform to platform.

Linux

To compile the example code, `run_bitacc_cmodel.c`, first ensure that the directory in which the files `libIp_xdfe_cc_mixer_v1_0_bitacc_cmodel.so`, `libgmp.so.11` and `libgmpxx.so.4` are located is present on your $LD_LIBRARY_PATH environment variable. These shared libraries are referenced during the compilation and linking process.

Place the header file and C++ source file in a single directory. Then in that directory, compile using the GNU C++ Compiler:

```
gcc -x c++ -I. -L. -lIp_xdfe_cc_mixer_v1_0_bitacc_cmodel -Wl,-rpath,. -o
run_bitacc_cmodel run_bitacc_cmodel.c
```

***Note*:** The C model dynamically links to `libgmpxx.so.4` and therefore this library must be visible to the model while running.

Windows

When compiling on Windows, the symbol 'NT' must be defined either by a compiler option or in the user source code before the `xdfe_cc_mixer_v1_0_bitacc_cmodel.h` header file is included.

Link to the import libraries `libIp_xdfe_cc_mixer_v1_0_bitacc_cmodel.lib` and `libgmp.lib`. For example, in Microsoft Visual Studio.NET, in Project Properties, under **Linker→ Input**, for Additional Dependencies, specify `libIp_xdfe_cc_mixer_v1_0_bitacc_cmodel.lib` and `libgmp.lib`.

Appendix A

Debugging

This appendix includes details about resources available on the Xilinx® Support website and debugging tools.

If the IP requires a license key, the key must be verified. The Vivado® design tools have several license checkpoints for gating licensed IP through the flow. If the license check succeeds, the IP can continue generation. Otherwise, generation halts with an error. License checkpoints are enforced by the following tools:

- Vivado Synthesis
- Vivado Implementation
- write_bitstream (Tcl command)

IMPORTANT! *IP license level is ignored at checkpoints. The test confirms a valid license exists. It does not check IP license level.*

Finding Help on Xilinx.com

To help in the design and debug process when using the core, the Xilinx Support web page contains key resources such as product documentation, release notes, answer records, information about known issues, and links for obtaining further product support. The Xilinx Community Forums are also available where members can learn, participate, share, and ask questions about Xilinx solutions.

Documentation

This product guide is the main document associated with the core. This guide, along with documentation related to all products that aid in the design process, can be found on the Xilinx Support web page or by using the Xilinx® Documentation Navigator. Download the Xilinx Documentation Navigator from the Downloads page. For more information about this tool and the features available, open the online help after installation.

Technical Support

Xilinx provides technical support on the Xilinx Community Forums for this LogiCORE™ IP product when used as described in the product documentation. Xilinx cannot guarantee timing, functionality, or support if you do any of the following:

- Implement the solution in devices that are not defined in the documentation.
- Customize the solution beyond that allowed in the product documentation.
- Change any section of the design labeled DO NOT MODIFY.

To ask questions, navigate to the Xilinx Community Forums.

Hardware Debug

Hardware issues can range from link bring-up to problems seen after hours of testing. This section provides debug steps for common issues. The Vivado® debug feature is a valuable resource to use in hardware debug. The signal names mentioned in the following individual sections can be probed using the debug feature for debugging the specific problems.

General Checks

- Ensure that all the timing constraints for the core were properly applied and that all constraints were met during implementation.
- Does it work in post-place and route timing simulation? If problems are seen in hardware but not in timing simulation, this could indicate a PCB issue. Ensure that all clock sources are active and clean.
- If using MMCMs in the design, ensure that all MMCMs have obtained lock by monitoring the `locked` port.

API Reference

Overview

The RFSoC DFE DUC-DDC Mixer API defines the following data types and functions to allow the core to be configured and controlled by a software application.

Table 30: **Quick Function Reference**

Type	Name	Arguments
XDfeMix *	XDfeMix_InstanceInit	const char * DeviceNodeName
void	XDfeMix_InstanceClose	XDfeMix * InstancePtr
void	XDfeMix_Reset	XDfeMix * InstancePtr
void	XDfeMix_Configure	XDfeMix * InstancePtr XDfeMix_Cfg * Cfg
void	XDfeMix_Initialize	XDfeMix * InstancePtr XDfeMix_Init * Init
void	XDfeMix_Activate	XDfeMix * InstancePtr bool EnableLowPower
void	XDfeMix_Deactivate	XDfeMix * InstancePtr
XDfeMix_StateId	XDfeMix_GetStateID	XDfeMix * InstancePtr
void	XDfeMix_GetCurrentCCCfg	const XDfeMix * InstancePtr XDfeMix_CCCfg * CurrCCCfg
void	XDfeMix_GetEmptyCCCfg	const XDfeMix * InstancePtr XDfeMix_CCCfg * CCCfg

Table 30: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
void	XDfeMix_GetCarrierCfgAndNCO	const XDfeMix * InstancePtr XDfeMix_CCCfg * CCCfg s32 CCID u32 * CCSeqBitmap XDfeMix_CarrierCfg * CarrierCfg XDfeMix_NCO * NCO
void	XDfeMix_SetAntennaCfgInCCCfg	const XDfeMix * InstancePtr XDfeMix_CCCfg * CCCfg XDfeMix_AntennaCfg * AntennaCfg
u32	XDfeMix_AddCCtoCCCfg	XDfeMix * InstancePtr XDfeMix_CCCfg * CCCfg s32 CCID u32 CCSeqBitmap const XDfeMix_CarrierCfg * CarrierCfg const XDfeMix_NCO * NCO
void	XDfeMix_RemoveCCfromCCCfg	XDfeMix * InstancePtr XDfeMix_CCCfg * CCCfg s32 CCID
u32	XDfeMix_UpdateCCinCCCfg	const XDfeMix * InstancePtr XDfeMix_CCCfg * CCCfg s32 CCID const XDfeMix_CarrierCfg * CarrierCfg
u32	XDfeMix_SetNextCCCfgAndTrigger	const XDfeMix * InstancePtr const XDfeMix_CCCfg * CCCfg
u32	XDfeMix_AddCC	XDfeMix * InstancePtr s32 CCID u32 CCSeqBitmap const XDfeMix_CarrierCfg * CarrierCfg const XDfeMix_NCO * NCO
u32	XDfeMix_RemoveCC	XDfeMix * InstancePtr s32 CCID

Table 30: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
u32	XDfeMix_MoveCC	XDfeMix * InstancePtr s32 CCID u32 Rate u32 FromNCO u32 ToNCO
u32	XDfeMix_UpdateCC	const XDfeMix * InstancePtr s32 CCID const XDfeMix_CarrierCfg * CarrierCfg
u32	XDfeMix_SetAntennaGain	XDfeMix * InstancePtr u32 AntennaId u32 AntennaGain
u32	XDfeMix_UpdateAntennaCfg	XDfeMix * InstancePtr XDfeMix_AntennaCfg * AntennaCfg
void	XDfeMix_GetTriggersCfg	const XDfeMix * InstancePtr XDfeMix_TriggerCfg * TriggerCfg
void	XDfeMix_SetTriggersCfg	const XDfeMix * InstancePtr XDfeMix_TriggerCfg * TriggerCfg
void	XDfeMix_GetDUCDDCStatus	const XDfeMix * InstancePtr s32 CCID XDfeMix_DUCDDCStatus * DUCDDCStatus
void	XDfeMix_GetMixerStatus	const XDfeMix * InstancePtr s32 CCID XDfeMix_MixerStatus * MixerStatus
void	XDfeMix_SetTUserDelay	const XDfeMix * InstancePtr u32 Delay
u32	XDfeMix_GetTUserDelay	const XDfeMix * InstancePtr
u32	XDfeMix_GetTDataDelay	const XDfeMix * InstancePtr u32 Tap

Table 30: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
void	XDfeMix_GetVersions	XDfeMix_Version * SwVersion XDfeMix_Version * HwVersion
void	XDfeMix_GetInterruptMask	const XDfeMix * InstancePtr XDfeMix_InterruptMask * Mask
void	XDfeMix_SetInterruptMask	const XDfeMix * InstancePtr const XDfeMix_InterruptMask * Mask
void	XDfeMix_GetEventStatus	const XDfeMix * InstancePtr XDfeMix_Status * Status
void	XDfeMix_ClearEventStatus	const XDfeMix * InstancePtr const XDfeMix_Status * Status

Functions

XDfeMix_InstanceInit

API initialises one instance of a Mixer driver.

Traverses "/sys/bus/platform/device" directory (in Linux), to find registered `XDfeMix` device with the name DeviceNodeName. The first available slot in the instances array XDfeMix_Mixer[] will be taken as a DeviceNodeName object. On success it moves the state machine to a Ready state, while on failure stays in a Not Ready state.

Prototype

```
                XDfeMix
             * XDfeMix_InstanceInit(const char *DeviceNodeName);
```

Parameters

The following table lists the `XDfeMix_InstanceInit` function arguments.

Table 31: **XDfeMix_InstanceInit Arguments**

Type	Name	Description
const char *	DeviceNodeName	Device node name.

Returns

- Pointer to the instance if successful.
- NULL on error.

XDfeMix_InstanceClose

API closes the instance of a Mixer driver and moves the state machine to a Not Ready state.

Prototype

```
void XDfeMix_InstanceClose(XDfeMix *InstancePtr);
```

Parameters

The following table lists the `XDfeMix_InstanceClose` function arguments.

Table 32: **XDfeMix_InstanceClose Arguments**

Type	Name	Description
XDfeMix *	InstancePtr	Pointer to the Mixer instance.

XDfeMix_Reset

Resets Mixer and puts block into a reset state.

Prototype

```
void XDfeMix_Reset(XDfeMix *InstancePtr);
```

Parameters

The following table lists the `XDfeMix_Reset` function arguments.

Table 33: **XDfeMix_Reset Arguments**

Type	Name	Description
XDfeMix *	InstancePtr	Pointer to the Mixer instance.

XDfeMix_Configure

Reads configuration from device tree/xparameters.h and IP registers.

Removes S/W reset and moves the state machine to a Configured state.

Prototype

```
void XDfeMix_Configure(XDfeMix *InstancePtr, XDfeMix_Cfg *Cfg);
```

Parameters

The following table lists the `XDfeMix_Configure` function arguments.

Table 34: **XDfeMix_Configure Arguments**

Type	Name	Description
XDfeMix *	InstancePtr	Pointer to the Mixer instance.
XDfeMix_Cfg *	Cfg	Configuration data container.

XDfeMix_Initialize

DFE Mixer driver one time initialisation and moves the state machine to a Initialised state.

Prototype

```
void XDfeMix_Initialize(XDfeMix *InstancePtr, XDfeMix_Init *Init);
```

Parameters

The following table lists the `XDfeMix_Initialize` function arguments.

Table 35: **XDfeMix_Initialize Arguments**

Type	Name	Description
XDfeMix *	InstancePtr	Pointer to the Mixer instance.
XDfeMix_Init *	Init	Initialisation data container.

XDfeMix_Activate

Activates Mixer and moves the state machine to an Activated state.

Prototype

```
void XDfeMix_Activate(XDfeMix *InstancePtr, bool EnableLowPower);
```

Parameters

The following table lists the `XDfeMix_Activate` function arguments.

Table 36: **XDfeMix_Activate Arguments**

Type	Name	Description
XDfeMix *	InstancePtr	Pointer to the Mixer instance.
bool	EnableLowPower	Flag indicating low power.

XDfeMix_Deactivate

Deactivates Mixer and moves the state machine to Initialised state.

Prototype

```
void XDfeMix_Deactivate(XDfeMix *InstancePtr);
```

Parameters

The following table lists the `XDfeMix_Deactivate` function arguments.

Table 37: **XDfeMix_Deactivate Arguments**

Type	Name	Description
XDfeMix *	InstancePtr	Pointer to the Mixer instance.

XDfeMix_GetStateID

Gets a state machine state id.

Prototype

```
XDfeMix_StateId
XDfeMix_GetStateID(XDfeMix *InstancePtr);
```

Parameters

The following table lists the `XDfeMix_GetStateID` function arguments.

Table 38: **XDfeMix_GetStateID Arguments**

Type	Name	Description
XDfeMix *	InstancePtr	Pointer to the Mixer instance.

Returns

State machine StateID

XDfeMix_GetCurrentCCCfg

Returns the current CC configuration.

Not used slot ID in a sequence (Sequence.CCID[Index]) are represented as (-1), not the value in registers.

Prototype

```
void XDfeMix_GetCurrentCCCfg(const XDfeMix *InstancePtr, XDfeMix_CCCfg
*CurrCCCfg);
```

Parameters

The following table lists the `XDfeMix_GetCurrentCCCfg` function arguments.

Table 39: **XDfeMix_GetCurrentCCCfg Arguments**

Type	Name	Description
const XDfeMix *	InstancePtr	Pointer to the Mixer instance.
XDfeMix_CCCfg *	CurrCCCfg	CC configuration container.

XDfeMix_GetEmptyCCCfg

Returns configuration structure CCCfg with CCCfg->Sequence.Lengt value set in `XDfeMix_Configure()`, array CCCfg->Sequence.CCID[] members are set to not used value (-1) and the other CCCfg members are set to 0.

Prototype

```
void XDfeMix_GetEmptyCCCfg(const XDfeMix *InstancePtr, XDfeMix_CCCfg
*CCCfg);
```

Parameters

The following table lists the `XDfeMix_GetEmptyCCCfg` function arguments.

Table 40: **XDfeMix_GetEmptyCCCfg Arguments**

Type	Name	Description
const XDfeMix *	InstancePtr	Pointer to the Mixer instance.
XDfeMix_CCCfg *	CCCfg	CC configuration container.

XDfeMix_GetCarrierCfgAndNCO

Returns the current CC sequence bitmap, CCID carrier configuration and NCO configuration.

Prototype

```
void XDfeMix_GetCarrierCfgAndNCO(const XDfeMix *InstancePtr, XDfeMix_CCCfg
*CCCfg, s32 CCID, u32 *CCSeqBitmap, XDfeMix_CarrierCfg *CarrierCfg,
XDfeMix_NCO *NCO);
```

Parameters

The following table lists the `XDfeMix_GetCarrierCfgAndNCO` function arguments.

Table 41: **XDfeMix_GetCarrierCfgAndNCO Arguments**

Type	Name	Description
const XDfeMix *	InstancePtr	Pointer to the Mixer instance.
XDfeMix_CCCfg *	CCCfg	Component carrier (CC) configuration container.
s32	CCID	Channel ID.
u32 *	CCSeqBitmap	CC slot position container.
XDfeMix_CarrierCfg *	CarrierCfg	CC configuration container.
XDfeMix_NCO *	NCO	NCO configuration container.

XDfeMix_SetAntennaCfgInCCCfg

Set antenna configuration in CC configuration container.

Prototype

```
void XDfeMix_SetAntennaCfgInCCCfg(const XDfeMix *InstancePtr, XDfeMix_CCCfg
*CCCfg, XDfeMix_AntennaCfg *AntennaCfg);
```

Parameters

The following table lists the `XDfeMix_SetAntennaCfgInCCCfg` function arguments.

Table 42: **XDfeMix_SetAntennaCfgInCCCfg Arguments**

Type	Name	Description
const XDfeMix *	InstancePtr	Pointer to the Mixer instance.
XDfeMix_CCCfg *	CCCfg	CC configuration container.
XDfeMix_AntennaCfg *	AntennaCfg	Array of all antenna configurations.

XDfeMix_AddCCtoCCCfg

Adds specified CCID, with specified configuration, to a local CC configuration structure.

If there is insufficient capacity for the new CC the function will return an error. Initiates CC update (enable CCUpdate trigger TUSER Single Shot).

The returned CCCfg.Sequence is transleted as there is no explicit indication that SEQUENCE[i] is not used - 0 can define the slot as either used or not used. Sequence data that is returned in the CCIDSequence is not the same as what is written in the registers. The translation is:

- CCIDSequence.CCID[i] = -1 - if [i] is unused slot
- CCIDSequence.CCID[i] = CCID - if [i] is used slot
- a returned CCIDSequence->Length = length in register + 1

Prototype

```
u32 XDfeMix_AddCCtoCCCfg(XDfeMix *InstancePtr, XDfeMix_CCCfg *CCCfg, s32
CCID, u32 CCSeqBitmap, const XDfeMix_CarrierCfg *CarrierCfg, const
XDfeMix_NCO *NCO);
```

Parameters

The following table lists the `XDfeMix_AddCCtoCCCfg` function arguments.

Table 43: **XDfeMix_AddCCtoCCCfg Arguments**

Type	Name	Description
XDfeMix *	InstancePtr	Pointer to the Mixer instance.
XDfeMix_CCCfg *	CCCfg	Component carrier (CC) configuration container.
s32	CCID	Channel ID.
u32	CCSeqBitmap	CC slot position container.
const XDfeMix_CarrierCfg *	CarrierCfg	CC configuration container.
const XDfeMix_NCO *	NCO	NCO configuration container.

Returns

- XST_SUCCESS if successful.
- XST_FAILURE if error occurs.

XDfeMix_RemoveCCfromCCCfg

Removes specified CCID from a local CC configuration structure.

Note: For a sequence conversion see `XDfeMix_AddCCtoCCCfg()` comment.

Prototype

```
void XDfeMix_RemoveCCfromCCCfg(XDfeMix *InstancePtr, XDfeMix_CCCfg *CCCfg,
s32 CCID);
```

Parameters

The following table lists the `XDfeMix_RemoveCCfromCCCfg` function arguments.

Table 44: **XDfeMix_RemoveCCfromCCCfg Arguments**

Type	Name	Description
XDfeMix *	InstancePtr	Pointer to the Mixer instance.
XDfeMix_CCCfg *	CCCfg	Component carrier (CC) configuration container.
s32	CCID	Channel ID.

Returns

XDfeMix_UpdateCCinCCCfg

Updates specified CCID, with specified configuration to a local CC configuration structure.

If there is insufficient capacity for the new CC the function will return an error.

Prototype

```
u32 XDfeMix_UpdateCCinCCCfg(const XDfeMix *InstancePtr, XDfeMix_CCCfg
*CCCfg, s32 CCID, const XDfeMix_CarrierCfg *CarrierCfg);
```

Parameters

The following table lists the `XDfeMix_UpdateCCinCCCfg` function arguments.

Table 45: **XDfeMix_UpdateCCinCCCfg Arguments**

Type	Name	Description
const XDfeMix *	InstancePtr	Pointer to the Mixer instance.
XDfeMix_CCCfg *	CCCfg	Component carrier (CC) configuration container.
s32	CCID	Channel ID.
const XDfeMix_CarrierCfg *	CarrierCfg	CC configuration container.

Returns

- XST_SUCCESS if successful.
- XST_FAILURE if error occurs.

XDfeMix_SetNextCCCfgAndTrigger

Writes local CC configuration to the shadow (NEXT) registers and triggers copying from shadow to operational registers.

Prototype

```
u32 XDfeMix_SetNextCCCfgAndTrigger(const XDfeMix *InstancePtr, const
XDfeMix_CCCfg *CCCfg);
```

Parameters

The following table lists the `XDfeMix_SetNextCCCfgAndTrigger` function arguments.

Table 46: **XDfeMix_SetNextCCCfgAndTrigger Arguments**

Type	Name	Description
const XDfeMix *	InstancePtr	Pointer to the Mixer instance.
const XDfeMix_CCCfg *	CCCfg	CC configuration container.

Returns

- XST_SUCCESS if successful.
- XST_FAILURE if error occurs.

XDfeMix_AddCC

Adds specified CCID, with specified configuration.

If there is insufficient capacity for the new CC the function will return an error. Initiates CC update (enable CCUpdate trigger TUSER Single Shot).

Note: Clear event status with `XDfeMix_ClearEventStatus()` before running this API.

Prototype

```
u32 XDfeMix_AddCC(XDfeMix *InstancePtr, s32 CCID, u32 CCSeqBitmap, const
XDfeMix_CarrierCfg *CarrierCfg, const XDfeMix_NCO *NCO);
```

Parameters

The following table lists the `XDfeMix_AddCC` function arguments.

Table 47: **XDfeMix_AddCC Arguments**

Type	Name	Description
XDfeMix *	InstancePtr	Pointer to the Mixer instance.
s32	CCID	Channel ID.

Table 47: **XDfeMix_AddCC Arguments** *(cont'd)*

Type	Name	Description
u32	CCSeqBitmap	- up to 16 defined slots into which a CC can be allocated. The number of slots can be from 1 to 16 depending on system initialization. The number of slots is defined by the "sequence length" parameter which is provided during initialization. The Bit offset within the CCSeqBitmap indicates the equivalent Slot number to allocate. e.g. 0x0003 means the caller wants the passed component carrier (CC) to be allocated to slots 0 and 1.
const XDfeMix_CarrierCfg *	CarrierCfg	CC configuration container.
const XDfeMix_NCO *	NCO	NCO configuration container.

Returns

- XST_SUCCESS if successful.
- XST_FAILURE if error occurs.

XDfeMix_RemoveCC

Removes specified CCID.

Initiates CC update (enable CCUpdate trigger TUSER Single Shot).

Note: Clear event status with `XDfeMix_ClearEventStatus()` before running this API.

Prototype

```
u32 XDfeMix_RemoveCC(XDfeMix *InstancePtr, s32 CCID);
```

Parameters

The following table lists the `XDfeMix_RemoveCC` function arguments.

Table 48: **XDfeMix_RemoveCC Arguments**

Type	Name	Description
XDfeMix *	InstancePtr	Pointer to the Mixer instance.
s32	CCID	Channel ID.

Returns

- XST_SUCCESS if successful.
- XST_FAILURE if error occurs.

XDfeMix_MoveCC

Moves specified CCID from one NCO to another aligning phase to make it transparent.

Initiates CC update (enable CCUpdate trigger TUSER Single Shot).

Note: Clear event status with `XDfeMix_ClearEventStatus()` before running this API.

Prototype

```
u32 XDfeMix_MoveCC(XDfeMix *InstancePtr, s32 CCID, u32 Rate, u32 FromNCO,
u32 ToNCO);
```

Parameters

The following table lists the `XDfeMix_MoveCC` function arguments.

Table 49: **XDfeMix_MoveCC Arguments**

Type	Name	Description
XDfeMix *	InstancePtr	Pointer to the Mixer instance.
s32	CCID	Channel ID.
u32	Rate	NCO rate value [1,2,4].
u32	FromNCO	NCO value moving from.
u32	ToNCO	NCO value moving to.

Returns

- XST_SUCCESS if successful.
- XST_FAILURE if error occurs.

XDfeMix_UpdateCC

Updates specified CCID, with a configuration defined in CarrierCfg structure.

If there is insufficient capacity for the new CC the function will return an error.

Note: Clear event status with `XDfeMix_ClearEventStatus()` before running this API.

Prototype

```
u32 XDfeMix_UpdateCC(const XDfeMix *InstancePtr, s32 CCID, const
XDfeMix_CarrierCfg *CarrierCfg);
```

Parameters

The following table lists the `XDfeMix_UpdateCC` function arguments.

Table 50: **XDfeMix_UpdateCC Arguments**

Type	Name	Description
const XDfeMix *	InstancePtr	Pointer to the Mixer instance.
s32	CCID	Channel ID.
const XDfeMix_CarrierCfg *	CarrierCfg	CC configuration container.

Returns

- XST_SUCCESS if successful.
- XST_FAILURE if error occurs.

XDfeMix_SetAntennaGain

Sets antenna gain.

Initiate CC update (enable CCUpdate trigger TUSER Single Shot).

Note: Clear event status with `XDfeMix_ClearEventStatus()` before running this API.

Prototype

```
u32 XDfeMix_SetAntennaGain(XDfeMix *InstancePtr, u32 AntennaId, u32
AntennaGain);
```

Parameters

The following table lists the `XDfeMix_SetAntennaGain` function arguments.

Table 51: **XDfeMix_SetAntennaGain Arguments**

Type	Name	Description
XDfeMix *	InstancePtr	Pointer to the Mixer instance.
u32	AntennaId	Antenna ID.
u32	AntennaGain	Antenna gain, 0 for -6dB and 1 for 0dB.

Returns

- XST_SUCCESS if successful.
- XST_FAILURE if error occurs.

XDfeMix_UpdateAntennaCfg

Updates antenna cofiguration to all antennas.

Note: Clear event status with `XDfeMix_ClearEventStatus()` before running this API.

Prototype

```
u32 XDfeMix_UpdateAntennaCfg(XDfeMix *InstancePtr, XDfeMix_AntennaCfg
*AntennaCfg);
```

Parameters

The following table lists the `XDfeMix_UpdateAntennaCfg` function arguments.

Table 52: **XDfeMix_UpdateAntennaCfg Arguments**

Type	Name	Description
XDfeMix *	InstancePtr	Pointer to the Mixer instance.
XDfeMix_AntennaCfg *	AntennaCfg	Array of all antenna configurations.

Returns

- XST_SUCCESS if successful.
- XST_FAILURE if error occurs.

XDfeMix_GetTriggersCfg

Returns current trigger configuration.

Prototype

```
void XDfeMix_GetTriggersCfg(const XDfeMix *InstancePtr, XDfeMix_TriggerCfg
*TriggerCfg);
```

Parameters

The following table lists the `XDfeMix_GetTriggersCfg` function arguments.

Table 53: **XDfeMix_GetTriggersCfg Arguments**

Type	Name	Description
const XDfeMix *	InstancePtr	Pointer to the Mixer instance.
XDfeMix_TriggerCfg *	TriggerCfg	Trigger configuration container.

XDfeMix_SetTriggersCfg

Sets trigger configuration.

Prototype

```
void XDfeMix_SetTriggersCfg(const XDfeMix *InstancePtr, XDfeMix_TriggerCfg
*TriggerCfg);
```

Parameters

The following table lists the `XDfeMix_SetTriggersCfg` function arguments.

Table 54: **XDfeMix_SetTriggersCfg Arguments**

Type	Name	Description
const XDfeMix *	InstancePtr	Pointer to the Mixer instance.
XDfeMix_TriggerCfg *	TriggerCfg	Trigger configuration container.

XDfeMix_GetDUCDDCStatus

Gets DUC/DDC status for a specified CCID.

Prototype

```
void XDfeMix_GetDUCDDCStatus(const XDfeMix *InstancePtr, s32 CCID,
XDfeMix_DUCDDCStatus *DUCDDCStatus);
```

Parameters

The following table lists the `XDfeMix_GetDUCDDCStatus` function arguments.

Table 55: **XDfeMix_GetDUCDDCStatus Arguments**

Type	Name	Description
const XDfeMix *	InstancePtr	Pointer to the Mixer instance.
s32	CCID	Channel ID.
XDfeMix_DUCDDCStatus *	DUCDDCStatus	DUC/DDC status container.

XDfeMix_GetMixerStatus

Gets Mixer status for a specified CCID.

Prototype

```
void XDfeMix_GetMixerStatus(const XDfeMix *InstancePtr, s32 CCID,
XDfeMix_MixerStatus *MixerStatus);
```

Parameters

The following table lists the `XDfeMix_GetMixerStatus` function arguments.

Table 56: **XDfeMix_GetMixerStatus Arguments**

Type	Name	Description
const XDfeMix *	InstancePtr	Pointer to the Mixer instance.
s32	CCID	Channel ID.
XDfeMix_MixerStatus *	MixerStatus	Mixer status container.

XDfeMix_SetTUserDelay

Sets the delay, which will be added to TUSER and TLAST (delay matched through the IP).

Prototype

```
void XDfeMix_SetTUserDelay(const XDfeMix *InstancePtr, u32 Delay);
```

Parameters

The following table lists the `XDfeMix_SetTUserDelay` function arguments.

Table 57: **XDfeMix_SetTUserDelay Arguments**

Type	Name	Description
const XDfeMix *	InstancePtr	Pointer to the Mixer instance.
u32	Delay	Requested delay variable.

XDfeMix_GetTUserDelay

Reads the delay, which will be added to TUSER and TLAST (delay matched through the IP).

Prototype

```
u32 XDfeMix_GetTUserDelay(const XDfeMix *InstancePtr);
```

Parameters

The following table lists the `XDfeMix_GetTUserDelay` function arguments.

Table 58: **XDfeMix_GetTUserDelay Arguments**

Type	Name	Description
const XDfeMix *	InstancePtr	Pointer to the Mixer instance.

Returns

Delay value

XDfeMix_GetTDataDelay

Returns data latency + tap.

Prototype

```
u32 XDfeMix_GetTDataDelay(const XDfeMix *InstancePtr, u32 Tap);
```

Parameters

The following table lists the `XDfeMix_GetTDataDelay` function arguments.

Table 59: **XDfeMix_GetTDataDelay Arguments**

Type	Name	Description
const XDfeMix *	InstancePtr	Pointer to the Mixer instance.
u32	Tap	Tap value.

Returns

Data latency value.

XDfeMix_GetVersions

This API gets the driver and HW design version.

Prototype

```
void XDfeMix_GetVersions(const XDfeMix *InstancePtr, XDfeMix_Version
*SwVersion, XDfeMix_Version *HwVersion);
```

Parameters

The following table lists the `XDfeMix_GetVersions` function arguments.

Table 60: **XDfeMix_GetVersions Arguments**

Type	Name	Description
XDfeMix_Version *	SwVersion	Driver version number.
XDfeMix_Version *	HwVersion	HW version number.

XDfeMix_GetInterruptMask

Gets interrupt mask status.

Prototype

```
void XDfeMix_GetInterruptMask(const XDfeMix *InstancePtr,
XDfeMix_InterruptMask *Mask);
```

Parameters

The following table lists the `XDfeMix_GetInterruptMask` function arguments.

Table 61: **XDfeMix_GetInterruptMask Arguments**

Type	Name	Description
const XDfeMix *	InstancePtr	Pointer to the Mixer instance.
XDfeMix_InterruptMask *	Mask	Interrupt masks container.

XDfeMix_SetInterruptMask

Sets interrupt mask.

Prototype

```
void XDfeMix_SetInterruptMask(const XDfeMix *InstancePtr, const
XDfeMix_InterruptMask *Mask);
```

Parameters

The following table lists the `XDfeMix_SetInterruptMask` function arguments.

Table 62: **XDfeMix_SetInterruptMask Arguments**

Type	Name	Description
const XDfeMix *	InstancePtr	Pointer to the Mixer instance.
const XDfeMix_InterruptMask *	Mask	Interrupt mask flags container. • 0 - does not mask coresponding interrupt • 1 - masks coresponding interrupt

XDfeMix_GetEventStatus

Gets event status.

Prototype

```
void XDfeMix_GetEventStatus(const XDfeMix *InstancePtr, XDfeMix_Status
*Status);
```

Parameters

The following table lists the `XDfeMix_GetEventStatus` function arguments.

Table 63: **XDfeMix_GetEventStatus Arguments**

Type	Name	Description
const XDfeMix *	InstancePtr	Pointer to the Mixer instance.
XDfeMix_Status *	Status	Event status container.

XDfeMix_ClearEventStatus

Clears event status.

Prototype

```
void XDfeMix_ClearEventStatus(const XDfeMix *InstancePtr, const
XDfeMix_Status *Status);
```

Parameters

The following table lists the `XDfeMix_ClearEventStatus` function arguments.

Table 64: **XDfeMix_ClearEventStatus Arguments**

Type	Name	Description
const XDfeMix *	InstancePtr	Pointer to the Mixer instance.
const XDfeMix_Status *	Status	Clear event status container. • 0 - does not clear coresponding event status • 1 - clear coresponding event status

Enumerations

Enumeration XDfeMix_StateId

Table 65: **Enumeration XDfeMix_StateId Values**

Value	Description
XDFEMIX_STATE_NOT_READY	Not ready state.
XDFEMIX_STATE_READY	Ready state.
XDFEMIX_STATE_RESET	Reset state.
XDFEMIX_STATE_CONFIGURED	Configured state.
XDFEMIX_STATE_INITIALISED	Initialised state.

Table 65: **Enumeration XDfeMix_StateId Values** *(cont'd)*

Value	Description
XDFEMIX_STATE_OPERATIONAL	Operational state.

Definitions

XDFEMIX_DRIVER_VERSION_MINOR

Driver's minor version number.

Definition

```
#define XDFEMIX_DRIVER_VERSION_MINOR(2U)
```

XDFEMIX_DRIVER_VERSION_MAJOR

Driver's major version number.

Definition

```
#define XDFEMIX_DRIVER_VERSION_MAJOR(1U)
```

XDFEMIX_MAX_NUM_INSTANCES

Maximum number of driver instances running at the same time.

Definition

```
#define XDFEMIX_MAX_NUM_INSTANCES    (10U)
```

XST_SUCCESS

Success flag.

Definition

```
#define XST_SUCCESS(0U)
```

XST_FAILURE

Failure flag.

Definition

```
#define XST_FAILURE(1U)
```

XDFEMIX_NODE_NAME_MAX_LENGTH

Node name maximum length.

Definition

```
#define XDFEMIX_NODE_NAME_MAX_LENGTH(50U)
```

XDFEMIX_CC_NUM

Maximum CC number.

Definition

```
#define XDFEMIX_CC_NUM(16)
```

XDFEMIX_ANT_NUM_MAX

Maximum anntena number.

Definition

```
#define XDFEMIX_ANT_NUM_MAX(8U)
```

XDFEMIX_SEQ_LENGTH_MAX

Maximum sequence length.

Definition

```
#define XDFEMIX_SEQ_LENGTH_MAX(16U)
```

XDFEMIX_CC_GAIN_MAX

Maximum CC gain.

Definition

```
#define XDFEMIX_CC_GAIN_MAX(3U)
```

XDFEMIX_COMPATIBLE_STRING

Device name property.

Definition

```
#define XDFEMIX_COMPATIBLE_STRING    "xlnx,xdfe-cc-mixer-1.0"
```

XDFEMIX_PLATFORM_DEVICE_DIR

Device location in a file system.

Definition

```
#define XDFEMIX_PLATFORM_DEVICE_DIR    "/sys/bus/platform/devices/"
```

XDFEMIX_COMPATIBLE_PROPERTY

Device tree property.

Definition

```
#define XDFEMIX_COMPATIBLE_PROPERTY"compatible"
```

XDFEMIX_BUS_NAME

System bus name.

Definition

```
#define XDFEMIX_BUS_NAME"platform"
```

XDFEMIX_BASEADDR_PROPERTY

Base address property.

Definition

```
#define XDFEMIX_BASEADDR_PROPERTY"reg"
```

XDFEMIX_BASEADDR_SIZE

Base address bit-size.

Definition

```
#define XDFEMIX_BASEADDR_SIZE8U
```

XDFEMIX_MODE_CFG

Mode: 0 = DOWNLINK, 1 = UPLINK.

Definition

```
#define XDFEMIX_MODE_CFG"xlnx,mode"
```

XDFEMIX_NUM_ANTENNA_CFG

Number of antenna property.

Definition

```
#define XDFEMIX_NUM_ANTENNA_CFG    "xlnx,num-antenna"
```

XDFEMIX_MAX_USABLE_CCIDS_CFG

Maximum number of CC's per antenna.

Definition

```
#define XDFEMIX_MAX_USABLE_CCIDS_CFG    "xlnx,max-useable-ccids"
```

XDFEMIX_LANES_CFG

Number of parallel data channels required.

Definition

```
#define XDFEMIX_LANES_CFG    "xlnx,lanes"
```

XDFEMIX_ANTENNA_INTERLEAVE_CFG

Number of TDM antenna.

Definition

```
#define XDFEMIX_ANTENNA_INTERLEAVE_CFG    "xlnx,antenna-interleave"
```

XDFEMIX_MIXER_CPS_CFG

Mixer clock per sample property.

Definition

```
#define XDFEMIX_MIXER_CPS_CFG    "xlnx,mixer-cps"
```

XDFEMIX_DATA_IWIDTH_CFG

Input stream data bit width.

Definition

```
#define XDFEMIX_DATA_IWIDTH_CFG    "xlnx,data-iwidth"
```

XDFEMIX_DATA_OWIDTH_CFG

Output stream data bit width.

Definition

```
#define XDFEMIX_DATA_OWIDTH_CFG    "xlnx,data-owidth"
```

XDFEMIX_TUSER_WIDTH_CFG

Width of the tuser input.

Definition

```
#define XDFEMIX_TUSER_WIDTH_CFG    "xlnx,tuser-width"
```

Data Structure Index

The following is a list of data structures:

- XDfeMix
- XDfeMix_AntennaCfg
- XDfeMix_CCCfg
- XDfeMix_CCSequence
- XDfeMix_CarrierCfg

- XDfeMix_Cfg
- XDfeMix_Config
- XDfeMix_DUCDDCCfg
- XDfeMix_DUCDDCStatus
- XDfeMix_Frequency
- XDfeMix_Init
- XDfeMix_InternalDUCDDCCfg
- XDfeMix_MixerStatus
- XDfeMix_ModelParameters
- XDfeMix_NCO
- XDfeMix_Phase
- XDfeMix_PhaseOffset
- XDfeMix_Status
- XDfeMix_Trigger
- XDfeMix_TriggerCfg
- XDfeMix_Version

XDfeMix

Mixer Structure.

Declaration

```
typedef struct
{
  XDfeMix_Config Config,
  XDfeMix_StateId StateId,
  s32 NotUsedCCID,
  u32 SequenceLength,
  char NodeName[XDFEMIX_NODE_NAME_MAX_LENGTH],
  struct metal_io_region * Io,
  struct metal_device * Device
} XDfeMix;
```

Table 66: **Structure XDfeMix Member Description**

Member	Description
Config	Config Structure.
StateId	StateId.
NotUsedCCID	Lowest CCID number not allocated.
SequenceLength	Exact sequence length.

Table 66: **Structure XDfeMix Member Description** *(cont'd)*

Member	Description
NodeName	Node name.
Io	Libmetal IO structure.
Device	Libmetal device structure.

XDfeMix_AntennaCfg

Configuration for a single Antenna.

Declaration

```
typedef struct
{
  u32 Gain[XDFEMIX_ANT_NUM_MAX]
} XDfeMix_AntennaCfg;
```

Table 67: **Structure XDfeMix_AntennaCfg Member Description**

Member	Description
Gain	[0: 0dB,1:-6dB] Antenna gain adjustment

XDfeMix_CarrierCfg

Configuration for a single CC (implementation note: notice that there are two parts, one part (DUCDDCCfg) mapping to the CCCfg state, and another that is written directly to NCO registers (xDFECCMixerNCOT).

Note that CC Filter does not have the second part. However, from an API perspective, this is hidden.

Declaration

```
typedef struct
{
  XDfeMix_DUCDDCCfg DUCDDCCfg
} XDfeMix_CarrierCfg;
```

Table 68: **Structure XDfeMix_CarrierCfg Member Description**

Member	Description
DUCDDCCfg	Defines settings for single CC's DUC/DDC.

XDfeMix_CCCfg

Full CC configuration.

Declaration

```
typedef struct
{
  XDfeMix_CCSequence Sequence,
  XDfeMix_InternalDUCDDCCfg DUCDDCCfg[16],
  XDfeMix_NCO NCO[XDFEMIX_NCO_MAX+1],
  XDfeMix_AntennaCfg AntennaCfg
} XDfeMix_CCCfg;
```

Table 69: **Structure XDfeMix_CCCfg Member Description**

Member	Description
Sequence	CCID sequence.
DUCDDCCfg	DUC/DDC configurations for all CCs.
NCO	Defines settings for single CC's NCO.
AntennaCfg	Antenna configuration.

XDfeMix_CCSequence

Defines a CCID sequence.

Declaration

```
typedef struct
{
  u32 Length,
  s32 CCID[XDFEMIX_SEQ_LENGTH_MAX]
} XDfeMix_CCSequence;
```

Table 70: **Structure XDfeMix_CCSequence Member Description**

Member	Description
Length	[1-16] Sequence length.
CCID	[0-15].Array of CCID's arranged in the order the CCIDs are required to be processed in the DUC/DDC Mixer.

XDfeMix_Cfg

Configuration.

Declaration

```
typedef struct
{
  XDfeMix_Version Version,
  XDfeMix_ModelParameters ModelParams
} XDfeMix_Cfg;
```

Table 71: **Structure XDfeMix_Cfg Member Description**

Member	Description
Version	Logicore version.
ModelParams	Logicore parameterization.

XDfeMix_Config

Mixer Config Structure.

Declaration

```
typedef struct
{
  u32 DeviceId,
  metal_phys_addr_t BaseAddr,
  u32 Mode,
  u32 NumAntenna,
  u32 MaxUseableCcids,
  u32 Lanes,
  u32 AntennaInterleave,
  u32 MixerCps,
  u32 DataIWidth,
  u32 DataOWidth,
  u32 TUserWidth
} XDfeMix_Config;
```

Table 72: **Structure XDfeMix_Config Member Description**

Member	Description
DeviceId	Device Id.
BaseAddr	Device base address.
Mode	[0,1] 0=downlink, 1=uplink
NumAntenna	[1,2,4,8]
MaxUseableCcids	[4,8]
Lanes	[1-4]
AntennaInterleave	[1,2,4,8]
MixerCps	[1,2,4]
DataIWidth	[16,24] 16 for 16-bit sample data and 24 for 18-bit sample data.
DataOWidth	[16,24] 16 for 16-bit sample data and 24 for 18-bit sample data.
TUserWidth	[0-64]

XDfeMix_DUCDDCCfg

Defines settings for single CC's DUC/DDC.

Declaration

```
typedef struct
{
  u32 NCOIdx,
  u32 CCGain
} XDfeMix_DUCDDCCfg;
```

Table 73: **Structure XDfeMix_DUCDDCCfg Member Description**

Member	Description
NCOIdx	[0-7] DUC/DDC NCO assignment
CCGain	[0-3] Adjust gain of CCID after mixing (applies to all antennas for that CCID). Only applicable to downlink. • 0 = MINUS18DB: Apply -18dB gain. • 1 = MINUS12DB: Apply -12dB gain. • 2 = MINUS6DB: Apply -6dB gain. • 3 = ZERODB: Apply -0dB gain.

XDfeMix_DUCDDCStatus

DUC/DDC status.

Declaration

```
typedef struct
{
  u32 RealOverflowStage,
  u32 ImagOverflowStage,
  u32 FirstAntennaOverflowing,
  u32 FirstCCIDOverflowing
} XDfeMix_DUCDDCStatus;
```

Table 74: **Structure XDfeMix_DUCDDCStatus Member Description**

Member	Description
RealOverflowStage	[0-3] First stage in which overflow in real data has occurred.
ImagOverflowStage	[0-3] First stage in which overflow in imaginary data has occurred.
FirstAntennaOverflowing	[0-7] Lowest antenna in which overflow has occurred.
FirstCCIDOverflowing	[0-7] Lowest CCID in which overflow has occurred.

XDfeMix_Frequency

Defines frequency for single CC's NCO.

Declaration

```
typedef struct
{
  u32 FrequencyControlWord,
  u32 SingleModCount,
  u32 DualModCount,
  XDfeMix_PhaseOffset PhaseOffset
} XDfeMix_Frequency;
```

Table 75: **Structure XDfeMix_Frequency Member Description**

Member	Description
FrequencyControlWord	[0-2^32-1] Phase increment
SingleModCount	[0-2^32-1] Single modulus cycle count (S)
DualModCount	[0-2^32-1] Dual modulus cycle count (T-S)
PhaseOffset	[0-2^17-1] Phase offset

XDfeMix_Init

Initialization, "one-time" configuration parameters.

Declaration

```
typedef struct
{
  XDfeMix_CCSequence Sequence
} XDfeMix_Init;
```

Table 76: **Structure XDfeMix_Init Member Description**

Member	Description
Sequence	CCID Sequence.

XDfeMix_InternalDUCDDCCfg

Defines settings for internal single CC's DUC/DDC.

Declaration

```
typedef struct
{
  u32 NCOIdx,
  u32 Rate,
  u32 CCGain
} XDfeMix_InternalDUCDDCCfg;
```

Table 77: **Structure XDfeMix_InternalDUCDDCCfg Member Description**

Member	Description
NCOIdx	[0-7] DUC/DDC NCO assignment
Rate	[0-5].Interpolation/decimation rate: • 0: This CCID is disabled • 1: 1x interpolation/decimation • 2: 2x interpolation/decimation • 3: 4x interpolation/decimation • 4: 8x interpolation/decimation • 5: 16x interpolation/decimation
CCGain	[0-3] Adjust gain of CCID after mixing (applies to all antennas for that CCID). Only applicable to downlink. • 0 = MINUS18DB: Apply -18dB gain. • 1 = MINUS12DB: Apply -12dB gain. • 2 = MINUS6DB: Apply -6dB gain. • 3 = ZERODB: Apply -0dB gain.

XDfeMix_MixerStatus

Mixer status.

Declaration

```
typedef struct
{
   u32 AdderStage,
   u32 AdderAntenna,
   u32 MixCCID,
   u32 MixAntenna
} XDfeMix_MixerStatus;
```

Table 78: **Structure XDfeMix_MixerStatus Member Description**

Member	Description
AdderStage	[0,1] Earliest stage in which overflow occurred. • 0 = COMPLEX_MULT: Complex multiplier output overflowed and has been saturated. • 1 = FIRST_ADDER: First antenna adder output overflowed and has been saturated. • 2 = SECOND_ADDER: Second antenna adder output overflowed and has been saturated.

Table 78: **Structure XDfeMix_MixerStatus Member Description** *(cont'd)*

Member	Description
AdderAntenna	[0-7] Lowest antenna in which overflow has occurred.
MixCCID	[0-7] Lowest CCID on which overflow has occurred in mixer.
MixAntenna	[0-7] Lowest antenna in which overflow has occurred.

XDfeMix_ModelParameters

Mixer model parameters structure.

Data defined in Device tree/xparameters.h.

Declaration

```
typedef struct
{
  u32 Mode,
  u32 NumAntenna,
  u32 MaxUseableCcids,
  u32 Lanes,
  u32 AntennaInterleave,
  u32 MixerCps,
  u32 DataIWidth,
  u32 DataOWidth,
  u32 TUserWidth
} XDfeMix_ModelParameters;
```

Table 79: **Structure XDfeMix_ModelParameters Member Description**

Member	Description
Mode	[0,1] 0=downlink, 1=uplink
NumAntenna	[1,2,4,8] Number of antennas
MaxUseableCcids	[4,8] Maximum CC usable
Lanes	[1-4] Number of lanes
AntennaInterleave	[1,2,4,8] Number of Antenna slots
MixerCps	[1,2,4]
DataIWidth	[16,24] 16 for 16-bit sample data and 24 for 18-bit sample data.
DataOWidth	[16,24] 16 for 16-bit sample data and 24 for 18-bit sample data.
TUserWidth	[0-64]

XDfeMix_NCO

Defines settings for single CC's NCO.

Declaration

```
typedef struct
{
  XDfeMix_Frequency FrequencyCfg,
  XDfeMix_Phase PhaseCfg,
  u32 NCOGain
} XDfeMix_NCO;
```

Table 80: **Structure XDfeMix_NCO Member Description**

Member	Description
FrequencyCfg	Frequency configuration.
PhaseCfg	Phase configuration.
NCOGain	[0,1,2,3] Scaling of NCO output (0=0dB, 1=-3dB, 2=-6dB, 3=-9dB)

XDfeMix_Phase

Defined phase for single CC's NCO.

Declaration

```
typedef struct
{
  u32 PhaseAcc,
  u32 DualModCount,
  u32 DualModSel
} XDfeMix_Phase;
```

Table 81: **Structure XDfeMix_Phase Member Description**

Member	Description
PhaseAcc	[0-2^32-1] Phase accumulator value
DualModCount	[0-2^32-1] Dual modulus count value (T-S)
DualModSel	[0,1] Dual modulus select value

XDfeMix_PhaseOffset

Phase Offset.

Declaration

```
typedef struct
{
  u32 PhaseOffset
} XDfeMix_PhaseOffset;
```

Table 82: **Structure XDfeMix_PhaseOffset Member Description**

Member	Description
PhaseOffset	[0-2^17-1] Phase offset

XDfeMix_Status

Interrupt status and mask.

Declaration

```
typedef struct
{
  u32 DUCDDCOverflow,
  u32 MixerOverflow,
  u32 CCUpdate,
  u32 CCSequenceError
} XDfeMix_Status;
```

Table 83: **Structure XDfeMix_Status Member Description**

Member	Description
DUCDDCOverflow	[0,1] Mask overflow in DUC/DDC
MixerOverflow	[0,1] Mask overflow in mixer
CCUpdate	[0,1] Mask update interrupt
CCSequenceError	[0,1] Mask sequence error

XDfeMix_Trigger

Trigger configuration.

Declaration

```
typedef struct
{
  u32 TriggerEnable,
  u32 Mode,
  u32 TuserEdgeLevel,
  u32 StateOutput,
  u32 TUSERBit
} XDfeMix_Trigger;
```

Table 84: **Structure XDfeMix_Trigger Member Description**

Member	Description
TriggerEnable	[0,1], Enable Trigger: • 0 = DISABLED: Trigger Pulse and State outputs are disabled. • 1 = ENABLED: Trigger Pulse and State outputs are enabled and follow the settings described below.
Mode	[0-3], Specify Trigger Mode. In TUSER_Single_Shot mode as soon as the TUSER_Edge_level condition is met the State output will be driven to the value specified in STATE_OUTPUT. The Pulse output will pulse high at the same time. No further change will occur until the trigger register is re-written. In TUSER Continuous mode each time a TUSER_Edge_level condition is met the State output will be driven to the value specified in STATE_OUTPUT This will happen continuously until the trigger register is re-written. The pulse output is disabled in Continuous mode: • 0 = IMMEDIATE: Applies the value of STATE_OUTPUT immediatetly the register is written. • 1 = TUSER_SINGLE_SHOT: Applies the value of STATE_OUTPUT once when the TUSER_EDGE_LEVEL condition is satisfied. • 2 = TUSER_CONTINUOUS: Applies the value of STATE_OUTPUT continually when TUSER_EDGE_LEVEL condition is satisfied. • 3 = RESERVED: Reserved - will default to 0 behaviour.
TuserEdgeLevel	[0-3], Specify either Edge or Level of the TUSER input as the source condition of the trigger. Difference between Level and Edge is Level will generate a trigger immediately the TUSER level is detected. Edge will ensure a TUSER transition has come first: • 0 = LOW: Trigger occurs immediately after a low-level is seen on TUSER provided tvalid is high. • 1 = HIGH: Trigger occurs immediately after a high-level is seen on TUSER provided tvalid is high. • 2 = FALLING: Trigger occurs immediately after a high to low transition on TUSER provided tvalid is high. • 3 = RISING: Trigger occurs immediately after a low to high transition on TUSER provided tvalid is high.
StateOutput	[0,1], Specify the State output value: • 0 = DISABLED: Place the State output into the Disabled state. • 1 = ENABLED: Place the State output into the Enabled state.
TUSERBit	[0-255], Specify which DIN TUSER bit to use as the source for the trigger when MODE = 1 or 2.

XDfeMix_TriggerCfg

All IP triggers.

Declaration

```
typedef struct
{
  XDfeMix_Trigger Activate,
  XDfeMix_Trigger LowPower,
  XDfeMix_Trigger CCUpdate
} XDfeMix_TriggerCfg;
```

Table 85: **Structure XDfeMix_TriggerCfg Member Description**

Member	Description
Activate	Switch between "Initialized", ultra-low power state, and "Operational". One-shot trigger, disabled following a single event.
LowPower	Switch between "Low-power" and "Operational" state.
CCUpdate	Transition to next CC configuration. Will initiate flush based on CC configuration.

XDfeMix_Version

Logicore version.

Declaration

```
typedef struct
{
  u32 Major,
  u32 Minor,
  u32 Revision,
  u32 Patch
} XDfeMix_Version;
```

Table 86: **Structure XDfeMix_Version Member Description**

Member	Description
Major	Major version number.
Minor	Minor version number.
Revision	Revision number.
Patch	Patch number.

Examples

Pass-Through Example

This function runs the DFE Equalizer device using the driver APIs and performs the following tasks:

- Create and system initialize the device driver instance.
- Read SW and HW version numbers.
- Reset the device.
- Configure the device.
- Initialize the device.
- Set the triggers
- Activate the device.
- Load an equalizer coefficients.
- DeActivate the device.

Note: For more information, browse your local `embeddedsw` directory and go to path `./XilinxProcessorIPLib/drivers/dfeequ/examples` relative to the embeddedsw root folder.

Appendix C

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado® IDE, select **Help → Documentation and Tutorials**.
- On Windows, select **Start → All Programs → Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

***Note*:** For more information on DocNav, see the Documentation Navigator page on the Xilinx website.

References

These documents provide supplemental material useful with this guide:

1. *Vivado Design Suite: AXI Reference Guide* (UG1037)
2. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
3. *Vivado Design Suite User Guide: Designing with IP* (UG896)
4. *Vivado Design Suite User Guide: Getting Started* (UG910)
5. *Vivado Design Suite User Guide: Logic Simulation* (UG900)

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
12/15/2021 Version Early Access Draft	
Xilinx Confidential Draft. Approved for external release under NDA only.	N/A

Please Read: Important Legal Notices

AUTOMOTIVE APPLICATIONS DISCLAIMER

Copyright