# Table of Contents

**Chapter 1: IP Facts**
- Features .............................................................................................................. 4
- IP Facts ................................................................................................................ 5

**Chapter 2: Overview**
- RF-ADC .............................................................................................................. 8
- RF-DAC ................................................................................................................ 11
- Licensing and Ordering ...................................................................................... 13

**Chapter 3: Product Specification**
- Performance ...................................................................................................... 16
- Resource Use ...................................................................................................... 16
- Port Descriptions ............................................................................................... 16
- Register Space ................................................................................................... 21

**Chapter 4: Designing with the Core**
- IP Core Configuration in Vivado Design Suite .................................................. 32
- Software Driver ................................................................................................. 32
- RF-ADC .............................................................................................................. 32
- RF-DAC .............................................................................................................. 65
- Quadrature Modulator Correction ..................................................................... 91
- Dynamic Update Events .................................................................................... 94
- PLL ..................................................................................................................... 95
- Interrupt Handling ............................................................................................. 97
- Clocking ............................................................................................................ 100
- Resets ............................................................................................................... 105
- Power-up Sequence ........................................................................................... 106
- Interfacing to the AXI4-Stream Interface ........................................................... 108
- Applications ..................................................................................................... 109

**Chapter 5: Design Flow Steps**
- Customizing and Generating the Core ............................................................... 122
IP Facts

The Xilinx® LogiCORE™ IP Zynq® UltraScale+™ RFSoC RF Data Converter IP core provides a configurable wrapper to allow the RF-DAC and RF-ADC blocks to be used in IP integrator designs.

Features

- Up to 16 14-bit 6.5 GSPS RF-DACs
- Eight 12-bit 4 GSPS RF-ADCs, or 16 12-bit 2 GSPS RF-ADCs, depending on device
- Supports alignment between multiple converters
- Pre-programs RF-DAC and RF-ADC with key user-defined parameters
- Multiple AXI4-Stream data interfaces for RF-ADCs and RF-DACs
- Single AXI4-Lite configuration interface
- 1x (bypass), 2x, 4x, 8x interpolation
- 1x (bypass), 2x, 4x, 8x decimation
- Digital complex mixers
- Numerical Controlled Oscillator (NCO)
- Quadrature Modulation Correction (QMC)

In this guide reference is made to the 2 GSPS RF-ADC, the 4 GSPS RF-ADC, and the 6.5 G RF-DAC; for the actual sampling rate specifications, see the Zynq UltraScale+ RFSoC Data Sheet: DC and AC Switching Characteristics (DS926)
IP Facts

LogiCORE IP Facts Table

<table>
<thead>
<tr>
<th>Core Specifics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supported Device Family</td>
</tr>
<tr>
<td>Supported User Interfaces</td>
</tr>
<tr>
<td>Resources</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Provided with Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design Files</td>
</tr>
<tr>
<td>Example Design</td>
</tr>
<tr>
<td>Test Bench</td>
</tr>
<tr>
<td>Constraints File</td>
</tr>
<tr>
<td>Simulation Model</td>
</tr>
<tr>
<td>Supported S/W Driver</td>
</tr>
</tbody>
</table>

Tested Design Flows

<table>
<thead>
<tr>
<th>Tested Design Flows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design Entry</td>
</tr>
<tr>
<td>Simulation</td>
</tr>
<tr>
<td>Synthesis</td>
</tr>
</tbody>
</table>

Support

Provided by Xilinx at the Xilinx Support web page

Notes:

1. For a complete list of supported devices, see the Vivado® IP catalog.
2. For the supported versions of the tools, see the Xilinx Design Tools: Release Notes Guide.
Overview

The Xilinx® Zynq® UltraScale+™ RFSoC family integrates the key subsystems required to implement a complete software-defined radio including direct RF sampling data converters, enabling CPRI and Gigabit Ethernet-to-RF on a single, highly programmable SoC.

Each RFSoC offers multiple RF-sampling analog-to-digital (RF-ADC) and RF-sampling digital-to-analog (RF-DAC) data converters. The data converters are high-precision, high-speed and power-efficient. Both are highly configurable and tightly integrated with the programmable logic (PL) resources of the Zynq UltraScale+ RFSoC.

The RF-ADC supports a sample rate of over 4 GSPS (see the Zynq UltraScale+ RFSoC Data Sheet: DC and AC Switching Characteristics (DS926) for the exact sample rate) and input signal frequencies up to 4 GHz, with excellent noise spectral density. The RF-DAC generates output carrier frequencies up to 4 GHz while operating in the second Nyquist zone with excellent noise spectral density at a sample rate of over 6.5 GSPS (see the Zynq UltraScale+ RFSoC Data Sheet: DC and AC Switching Characteristics (DS926) for the exact sample rate). The RF data converters also include power efficient digital down converters (DDCs) and digital up converters (DUCs) that include programmable interpolation and decimation rates, a numerically controlled oscillator (NCO), and a complex mixer. The DDCs and DUCs can also support dual-band operation. The following figure shows the block diagram of the Zynq® UltraScale+™ RFSoC RF Data Converter.

The RF-ADCs and RF-DACs are organized into tiles, each containing four RF-DACs or two or four RF-ADCs. Multiple tiles are available in each Zynq® UltraScale+™ RFSoC (see the specific device data sheet for the number of tiles and converters per device). Each tile also includes a block with a PLL and all the necessary clock handling logic and distribution routing for the analog and digital logic.

This guide describes the Zynq® UltraScale+™ RFSoC RF Data Converter IP core and software drivers that are used to configure the data converters and instantiate them for use in a design. In this guide, reference is made to the 2 GSPS RF-ADC RF-ADC, the 4 GSPS RF-ADC, and the 6.5 G RF-DAC; for the actual sampling rate specifications see the Zynq UltraScale+ RFSoC Data Sheet: DC and AC Switching Characteristics (DS926)

For device specifications and additional information, see:

- Zynq UltraScale+ RFSoC Data Sheet: Overview (DS889)
- Zynq UltraScale+ RFSoC Data Sheet: DC and AC Switching Characteristics (DS926)
Figure 1: Zynq® UltraScale+™ RFSoC RF Data Converter IP Core in Zynq UltraScale+ RFSoC

Figure 2: Converter Tile Structure
RF-ADC

There are two types of RF-ADC tile, which support either 2 GSPS or 4 GSPS RF-ADCs. The type of tile available is device dependent (see the Zynq UltraScale+ RFSoC Data Sheet: Overview (DS889)). Each tile includes a PLL and clocking instance. All RF-ADCs within a tile share this common clocking infrastructure.

The 2 GSPS RF-ADC tile consists of four 2 GSPS converters, arranged in two pairs. Each of these converters can be configured individually for real input signals or, as a pair, for I/Q input signals. The following figure shows an overview of the 2 GSPS RF-ADC.

Figure 3: 2 GSPS RF-ADC Overview
The 4 GSPS RF-ADC tile consists of two 4 GSPS converters. These converters can be configured individually for real input signals or, as a pair, for I/Q input signals. The following figure shows an overview of the 4 GSPS RF-ADC.

**Figure 4: 4 GSPS RF-ADC Overview**

**RF-ADC Features**

- Tile configuration
  - Four RF-ADCs and one PLL per tile
  - 12-bit RF-ADC resolution, with 16-bit digital signal processing datapath
- Implemented as either four channels of 2 GSPS, or two channels of 4 GSPS (device dependent; for the actual sampling rate specifications, see the Zynq UltraScale+ RFSoC Data Sheet: DC and AC Switching Characteristics (DS926))

- Decimation filters
  - 1x (bypass filter), 2x, 4x, 8x
  - 80% of Nyquist bandwidth, 89 dB stop-band attenuation

- Digital Complex Mixers
  - Full complex mixers support real or I/Q inputs from the RF-ADC
  - 48-bit Numeric Controlled Oscillator (NCO) per RF-ADC
  - Fixed $F_s/4$, $F_s/2$ low power frequency mixing mode, where $F_s$ is the sample frequency
  - I/Q and real input signals supported

- Single/multi-band flexibility
  - 2x bands per 2 GSPS RF-ADC pair
  - Can be configured for real or I/Q inputs

- Full bandwidth of the RF-ADC at 4 GSPS can be accessed in bypass mode

- Input signal amplitude threshold: Two programmable threshold flags per RF-ADC

- Built-in digital correction for external analog quadrature modulators:
  - Supports gain, phase, and offset correction for an I/Q input pair (two RF-ADCs)

- SYSREF input signal for multi-channel synchronization

- Flexible AXI4-Stream interface supports a wide range of programmable logic clock rates and converter sample rates

- Per tile current-mode logic (CML) clock input buffer with on-chip calibrated 100 Ω termination; supplies the RF-ADC sampling clocks or provides a reference clock for the on-chip PLL

- Dedicated high-speed, high-performance, differential input buffer per RF-ADC with on-chip calibrated 100 Ω termination (on-die termination)

- Output common mode reference voltage for DC coupling RF-ADC inputs
RF-DAC

Each RF-DAC tile consists of four RF-DACs that can be configured individually for real output signals or, as a pair, for I/Q output signal generation. Each RF-DAC runs at a data rate of up to 6.5 GSPS. The RF-DAC tile has one PLL and a clocking instance. The following figure shows an overview of the RF-DAC.

**Figure 5: RF-DAC Overview**

**RF-DAC Features**
- Tile configuration
- Four RF-DACs and one PLL per tile
- 14-bit RF-DAC resolution with 16-bit digital signal processing path
- Sampling speed of 6.5 GSPS per RF-DAC (see the Zynq UltraScale+ RFSoC Data Sheet: DC and AC Switching Characteristics (DS926) for the exact sampling rate)
- 4 GHz full power output bandwidth

- Interpolation
  - 1x (bypass filter), 2x, 4x, 8x
  - 80% pass band, 89 dB stop band attenuation

- Digital Complex Mixers
  - Full complex mixers support real or I/Q output signals to the DACs
  - 48-bit NCO per RF-DAC
  - Fixed $F_s/4$, $F_s/2$ low-power frequency mixing mode
  - Supports mixed mode RF-DAC functionality which maximizes RF-DAC power in the second Nyquist zone

- Single/multi-band flexibility
  - 2x bands per RF-DAC pair
  - Can be configured for real or I/Q outputs

- Full bandwidth in bypass mode

- Digital Correction for external analog quadrature modulators:
  - Supports gain, phase, and offset correction for an I/Q output pair (two RF-DACs)
  - $\sin x/x$ correction for first Nyquist zone
  - External input signal (SYSREF) for multi-channel synchronization of data converter channels
  - Per tile current mode logic (CML) clock input buffer with on-chip calibrated 100 $\Omega$ termination; supplies the RF-DAC sampling clocks or provides a reference clock for the on-chip PLL
  - Supports 20 mA or 32 mA output power mode

For the RF-ADC and RF-DAC operating and absolute maximum/minimum parameters see:

- Zynq UltraScale+ RFSoC Data Sheet: Overview (DS889)
- Zynq UltraScale+ RFSoC Data Sheet: DC and AC Switching Characteristics (DS926)
**Applications**

- Multi-band, multi-mode 3G, 4G, and 5G cellular radios
- Cable infrastructure (DOCSIS 3.0 and DOCSIS 3.1)
- Software defined radios
- Microwave and millimeter wave radios
- Test and measurement applications

**Related Information**

**Applications**

---

**Licensing and Ordering**

This Xilinx® LogiCORE™ IP module is provided at no additional cost with the Xilinx® Vivado® under the terms of the Xilinx End User License.

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.
Chapter 3

Product Specification

The Zynq® UltraScale+™ RFSoC RF Data Converter IP core provides a way of instantiating all the RF-DAC and RF-ADC blocks in Zynq UltraScale+ RFSoCs in IP integrator. A single IP core instance allows access to all converters in the device.
An RF-ADC tile has two or four RF-ADCs; an RF-DAC tile can have four RF-DACs. The number of converters and the maximum sample rate depend on the package. The converters in each tile are the same type.
Performance

To see the performance of the RF-ADC and RF-DAC blocks, see the Zynq UltraScale+ RFSoC Data Sheet: DC and AC Switching Characteristics (DS926).

Maximum Frequencies

The `s_axi_aclk` is limited to between 10 MHz and 200 MHz. All other maximum clock frequencies are per the Zynq UltraScale+ RFSoC Data Sheet: DC and AC Switching Characteristics (DS926).

Latency

No latency is added by the core. See the Zynq UltraScale+ RFSoC Data Sheet: DC and AC Switching Characteristics (DS926) for latencies through the RF-DAC and RF-ADC.

Related Information
Clocking

Resource Use

For full details about performance and resource utilization, visit the Performance and Resource Utilization web page (registration required).

Port Descriptions

Configuration Interface Ports

<table>
<thead>
<tr>
<th>Table 1: Configuration Interface Ports</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Port Name</strong></td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td><code>s_axi_aclk</code></td>
</tr>
<tr>
<td><code>s_axi_ar[reset]</code></td>
</tr>
<tr>
<td><code>s_axi_aw[addr][17:0]</code></td>
</tr>
<tr>
<td><code>s_axi_awvalid</code></td>
</tr>
<tr>
<td><code>s_axi_awready</code></td>
</tr>
<tr>
<td><code>s_axi_wdata[31:0]</code></td>
</tr>
</tbody>
</table>
### Table 1: Configuration Interface Ports (cont’d)

<table>
<thead>
<tr>
<th>Port Name</th>
<th>I/O</th>
<th>Clock</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s_axi_wstrb[3:0]</td>
<td>In</td>
<td>s_axi_aclk</td>
<td>Write Data Byte Strobe</td>
</tr>
<tr>
<td>s_axi_wvalid</td>
<td>In</td>
<td>s_axi_aclk</td>
<td>Write Data Valid</td>
</tr>
<tr>
<td>s_axi_wready</td>
<td>Out</td>
<td>s_axi_aclk</td>
<td>Write Data Ready</td>
</tr>
<tr>
<td>s_axi_bresp[1:0]</td>
<td>Out</td>
<td>s_axi_aclk</td>
<td>Write Response</td>
</tr>
<tr>
<td>s_axi_bvalid</td>
<td>Out</td>
<td>s_axi_aclk</td>
<td>Write Response Valid</td>
</tr>
<tr>
<td>s_axi_bready</td>
<td>In</td>
<td>s_axi_aclk</td>
<td>Write Response Ready</td>
</tr>
<tr>
<td>s_axi_araddr[17:0]</td>
<td></td>
<td>s_axi_aclk</td>
<td>Read Address</td>
</tr>
<tr>
<td>s_axi_arvalid</td>
<td>In</td>
<td>s_axi_aclk</td>
<td>Read Address Valid</td>
</tr>
<tr>
<td>s_axi_arready</td>
<td>Out</td>
<td>s_axi_aclk</td>
<td>Read Address Ready</td>
</tr>
<tr>
<td>s_axi_rdata[31:0]</td>
<td>Out</td>
<td>s_axi_aclk</td>
<td>Read Data</td>
</tr>
<tr>
<td>s_axi_rresp[1:0]</td>
<td>Out</td>
<td>s_axi_aclk</td>
<td>Read Response</td>
</tr>
<tr>
<td>s_axi_rvalid</td>
<td>Out</td>
<td>s_axi_aclk</td>
<td>Read Data Valid</td>
</tr>
<tr>
<td>s_axi_rready</td>
<td>In</td>
<td>s_axi_aclk</td>
<td>Read Data Ready</td>
</tr>
<tr>
<td>irq</td>
<td>Out</td>
<td>s_axi_aclk</td>
<td>Interrupt output</td>
</tr>
</tbody>
</table>

### Multi Converter Synchronization Ports

### Table 2: Multi Converter Synchronization Ports

<table>
<thead>
<tr>
<th>Port Name</th>
<th>I/O</th>
<th>Clock</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sysref_in_p</td>
<td>In</td>
<td>N/A</td>
<td>External analog SYSREF input</td>
</tr>
<tr>
<td>sysref_in_n</td>
<td>In</td>
<td>N/A</td>
<td>External analog SYSREF input</td>
</tr>
<tr>
<td>user_sysref_adc</td>
<td>In</td>
<td>m0_axis_aclk</td>
<td>RF-ADC SYSREF input from programmable logic (PL)/ user design; synchronous to RF-ADC tile 0 PL clock</td>
</tr>
<tr>
<td>user_sysref_dac</td>
<td>In</td>
<td>s0_axis_aclk</td>
<td>RF-ADC SYSREF input from programmable logic (PL)/user design; synchronous to RF-DAC tile 0 PL clock</td>
</tr>
</tbody>
</table>

**Notes:**

1. Please see Multi-converter synchronization section in the applications sub-section for more information.

### Clock Ports Common to RF-DAC Tile

### Table 3: Clock Ports Common to RF-DAC Tile

<table>
<thead>
<tr>
<th>Port Name¹</th>
<th>I/O</th>
<th>Clock</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dacX_clk_p</td>
<td>In</td>
<td>N/A</td>
<td>RF-DAC on-chip PLL reference clock or sampling clock input</td>
</tr>
<tr>
<td>dacX_clk_n</td>
<td>In</td>
<td>N/A</td>
<td>RF-DAC on-chip PLL reference clock or sampling clock input</td>
</tr>
</tbody>
</table>
### Table 3: Clock Ports Common to RF-DAC Tile (cont’d)

<table>
<thead>
<tr>
<th>Port Name</th>
<th>I/O</th>
<th>Clock</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk_dacX</td>
<td>Out</td>
<td>N/A</td>
<td>Output clock to user logic</td>
</tr>
</tbody>
</table>

**Notes:**
1. X refers to the location of the tile in the converter column.

### AXI4-Stream Input Ports for RF-DACs

### Table 4: AXI4-Stream Input Ports for RF-DACs

<table>
<thead>
<tr>
<th>Port Name</th>
<th>I/O</th>
<th>Clock</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sX_axis_aclk</td>
<td>In</td>
<td>N/A</td>
<td>Clock input for RF-DAC data input</td>
</tr>
<tr>
<td>sXY_axis_arsetn</td>
<td>In</td>
<td>N/A</td>
<td>Synchronous reset for the sX_axis_aclk domain. This should be held low until sX_axis_aclk is stable.</td>
</tr>
<tr>
<td>sXY_axis_tdata[M:0]</td>
<td>In</td>
<td>sX_axis_aclk</td>
<td>AXI4-Stream data input</td>
</tr>
<tr>
<td>sXY_axis_tvalid</td>
<td>In</td>
<td>sX_axis_aclk</td>
<td>AXI4-Stream valid</td>
</tr>
<tr>
<td>sXY_axis_tready</td>
<td>Out</td>
<td>sX_axis_aclk</td>
<td>AXI4-Stream ready</td>
</tr>
<tr>
<td>voutXZ_p</td>
<td>Out</td>
<td>N/A</td>
<td>Analog output</td>
</tr>
<tr>
<td>voutXZ_n</td>
<td>Out</td>
<td>N/A</td>
<td>Analog output</td>
</tr>
</tbody>
</table>

**Notes:**
1. X refers to the location of the tile in the converter column. Y refers to the location of the DUC block in the tile (0 to 3). Z refers to the location of the RF-DAC in the tile (0 to 3). M is the number of samples per AXI4-Stream word * 16 for converter XY.

### Real-Time Signal Interface Ports for RF-DACs

### Table 5: Real-Time Signal Interface Ports for RF-DACs

<table>
<thead>
<tr>
<th>Port Name</th>
<th>I/O</th>
<th>Clock</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dacXZ_fast_shutdown[2:0]</td>
<td>In</td>
<td>N/A</td>
<td>RF-DAC fast shutdown</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>001 - Scale output data by 0.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>011 - Scale output data by 0.25</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>111 - Scale output data by 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Others - Normal operation</td>
</tr>
<tr>
<td>dacXY_pl_event</td>
<td>In</td>
<td>clk_dacX</td>
<td>RF-DAC PL event</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Assert to update RF-DAC settings from the PL</td>
</tr>
</tbody>
</table>

**Notes:**
1. X refers to the location of the tile in the converter column. Y refers to the location of the DUC block in the tile (0 to 3). Z refers to the location of the RF-DAC in the tile (0 to 3).
Clock Ports Common to RF-ADC Tile

Table 6: Clock Ports Common to RF-ADC Tile

<table>
<thead>
<tr>
<th>Port Name</th>
<th>I/O</th>
<th>Clock</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adcX_clk_p</td>
<td>In</td>
<td>N/A</td>
<td>RF-ADC on-chip PLL reference clock or sampling clock input</td>
</tr>
<tr>
<td>adcX_clk_n</td>
<td>In</td>
<td>N/A</td>
<td>RF-ADC on-chip PLL reference clock or sampling clock input</td>
</tr>
<tr>
<td>clk_adcX</td>
<td>Out</td>
<td>N/A</td>
<td>Output clock to user logic</td>
</tr>
</tbody>
</table>

Notes:
1. X refers to the location of the tile in the converter column.

AXI4-Stream Input Ports for RF-ADCs

Table 7: AXI4-Stream Input Ports for RF-ADCs

<table>
<thead>
<tr>
<th>Port Name</th>
<th>I/O</th>
<th>Clock</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mX_axis_aclk</td>
<td>In</td>
<td>N/A</td>
<td>Clock input for RF-ADC data output</td>
</tr>
<tr>
<td>mXY_axis_arstn</td>
<td>In</td>
<td>N/A</td>
<td>Synchronous reset for the sX_axis_aclk domain. This should be held low until mX_axis_aclk is stable.</td>
</tr>
<tr>
<td>mXY_axis_tdata[M:0]</td>
<td>Out</td>
<td>mX_axis_aclk</td>
<td>AXI4-Stream data output</td>
</tr>
<tr>
<td>mXY_axis_tvalid</td>
<td>Out</td>
<td>mX_axis_aclk</td>
<td>AXI4-Stream valid</td>
</tr>
<tr>
<td>mXY_axis_tready</td>
<td>In</td>
<td>mX_axis_aclk</td>
<td>AXI4-Stream ready</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2 GSPS</th>
<th>Port Name</th>
<th>I/O</th>
<th>Clock</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>vinXZ_p</td>
<td>In</td>
<td>N/A</td>
<td>Analog input</td>
</tr>
<tr>
<td></td>
<td>vinXZ_n</td>
<td>In</td>
<td>N/A</td>
<td>Analog input</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4 GSPS</th>
<th>Port Name</th>
<th>I/O</th>
<th>Clock</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>vinX ZZ_p</td>
<td>In</td>
<td>N/A</td>
<td>Analog input</td>
</tr>
<tr>
<td></td>
<td>vinX ZZ_n</td>
<td>In</td>
<td>N/A</td>
<td>Analog input</td>
</tr>
</tbody>
</table>

Notes:
1. X refers to the location of the tile in the converter column. Y refers to the location of the DDC block in the tile (0 to 3). In 2 GSPS devices, Z refers to the location of the RF-ADC in the tile (0 to 3). In 4 GSPS devices, ZZ is either 01 (the lower RF-ADC in the tile) or 23 (the upper RF-ADC in the tile). M is the number of samples per AXI4-Stream word * 16 for converter XY.

Real-Time Signal Interface Ports for 2 GSPS RF-ADCs

Table 8: Real-Time Signal Interface Ports for 2 GSPS RF-ADCs

<table>
<thead>
<tr>
<th>Port Name</th>
<th>I/O</th>
<th>Clock</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adcXY_pl_event</td>
<td>In</td>
<td>clk_adcX</td>
<td>RF-ADC PL event Assert to update RF-ADC settings from the PL</td>
</tr>
</tbody>
</table>
Table 8: Real-Time Signal Interface Ports for 2 GSPS RF-ADCs (cont’d)

<table>
<thead>
<tr>
<th>Port Name</th>
<th>I/O</th>
<th>Clock</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adcXZ_over_range</td>
<td>Out</td>
<td>Async</td>
<td>Over range output. A High on this output indicates that the signal exceeds the full-scale input of the RF-ADC.</td>
</tr>
<tr>
<td>adcXZ_over_threshold1</td>
<td>Out</td>
<td>mX_axis_aclk</td>
<td>Over threshold1 output Signal amplitude level is above programmable threshold 1</td>
</tr>
<tr>
<td>adcXZ_over_threshold2</td>
<td>Out</td>
<td>mX_axis_aclk</td>
<td>Over threshold2 output Signal amplitude level is above programmable threshold 2</td>
</tr>
<tr>
<td>adcXZ_over_voltage</td>
<td>Out</td>
<td>Async</td>
<td>Over voltage output An Over Voltage condition occurs when a signal far exceeds the normal operating input-range.</td>
</tr>
</tbody>
</table>

Notes:
1. X refers to the location of the tile in the converter column. Y refers to the location of the DDC block in the tile (0 to 3). Z refers to the location of the RF-ADC in the tile (0 to 3).
2. See RF-ADC Threshold and Over Range Settings for details on the real-time signals.

Related Information
RF-ADC Threshold and Over Range Settings

Real-Time Signal Interface Ports for 4 GSPS RF-ADCs

Table 9: Real-Time Signal Interface Ports for 4 GSPS RF-ADCs

<table>
<thead>
<tr>
<th>Port Name</th>
<th>I/O</th>
<th>Clock</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adcXY_pl_event</td>
<td>In</td>
<td>clk_adcX</td>
<td>RF-ADC PL event Assert to update RF-ADC settings from the PL</td>
</tr>
<tr>
<td>adcX ZZ over_range</td>
<td>Out</td>
<td>Async</td>
<td>Over range output. A High on this output indicates that the signal exceeds the full-scale input of the RF-ADC.</td>
</tr>
<tr>
<td>adcX ZZ over_threshold1</td>
<td>Out</td>
<td>mX_axis_aclk</td>
<td>Over threshold1 output Signal amplitude level is above programmable threshold 1</td>
</tr>
<tr>
<td>adcX ZZ over_threshold2</td>
<td>Out</td>
<td>mX_axis_aclk</td>
<td>Over threshold2 output Signal amplitude level is above programmable threshold 2</td>
</tr>
<tr>
<td>adcX ZZ over_voltage</td>
<td>Out</td>
<td>Async</td>
<td>Over voltage output An Over Voltage condition occurs when a signal far exceeds the normal operating input-range.</td>
</tr>
</tbody>
</table>

Notes:
1. X refers to the location of the tile in the converter column. Y refers to the location of the DDC block in the tile (0 to 3). ZZ is either 01 (the lower RF-ADC in the tile) or 23 (the upper RF-ADC in the tile)
2. See RF-ADC Threshold and Over Range Settings for details on the real-time signals.
Related Information
RF-ADC Threshold and Over Range Settings

Calibration Freeze Ports for 2 GSPS RF-ADCs

Table 10: Calibration Freeze Ports for 2 GSPS RF-ADCs

<table>
<thead>
<tr>
<th>Port Name1</th>
<th>I/O</th>
<th>Clock</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adcXY_int_cal_freeze</td>
<td>In</td>
<td>s_axi_aclk</td>
<td>Signal from the PL to indicate that IP should freeze the calibration. This is typically asserted when the RF-ADC output is below a certain threshold.</td>
</tr>
<tr>
<td>adcXY_cal_frozen</td>
<td>Out</td>
<td>s_axi_aclk</td>
<td>Asserted when the calibration is frozen.</td>
</tr>
</tbody>
</table>

Notes:
1. X refers to the location of the tile in the converter column. Y refers to the converter location in the tile (0 to 3).

Calibration Freeze Ports for 4 GSPS RF-ADCs

Table 11: Calibration Freeze Ports for 4 GSPS RF-ADCs

<table>
<thead>
<tr>
<th>Port Name1</th>
<th>I/O</th>
<th>Clock</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adcX ZZ_int_cal_freeze</td>
<td>In</td>
<td>s_axi_aclk</td>
<td>Signal from the PL to indicate that IP should freeze the calibration. This is typically asserted when the RF-ADC output is below a certain threshold.</td>
</tr>
<tr>
<td>adcX ZZ_cal_frozen</td>
<td>Out</td>
<td>s_axi_aclk</td>
<td>Asserted when the calibration is frozen.</td>
</tr>
</tbody>
</table>

Notes:
1. X refers to the location of the tile in the converter column. ZZ is either 01 (the lower RF-ADC in the tile) or 23 (the upper RF-ADC in the tile).

Register Space

The address map, shown in the following table, is split on a per-tile basis. All banks are 8 KB. The first bank contains the functions common to all tiles. Each tile has a bank for control and status.

Table 12: Address Space

<table>
<thead>
<tr>
<th>AXI4-Lite Address Range ADDR[17:0]</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000 - 0x03FFF</td>
<td>IP Common Control and Status</td>
</tr>
<tr>
<td>0x04000 - 0x07FFF</td>
<td>RF-DAC Tile 0 registers (Tile &lt;n&gt; Registers)</td>
</tr>
<tr>
<td>0x08000 - 0x0BFFF</td>
<td>RF-DAC Tile 1 registers (Tile &lt;n&gt; Registers)</td>
</tr>
<tr>
<td>0x0C000 - 0x0FFFF</td>
<td>RF-DAC Tile 2 registers (Tile &lt;n&gt; Registers)</td>
</tr>
</tbody>
</table>
Table 12: Address Space (cont’d)

<table>
<thead>
<tr>
<th>AXI4-Lite Address Range ADDR[17:0]</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10000 - 0x13FFF</td>
<td>RF-DAC Tile 3 registers (Tile &lt;n&gt; Registers)</td>
</tr>
<tr>
<td>0x14000 - 0x17FFF</td>
<td>RF-ADC Tile 0 registers (Tile &lt;n&gt; Registers)</td>
</tr>
<tr>
<td>0x18000 - 0x1BFFF</td>
<td>RF-ADC Tile 1 registers (Tile &lt;n&gt; Registers)</td>
</tr>
<tr>
<td>0x1C000 - 0x1FFFF</td>
<td>RF-ADC Tile 2 registers (Tile &lt;n&gt; Registers)</td>
</tr>
<tr>
<td>0x20000 - 0x23FFF</td>
<td>RF-ADC Tile 3 registers (Tile &lt;n&gt; Registers)</td>
</tr>
<tr>
<td>0x24000 - 0x3FFFF</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

IP Common Control and Status

Table 13: IP Common Control and Status

<table>
<thead>
<tr>
<th>Address Range ADDR[13:0]</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>IP Versioning Information</td>
</tr>
<tr>
<td>0x0004</td>
<td>Master Reset Register</td>
</tr>
<tr>
<td>0x0100</td>
<td>Common Interrupt Status Register</td>
</tr>
<tr>
<td>0x0104</td>
<td>Common Interrupt Enable Register</td>
</tr>
</tbody>
</table>

IP Version Information (0x0000)

Table 14: IP Version Information (0x0000)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Default Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:24</td>
<td>02</td>
<td>RO</td>
<td>Major</td>
</tr>
<tr>
<td>23:16</td>
<td>00</td>
<td>RO</td>
<td>Minor</td>
</tr>
<tr>
<td>15:8</td>
<td>00</td>
<td>RO</td>
<td>Revision</td>
</tr>
<tr>
<td>7:0</td>
<td>00</td>
<td>RO</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Master Reset Register (0x0004)

Table 15: Master Reset Register (0x0004)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Default Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:1</td>
<td>-</td>
<td>-</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
Table 15: Master Reset Register (0x0004) (cont’d)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Default Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>R/W Auto Clear</td>
<td>Reset All Tiles. Write 1 to this bit to reset all logic in the core and restart the power-on sequence of all converters in the core. Each converter is configured as per the settings chosen during core generation. The AXI4-Lite registers are unaffected by this reset with the exception of bits 15:8 in the Reset State Register for each tile which is set to 0 automatically. The end state (bits 7:0) is not affected by this reset so the power-up sequence for each tile starts from state 0 and runs to the value programmed in bits 7:0 of the Restart State register for each tile (the default end state is 0x0F).</td>
</tr>
</tbody>
</table>

Related Information
Restart State Register (0x0008)

Common Interrupt Status Register (0x0100)

Table 16: Common Interrupt Status Register (0x0100)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Default Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
<td>RO, Clear on Read</td>
<td>AXI timeout interrupt</td>
</tr>
<tr>
<td>30:8</td>
<td>-</td>
<td>RO</td>
<td>Reserved (read 0)</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>RO</td>
<td>RF-ADC Tile 3 interrupt</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>RF-ADC Tile 2 interrupt</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>RF-ADC Tile 1 interrupt</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>RF-ADC Tile 0 interrupt</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>RF-DAC Tile 3 interrupt</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>RO</td>
<td>RF-DAC Tile 2 interrupt</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>RF-DAC Tile 1 interrupt</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td>RF-DAC Tile 0 interrupt</td>
</tr>
</tbody>
</table>
Common Interrupt Enable Register (0x0104)

Table 17: Common Interrupt Enable Register (0x0104)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Default Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>0</td>
<td>R/W</td>
<td>AXI timeout interrupt enable</td>
</tr>
<tr>
<td>30:8</td>
<td>-</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td></td>
<td>RF-ADC Tile 3 interrupt enable</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td></td>
<td>RF-ADC Tile 2 interrupt enable</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td></td>
<td>RF-ADC Tile 1 interrupt enable</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td>RF-ADC Tile 0 interrupt enable</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td>RF-DAC Tile 3 interrupt enable</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td>RF-DAC Tile 2 interrupt enable</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>RF-DAC Tile 1 interrupt enable</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td>RF-DAC Tile 0 interrupt enable</td>
</tr>
</tbody>
</table>

Tile <n> Registers

Do not attempt to write to any tile specific registers while the power-on state machine is operating. To ensure the power-on state machine is not running prior to any register access, poll the Restart Power-On State Machine register (for tile <n>) and wait for it to read all zeros.

Table 18: Tile <n> Registers

<table>
<thead>
<tr>
<th>ADDR[12:0]</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x0004</td>
<td>Restart Power-On State Machine Register</td>
</tr>
<tr>
<td>0x0008</td>
<td>Restart State Register</td>
</tr>
<tr>
<td>0x000C</td>
<td>Current State Register</td>
</tr>
<tr>
<td>0x0010 - 0x00FC</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x0100</td>
<td>Post-Implementation Simulation Speedup Register</td>
</tr>
<tr>
<td>0x0104 - 0x01FC</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x0200</td>
<td>Interrupt Status Register</td>
</tr>
<tr>
<td>0x0204</td>
<td>Interrupt Enable Register</td>
</tr>
<tr>
<td>0x0208</td>
<td>Converter 0 Interrupt Register</td>
</tr>
<tr>
<td>0x020C</td>
<td>Converter 0 Interrupt Enable Register</td>
</tr>
<tr>
<td>0x0210</td>
<td>Converter 1 Interrupt Register</td>
</tr>
<tr>
<td>0x0214</td>
<td>Converter 1 Interrupt Enable Register</td>
</tr>
<tr>
<td>0x0218</td>
<td>Converter 2 Interrupt Register</td>
</tr>
<tr>
<td>0x021C</td>
<td>Converter 2 Interrupt Enable Register</td>
</tr>
<tr>
<td>0x0220</td>
<td>Converter 3 Interrupt Register</td>
</tr>
</tbody>
</table>
Table 18: Tile <n> Registers (cont’d)

<table>
<thead>
<tr>
<th>ADDR[12:0]</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0224</td>
<td>Converter 3 Interrupt Enable Register^1</td>
</tr>
<tr>
<td>0x0228</td>
<td>RF-DAC/RF-ADC Tile &lt;n&gt; Common Status Register</td>
</tr>
<tr>
<td>0x022C</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x0230</td>
<td>RF-DAC/RF-ADC Tile &lt;n&gt; Disable Register</td>
</tr>
<tr>
<td>0x0234-0x3FFF</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Notes:
1. Converter 2 and 3 registers are not applicable for tiles with 4 GS/s RF-ADCs.

Restart Power-On State Machine Register (0x0004)

Table 19: Restart Power-On State Machine Register (0x0004)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Default Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:1</td>
<td>-</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>R/W Auto Clear</td>
<td>Write 1 to this bit to start the power-on state machine. The state machine starts and stops at the stages programmed in the Restart State Register. This bit stays High until the state machine has reached the chosen end state.</td>
</tr>
</tbody>
</table>

Related Information
Restart State Register (0x0008)

Restart State Register (0x0008)

Table 20: Restart State Register (0x0008)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Default Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:16</td>
<td>-</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>15:8</td>
<td>00</td>
<td>R/W</td>
<td>Start Enabled tiles only. Start and End states for the power-on sequence. The default start state of 0x00 and end state of 0xF should be used to enable the converters and a start state of 0x00 and an end state of 0x03 should be used to stop the converters. When a 1 is written to the bit in the Restart Power-on State Machine Register for tile&lt;n&gt;, the power-on state machine is started from the start state and runs to the end of the end state specified in this register. See Power-up Sequence for details about restarting and power-down.</td>
</tr>
<tr>
<td>7:0</td>
<td>0F</td>
<td>R/W</td>
<td>End</td>
</tr>
</tbody>
</table>

Related Information
Power-up Sequence
Current State Register (0x000C)

Table 21: Current State Register (0x000C)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Default Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:8</td>
<td>-</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>7:0</td>
<td>00</td>
<td>RO</td>
<td>Current state of Power-on state machine. See Power-on Sequence Steps.</td>
</tr>
</tbody>
</table>

Related Information

Power-on Sequence Steps

Post-Implementation Simulation Speedup Register (0x0100)

Table 22: Post-Implementation Simulation Speedup Register (0x0100)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Default Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:1</td>
<td>-</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>RW</td>
<td>Simulation speed-up for post-implementation simulations. Set High to speed up post-implementation simulations. This register must not be set in hardware.</td>
</tr>
</tbody>
</table>

Interrupt Status Register (0x0200)

Table 23: Interrupt Status Register (0x0200)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Default Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:4</td>
<td>-</td>
<td>-</td>
<td>Reserved (read back 0)</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>RO</td>
<td>Converter 3 interrupt bit</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td>Converter 2 interrupt bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>Converter 1 interrupt bit</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td>Converter 0 interrupt bit</td>
</tr>
</tbody>
</table>

Interrupt Enable Register (0x0204)

Table 24: Interrupt Enable Register (0x0204)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Default Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:4</td>
<td>-</td>
<td>-</td>
<td>Reserved Enable Read back 0</td>
</tr>
</tbody>
</table>
### Table 24: Interrupt Enable Register (0x0204) (cont’d)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Default Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>R/W</td>
<td>Converter 3 interrupt enable bit</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>R/W</td>
<td>Converter 2 interrupt enable bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>R/W</td>
<td>Converter 1 interrupt enable bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>R/W</td>
<td>Converter 0 interrupt enable bit</td>
</tr>
</tbody>
</table>

### Converter 0 Interrupt Register (0x0208)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Default Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:16</td>
<td>-</td>
<td>-</td>
<td>Reserved Enable Read back 0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>RO Clear on Reset</td>
<td>Flags a FIFO overflow in converter when High</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>-</td>
<td>Flags a datapath overflow in converter when High¹</td>
</tr>
<tr>
<td>13:4</td>
<td>-</td>
<td>-</td>
<td>Reserved (read back 0)</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>RO Clear on Reset</td>
<td>Flags an Over Range interrupt in converter²</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>RO Clear on Read</td>
<td>Flags an Over Voltage interrupt in converter²</td>
</tr>
<tr>
<td>1:0</td>
<td>N/A</td>
<td>-</td>
<td>Reserved (read back 0)</td>
</tr>
</tbody>
</table>

**Notes:**

1. A datapath overflow indicates one of the following conditions has occurred:
   - Interpolation filter overflow in the RF-DAC
   - Decimation filter overflow in the RF-ADC
   - Overflow in the Quadrature Modulation Correction block
   - Overflow in the RF-DAC Inverse Sinc filter
2. RF-ADC only.

### Converter 0 Interrupt Enable Register (0x020C)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Default Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:16</td>
<td>-</td>
<td>-</td>
<td>Reserved (read back 0)</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>R/W</td>
<td>Enable FIFO overflow interrupt in converter</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>R/W</td>
<td>Enable datapath overflow interrupt in converter</td>
</tr>
<tr>
<td>13:4</td>
<td>-</td>
<td>-</td>
<td>Reserved (read back 0)</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>R/W</td>
<td>Enable Over Range interrupt in converter¹</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>R/W</td>
<td>Enable Over Voltage interrupt in converter¹</td>
</tr>
</tbody>
</table>
### Table 25: Converter 0 Interrupt Enable Register (0x020C) (cont’d)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Default Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:0</td>
<td>-</td>
<td>-</td>
<td>Reserved (read back 0)</td>
</tr>
</tbody>
</table>

**Notes:**
1. RF-ADC only.

### Converter 1 Interrupt Register (0x0210)

### Table 26: Converter 1 Interrupt Register (0x0210)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Default Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:16</td>
<td>-</td>
<td>-</td>
<td>Reserved (read back 0)</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>RO Clear on Reset</td>
<td>Flags a FIFO overflow in converter when High</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>RO Clear on Reset</td>
<td>Flags a datapath overflow in converter when High</td>
</tr>
<tr>
<td>13:4</td>
<td>-</td>
<td>-</td>
<td>Reserved (read back 0)</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>RO Clear on Reset</td>
<td>Flags an Over Range interrupt in converter¹</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>RO Clear on Read</td>
<td>Flags an Over Voltage interrupt in converter¹</td>
</tr>
<tr>
<td>1:0</td>
<td>-</td>
<td>-</td>
<td>Reserved (read back 0)</td>
</tr>
</tbody>
</table>

**Notes:**
1. RF-ADC only.

### Converter 1 Interrupt Enable Register (0x0214)

### Table 27: Converter 1 Interrupt Enable Register (0x0214)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Default Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:16</td>
<td>-</td>
<td>-</td>
<td>Reserved (read back 0)</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>R/W</td>
<td>Enable FIFO overflow interrupt in converter</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>R/W</td>
<td>Enable datapath overflow interrupt in converter</td>
</tr>
<tr>
<td>13:4</td>
<td>-</td>
<td>-</td>
<td>Reserved (read back 0)</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>R/W</td>
<td>Enable Over Range interrupt in converter¹</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>R/W</td>
<td>Enable Over Voltage interrupt in converter¹</td>
</tr>
<tr>
<td>1:0</td>
<td>-</td>
<td>-</td>
<td>Reserved (read back 0)</td>
</tr>
</tbody>
</table>

**Notes:**
1. RF-ADC only.
### Converter 2 Interrupt Register (0x0218)

**Table 28: Converter 2 Interrupt Register (0x0218)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Default Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:16</td>
<td>-</td>
<td>-</td>
<td>Reserved (read back 0)</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>RO Clear on Reset</td>
<td>Flags a FIFO overflow in converter when High</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td></td>
<td>Flags a datapath overflow in converter when High</td>
</tr>
<tr>
<td>13:4</td>
<td>-</td>
<td>-</td>
<td>Reserved (read back 0)</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>RO Clear on Reset</td>
<td>Flags an Over Range interrupt in converter(^1)</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>RO Clear on Read</td>
<td>Flags an Over Voltage interrupt in converter(^1)</td>
</tr>
<tr>
<td>1:0</td>
<td>-</td>
<td>-</td>
<td>Reserved (read back 0)</td>
</tr>
</tbody>
</table>

**Notes:**
1. RF-ADC only.

### Converter 2 Interrupt Enable Register (0x021C)

**Table 29: Converter 2 Interrupt Enable Register (0x021C)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Default Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:16</td>
<td>-</td>
<td>-</td>
<td>Reserved (read back 0)</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>R/W</td>
<td>Enable FIFO overflow interrupt in converter</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>R/W</td>
<td>Enable datapath overflow interrupt in converter</td>
</tr>
<tr>
<td>13:4</td>
<td>-</td>
<td>-</td>
<td>Reserved (read back 0)</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>R/W</td>
<td>Enable Over Range interrupt in converter(^1)</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>R/W</td>
<td>Enable Over Voltage interrupt in converter(^1)</td>
</tr>
<tr>
<td>1:0</td>
<td>-</td>
<td>-</td>
<td>Reserved (read back 0)</td>
</tr>
</tbody>
</table>

**Notes:**
1. RF-ADC only.

### Converter 3 Interrupt Register (0x0220)

**Table 30: Converter 3 Interrupt Register (0x0220)**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Default Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:16</td>
<td>-</td>
<td>-</td>
<td>Reserved (read back 0)</td>
</tr>
</tbody>
</table>
### Table 30: Converter 3 Interrupt Register (0x0220) (cont’d)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Default Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>0</td>
<td>RO Clear on Reset</td>
<td>Flags a FIFO overflow in converter when High</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>-</td>
<td>Flags a datapath overflow in converter when High</td>
</tr>
<tr>
<td>13:4</td>
<td>-</td>
<td>-</td>
<td>Reserved (read back 0)</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>RO Clear on Reset</td>
<td>Flags an Over Range interrupt in converter¹</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>RO Clear on Read</td>
<td>Flags an Over Voltage interrupt in converter¹</td>
</tr>
<tr>
<td>1:0</td>
<td>-</td>
<td>-</td>
<td>Reserved (read back 0)</td>
</tr>
</tbody>
</table>

**Notes:**
1. RF-ADC only.

### Converter 3 Interrupt Enable Register (0x0224)

### Table 31: Converter 3 Interrupt Enable Register (0x0224)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Default Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:16</td>
<td>-</td>
<td>-</td>
<td>Reserved (read back 0)</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>R/W</td>
<td>Enable FIFO overflow interrupt in Converter</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>-</td>
<td>Enable datapath overflow interrupt in Converter</td>
</tr>
<tr>
<td>13:4</td>
<td>-</td>
<td>-</td>
<td>Reserved (read back 0)</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>R/W</td>
<td>Enable Over Range interrupt in converter¹</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>R/W</td>
<td>Enable Over Voltage interrupt in converter¹</td>
</tr>
<tr>
<td>1:0</td>
<td>-</td>
<td>-</td>
<td>Reserved (read back 0)</td>
</tr>
</tbody>
</table>

**Notes:**
1. RF-ADC only.
RF-DAC/RF-ADC Tile <n> Common Status Register (0x0228)

Table 32: RF-DAC/RF-ADC Tile <n> Common Status Register (0x0228)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Default Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:4</td>
<td>-</td>
<td>RO</td>
<td>Reserved</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>RO</td>
<td>PLL locked. Asserted when the tile PLL has achieved lock.</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>RO</td>
<td>Power-up state. Asserted when the tile is in operation.</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>RO</td>
<td>Supplies up. Asserted when the external supplies to the tile are stable.</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>RO</td>
<td>Clock present. Asserted when the reference clock for the tile is present.</td>
</tr>
</tbody>
</table>

Notes:
1. <n> is 0 to 3.
2. See Register Space for register <n> address.

Related Information
Register Space

RF-DAC/RF-ADC Tile <n> FIFO Disable Register (0x0230)

Table 33: RF-DAC/RF-ADC Tile <n> FIFO Disable Register (0x0230)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Default Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:1</td>
<td>0</td>
<td>R/W</td>
<td>Reserved</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>R/W</td>
<td>Disable the interface FIFO for converter &lt;n&gt;</td>
</tr>
</tbody>
</table>

Notes:
1. <n> is 0 to 3
2. See Register Space for register <n> address.

Related Information
Register Space
Designing with the Core

The RF Data Converter solution consists of the Zynq® UltraScale+™ RFSoC RF Data Converter IP core configuration in the Vivado® Integrated Design Environment (IDE) and the RF data converter (RFdc) driver Application Programming Interface (API).

IP Core Configuration in Vivado Design Suite

The Zynq® UltraScale+™ RFSoC RF Data Converter IP core configuration screen in the Vivado® IDE sets up the physical configuration of the RF-ADCs and RF-DACs in the RFSoC. The configuration screen is used to enable tiles, configure decimation, interpolation, and mixing, set up converter sample rates, the Programmable Logic (PL) interface word widths and data types, and enables the optional interface ports. The IP core also handles the configuration and power-up of the data converters. This ensures that the settings specified in the Vivado IDE are applied to the RF-ADCs and RF-DACs immediately after the PL configuration completes.

Software Driver

The RFdc driver API provides runtime interaction and monitoring of the data converters. This includes responding to interrupts, changing some settings, such as mixer frequency, and also interacting with the IP core to power up or down RF-ADC or RF-DAC tiles. The RFdc driver API is available as a bare-metal or Linux driver validated on both MicroBlaze™ and RFSoC Processing System (PS) APU or RPU processors.

RF-ADC

Every RF-ADC in a tile has its own dedicated high-performance input buffer and includes features optimized for direct conversion applications including quadrature modulator correction (QMC), full complex mixers, and decimation filters.
Certain functions can only be executed when the RF-ADCs in a tile are paired. The even-numbered RF-ADCs are used for I datapaths and the odd-numbered RF-ADCs are used for Q datapaths. All of the available built-in functionality of a tile and each of the RF-ADCs within a tile are configured with the supporting RFdc driver API and/or core configuration screen in the Vivado® IDE.

Related Information
Quadrature Modulator Correction

RF-ADC Analog Input

Every RF-ADC in a tile has its own differential analog input buffer. This input is optimized for performance and requires source impedance matching for best dynamic performance.

There are several ways to drive an RF-ADC in a tile. Driving the RF-ADC can be either active or passive. However, optimum performance is achieved by driving the analog input differentially. Using a differential amplifier, AC or DC coupled, to drive the RF-ADC provides a flexible interface with excellent performance. For AC coupled mode, the input signals should be AC coupled in using capacitors. For DC coupled mode, the output VCM buffer is enabled as shown in the figure above. This buffer is only enabled when DC coupled mode is selected. This VCM buffer allows...
the user to align the common mode of the external active driving circuit with the ADC internal common requirements. Two VCM buffers (VCM01, VCM23) are available for each tile. For all applications where the signal-to-noise ratio (SNR) is a key parameter, Xilinx recommends using a differential transformer or balun configuration. See the *UltraScale Architecture PCB Design User Guide (UG583)* for details on how to design the input networks and PCB.

**RF-ADC Digital Datapath**

An RF-ADC component in a tile has integrated DSP features which can be enabled by the user to pre-process the sampled data from the RF-ADC device before it is passed to the PL. The different DSP function blocks are as follows:

- **Detection functionality**—containing a dual level programmable threshold that provides two flags to the internal interconnect logic, and is asserted when the absolute value of the RF-ADC is greater or smaller than the programmed threshold values.
- **Compensation functionality**—containing a quadrature modulator correction (QMC) block with a coarse delay adjustment block
- **Digital down converter (DDC)**—consists of mixing followed by decimation
- **Mixer**—coarse (quarter and half rate) and fine (NCO with 48-bit frequency resolution)
- **Signal decimation functionality**—decimation by 2, 4, or 8 is supported

Single, multiple, or all DSP functions can be used or bypassed. Some functions, such as the QMC, require activation of the same function in both I and Q RF-ADCs. Even numbered RF-ADCs are always used for I datapaths, and odd numbered RF-ADCs are used for Q datapaths. You can implement, configure, or modify the functionality of one or multiple functions using the IP core. The following figure shows the available functions in an RF-ADC and the functions are described in this section.
**RF-ADC Threshold and Over Range Settings**

As with any ADC, the input analog signals must be kept within the full-scale range of the ADC, and at the correct input levels. Any signals that do not comply with these conditions result in data loss. To help prevent this, the threshold-detector feature can be used to adjust external gains to keep the signal within the ideal full-scale ranges. However, in the event that a signal does exceed the full-scale range, each RF-ADC channel has built-in detection and protection for this with the Over Range and Over Voltage features.

The Over Voltage and Over Range signals are provided to both the interrupt feature on the IP core, as well as to the RF-ADC real-time signals bus on the IP core for direct access to the PL design.

**Over Range**

An Over Range condition occurs when a signal exceeds the full-scale input of the RF-ADC. When this condition is detected, the converted data is saturated (clipped) to limit the data corruption, and the signal is flagged to the user by both the interrupt mechanism, and by asserting the `adcXY_over_range` real-time output signal. Because Over Range events can be as short as one RF-ADC sample, the output signal is sticky. To clear the Over Range output and the associated interrupt, the API interrupt handling mechanism is used.

**Related Information**

Interrupt Handling
Over Voltage

An Over Voltage condition occurs when a signal far exceeds the normal operating input range. Because an excessive voltage on the inputs can damage the input buffers, an Over Voltage event results in the automatic shutdown of the input buffer to protect it. The Over Voltage circuit monitors each of the signals of the differential inputs independently, and flags the condition when any individual input signal exceeds the maximum input voltage or is less than the minimum input voltage of the RF-ADC input buffer.

The Over Voltage feature offers protection for signals in the range defined in the Zynq UltraScale+ RFSoC Data Sheet: DC and AC Switching Characteristics (DS926). Signals exceeding this maximum are not permitted, and care must be taken externally to ensure that such voltages are not presented to the RF-ADC inputs.

When an Over Voltage condition is detected the signal is flagged to the user by both the interrupt mechanism, and by asserting the \texttt{adcXY\_over\_voltage} real-time output signal. The Over Voltage real-time output is asserted and deasserted asynchronously, and provides immediate notification of the event. As a result, the Over Voltage output self-clears when the Over Voltage condition is no longer present. The associated interrupt is sticky, so requires clearing by the API interrupt handling routines.

After an Over Voltage event, the input buffer automatically re-enables, and the RF-ADC resumes operation as before. Because the data produced by the RF-ADC must pass through the digital datapath, there is a latency of 20 ns + the datapath latency, before valid RF-ADC data is generated after the deassertion of the \texttt{adcXY\_over\_voltage} output.

The following figure illustrates the Threshold, Over Range, and Over Voltage levels and the response of these with an increasing input analog signal.
Related Information

Interrupt Handling

Threshold Settings

Instead of waiting for a signal to propagate through the signal processing blocks, the threshold feature provides an early indication of the incoming signal level. This early indication of the signal level can be used by the automatic gain control (AGC) implemented in the PL. Threshold levels used to indicate the input signal level are set using the RFdc driver API.

Threshold monitoring occurs when the RF-ADC sampled data enters the datapath. This data is compared to a user-defined threshold. A threshold status signal is sent to the outputs on the IP core, to indicate that a user-defined threshold has been exceeded. The outputs are called `adcXY_over_threshold1` and `adcXY_over_threshold2`. The modes of the threshold monitoring circuit are listed in the following table.

Table 34: Threshold Signaling Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Off</td>
<td>The threshold circuit is disabled and the status outputs are Low.</td>
</tr>
<tr>
<td>Sticky over</td>
<td>The threshold status signal is High when the data from the RF-ADC exceeds the programmed upper threshold value. The status is kept until a clear action is sent.</td>
</tr>
</tbody>
</table>

---

Chapter 4: Designing with the Core

PG269 (v2.0) April 17, 2018

Zynq UltraScale+ RFSoC RF Data Converter

www.xilinx.com
### Table 34: Threshold Signaling Modes (cont’d)

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sticky under</td>
<td>The threshold status signal is High when the data from the RF-ADC remains below the programmed lower threshold value for the duration of a user-specified time or delay. The status is kept until a clear action is sent. This delay value for the lower threshold is defined by a 32-bit counter. The counter is set using the RFdc driver API. Using this mechanism prevents short duration excursions triggering a threshold event.</td>
</tr>
<tr>
<td>Hysteresis</td>
<td>The status output is set when the programmed upper threshold value is exceeded, and is cleared when the signal remains below the lower threshold value for the duration of a user-specified delay value. This delay value for the lower threshold is defined by a 32-bit counter. The counter is set using the RFdc driver API. The delay adds hysteresis to the threshold detection to prevent short duration excursions triggering a threshold event.</td>
</tr>
</tbody>
</table>

**Notes:**

1. To clear see Clearing Threshold Flags.

Threshold levels are set as 14-bit unsigned values, with any value from 0 to 16383 allowed. The maximum value, 16383 represents the absolute value of the full-scale input of the RF-ADC. The 32-bit programmable delay counts RF-ADC samples. To relate this count to a specific time, the following formulas can be used:

\[
4 \text{ GSPS Tile Delay Value} = \frac{\text{delay}(ms) \times \text{ADC}_{\text{SampleRate}} \times (\text{GSPS}) \times 10^4}{8}
\]

\[
2 \text{ GSPS Tile Delay Value} = \frac{\text{Delay}(ms) \times \text{ADC}_{\text{SampleRate}}(\text{GSPS}) \times 10^4}{4}
\]

**Clearing Threshold Flags**

After the threshold values assert in the Sticky Over and Sticky Under modes, the flags can be cleared in two ways: by issuing the clear command directly, or by using the AutoClear feature. The AutoClear mode clears the threshold whenever the QMC Gain value associated with that RF-ADC is updated.

**Threshold Operation Example**

The following figure illustrates the threshold unit operation. The diagram shows three thresholds, a, b, and c, each configured in a different threshold mode. In hardware all threshold units have independent threshold levels and delay values. In this example the threshold levels are indicated by the dashed horizontal lines, and all units use the same levels for the purpose of illustration. The delay value is set to 5, and the sticky thresholds have been set up to be cleared by different clear modes. The behavior is described in the following sections.
Threshold-A, Sticky Over Mode

The flag asserts when the first sample exceeds the upper threshold value. The flag is cleared using a call to the `XRFdc_ThresholdStickyClear` API function.

Threshold-B, Sticky Under Mode

The flag asserts when the samples have been continuously below the lower threshold for the delay number of samples (6, in the example below). The flag is cleared through an update of the QMC Gain value, which is issued using a call to the `XRFdc_SetQMCSettings` API function, followed by an Update event, which applies the QMC gain update.

Threshold-C, Hysteresis Mode

The flag asserts when the first sample exceeds the upper threshold value. The flag is cleared when the samples have been continuously below the lower threshold for the delay number of samples.

Related Information

Dynamic Update Events
Threshold RFdc Driver API Commands

The threshold levels and delay values are configured using the RFdc driver API.

```c
// Initial Setup
XRFdc_Threshold_Settings Threshold_Settings;
Threshold_Settings.UpdateThreshold = XRFDC_UPDATE_THRESHOLD_BOTH; // Setup values
for threshold 0 and 1
Threshold_Settings.ThresholdMode[0] = XRFDC_TRSHD_STICKY_UNDER; // Set threshold0
to Sticky Under
Threshold_Settings.ThresholdUnderVal[0] = 1000; // Measured in 14-bit unsigned LSBs
Threshold_Settings.ThresholdAvgVal[0] = 10; // Data must be below lower threshold for
10*8 4 GSPS RF-ADC samples

// Write threshold values to the selected Tile / RF-ADC
XRFdc_SetThresholdSettings(ptr, Tile, Block, &Threshold_Settings);
```

The threshold clear operation is shown in the following code examples.

- Clear the thresholds by writing directly using the RFdc driver API:

```c
// Initial Setup
XRFdc_SetThresholdClrMode(ptr, Tile, Block, Threshold#, XRFDC_THRESHOLD_CLRMD_MANUAL_CLR);
....
// During application run-time (after a threshold asserts)
XRFdc_ThresholdStickyClear(ptr, Tile, Block, Threshold#);
```

- Clear the thresholds using the AutoClear function; the threshold clears with the QMC Gain Update:

```c
// Initial Setup
XRFdc_QMC_Settings QMC_Settings;
XRFdc_SetThresholdClrMode(ptr, Tile, Block, Threshold#, XRFDC_THRESHOLD_CLRMD_AUTO_CLR);
....
// During application run-time (after a threshold asserts - update t)
QMC_Settings.GainCorrectionFactor = new_gain_value;
XRFdc_SetQMCSettings(ptr, XRFDC_ADC_TILE, Tile, Block, &QMC_Settings);
....
// QMC Gain applied by an Update Event
```
Related Information
XRFdc_SetThresholdSettings
XRFdc_SetThresholdClrMode
XRFdc_ThresholdStickyClear
XRFdc_SetQMCSettings

Threshold Applications

A common use for threshold detectors is in Automatic Gain Control (AGC) applications.

Related Information
Automatic Gain Control Systems

Over Range Settings

There are two types of Over Range signals available as status outputs to the programmable logic, Over Voltage and Over Range. The IP core exposes these signals using the interrupt mechanism, so any violation is immediately flagged to the user application. These are triggered when normal or expected operating levels are exceeded.

Over Voltage Signal

An Over Voltage signal is detected whenever the input signal exceeds a safe input range from the RF-ADC input buffers.

Over Range Signal

- When the input signal exceeds the ± digital full-scale range of an RF-ADC, an Over Range signal is detected.
- An Over Range signal is measured at the raw digital output of the RF-ADC.
**RF-ADC Mixer with Numerical Controlled Oscillator**

The mixer function has three modes: bypass (no mixing), coarse mixing or fine mixing. Fine mixing automatically enables the NCO which is used to generate the carrier frequency. The mixer supports full quadrature mixing, with both real to I/Q and I/Q to I/Q modes supported.

*Figure 12: RF-ADC Mixer with NCO DSP Block*

**Coarse Mixer:**
- The coarse mixer allows the data to be mixed with a carrier of 0, Fs/2, Fs/4, or -Fs/4.
- Mixing with a 0 carrier bypasses the mixer component.

**Fine Mixer:**
- The fine mixer allows the data to be shifted up or down in frequency by an arbitrary amount.
- The frequency shift amount is obtained by programming the mixer frequency generated in the NCO. The fine mixer also supports 18-bit phase adjustment.
The NCO phase can be synchronized within a tile using `XRFdc_UpdateEvent`.

The NCO phase can be synchronized across tiles using an external event signal (SYSREF or MARKER).

To manage potential overflow, the fine mixer output includes 3 dBV attenuation, as shown in the figure above. This attenuation is not relevant in R2C mode, so the automatic mode selection from the API selects the correct attenuation level following the RF-ADC mixer scaling output factor (see the table below). A manual selection is also possible, allowing 0 dBV or -3 dBV.

<table>
<thead>
<tr>
<th>Tile Usage</th>
<th>Coarse Mixer</th>
<th>Auto Fine Mixer</th>
</tr>
</thead>
<tbody>
<tr>
<td>IQ (C2C)</td>
<td>1 (0 dBV)</td>
<td>0.707 (-3 dBV)</td>
</tr>
<tr>
<td>Real (R2C)</td>
<td>1 (0 dBV)</td>
<td>0.997 (~0 dBV)</td>
</tr>
</tbody>
</table>

The mixer settings can be configured in the core, or by using the RFdc driver API. The core is used to set the initial mixer settings (for example, mixer type and mixer mode), and the RFdc driver API is used to adjust the settings at runtime. Both the RFdc driver API and the core compute the required register settings based on the supplied sample rates and desired frequencies. A sample configuration screen is shown in the following figure. See the RF-ADC Converter Configuration section for information on the settings.
Figure 13: RF-ADC Mixer Settings Configuration

ADC 0

- Enable ADC
- Invert Q Output
- Dither

Data Settings
- Digital Output Data: UQ
- Decimation Mode: 1x
- Samples per AXI4-Stream Word: 8
- Required AXI4-Stream clock: 250,000 MHz

Mixer Settings
- Mixer Type: Fine
- Mixer Mode: Real->IQ
- NCO Frequency (GHz): 0.0
- NCO Phase: 0

Analog Settings
- Nyquist Zone: Zone 1
- Calibration Mode: Mode2

Related Information
XRFdc_UpdateEvent
RF-ADC Converter Configuration
RF-ADC Mixer RFdc API Example

Related RFdc driver API functions are shown in the following code. This code illustrates the use of the NCO Phase reset function. This function must be used at startup to initialize the phase of the fine mixer to a valid state. Note that the following code resets the NCOs in all tiles.

```c
XRFdc_Mixer_Settings Mixer_Settings;

for(tile=0; tile<4; tile++) {
    // Make sure the mixer settings update use the Tile event
    for(block=0; block<2; block++) {
        XRFdc_GetMixerSettings (ptr, XRFDC_ADC_TILE, tile, block, &Mixer_Settings);
        Mixer_Settings.EventSource = XRFDC_EVNT_SRC_TILE; // Mixer Settings are updated with a tile event
        XRFdc_SetMixerSettings (ptr, XRFDC_ADC_TILE, tile, block, &Mixer_Settings);
    }

    // Reset NCO phase of both DDCs in Tile0 (assuming both are active)
    XRFdc_ResetNCOPhase(ptr, XRFDC_ADC_TILE, tile, 0); // DDC Block0
    XRFdc_ResetNCOPhase(ptr, XRFDC_ADC_TILE, tile, 1); // DDC Block1

    XRFds_UpdateEvent(ptr, XRFDC_ADC_TILE, tile, 1, XRFDC_EVENT_MIXER); // Generate a Tile Event
}
```

Related Information
- XRFdc_GetMixerSettings
- XRFdc_SetMixerSettings
- XRFdc_ResetNCOPhase

RF-ADC Decimation Filters

Decimation filters are required to implement the down-sampling and filtering part of the digital-down conversion (DDC) process. The overall filter response is determined by the number of decimation stages used. The decimation chain consists of three FIR filter stages which can be combined to implement variable decimation rates. When a FIR stage is not used it is automatically powered down. The decimation filters allow for the creation of the following:

- **1x**: All filter stages are bypassed
- **2x**: Decimation filtering using a single stage
- **4x**: Decimation filtering using two stages
- **8x**: Decimation filtering using all three available stages

Each decimation filter element has a different number of taps and the stop-band attenuation and ripple are shown in Decimation Filter Details. The decimation filter chains can operate on either I/Q data or real data. Unused filter chains are powered down.
Each of the filter stages can overflow given the step-response of a FIR filter, especially when full-scale data is on the input. To detect and protect the datapath from overflow, each filter stage and sub-phase has a signed overflow status signal and saturation at the output. When a filter stage is not used, the flag is forced zero. These flags are connected to the datapath interrupt mechanism which is described in Interrupt Handling. The multiplexer in the following figure shows the decimation level selected in the IP configuration with the corresponding selection of decimation filter blocks.

![RF-ADC Decimation Filter](image)

**Figure 14**: RF-ADC Decimation Filter

### Table 36: Decimation Filter Operating Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>2 GSPS</strong></td>
<td></td>
</tr>
<tr>
<td>OFF</td>
<td>The entire filter is disabled/powered down (applies when RF-ADC is disabled)</td>
</tr>
<tr>
<td>1x</td>
<td>The entire filter is bypassed</td>
</tr>
<tr>
<td>2x</td>
<td>2x decimation, 80% Nyquist passband¹</td>
</tr>
<tr>
<td>4x</td>
<td>4x decimation, 80% Nyquist passband</td>
</tr>
<tr>
<td>8x</td>
<td>8x decimation, 80% Nyquist passband</td>
</tr>
<tr>
<td><strong>4 GSPS</strong></td>
<td></td>
</tr>
<tr>
<td>OFF</td>
<td>The entire filter is disabled/powered down (applies when RF-ADC is disabled)</td>
</tr>
<tr>
<td>1x</td>
<td>The entire filter is bypassed</td>
</tr>
<tr>
<td>2x</td>
<td>2x decimation, 80% Nyquist passband</td>
</tr>
<tr>
<td>4x</td>
<td>4x decimation, 80% Nyquist passband</td>
</tr>
<tr>
<td>8x</td>
<td>8x decimation, 80% Nyquist passband</td>
</tr>
</tbody>
</table>

**Notes:**

1. 80% Nyquist passband is 0.4*Fs
Related Information
Interrupt Handling
Decimation Filter Details

Decimation Filter Use

The IP core is used to set the decimation rate. This is set in the Vivado® IDE because changing the decimation rate directly affects the physical interface as the bandwidth to the PL changes. The filters that are enabled are as shown in the following figure. Enable the RF-ADC by checking the Enable ADC checkbox. See RF-ADC Converter Configuration for information on the settings.

*Figure 15: RF-ADC Decimation Filter Configuration*

![RF-ADC Decimation Filter Configuration](image)

Related Information
RF-ADC Converter Configuration

Related API Commands

The RFdc driver API can be used to get the decimation rate set in the IP core using the following code.

```c
// Get Decimation factor for Tile0, DDC Block1
int Tile = 0;
u32 Block = 1;
u32 Decimation_Factor;
if( XRFdc_GetDecimationFactor (ptr, Tile, Block, &DecimationFactor) == XST_SUCCESS)  {
xil_printf("ADC Tile%1d,%1d Decimation Factor is: %d", Tile, Block,
Decimation_Factor);
}
```
Decimation Filter Details

The decimation filter chain consists of three FIR filters: FIR2, FIR1 and FIR0, which can be enabled to give successive decimation by a factor of two per stage. The filter transfer functions are as shown in the following figures.

**Figure 16:** FIR2 and FIR1 Frequency Response

**Figure 17:** FIR0 and 2x Decimation Frequency Response
Figure 18: 4x Decimation and 8x Decimation Frequency Response

The filter coefficients for the decimation filters are shown in the following tables.

<table>
<thead>
<tr>
<th>Filter</th>
<th>1st Half</th>
<th>Centre Tap</th>
<th>2nd Half</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. FIR2</td>
<td>5</td>
<td>0</td>
<td>-17</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-172</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>44</td>
<td>0</td>
<td>-96</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-4723</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>187</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-335</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-13331</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>41526</td>
<td>0</td>
</tr>
<tr>
<td>2. FIR2</td>
<td>65536</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0</td>
<td>-17</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-172</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>44</td>
<td>0</td>
<td>-96</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-4723</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>187</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-335</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-13331</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>41526</td>
<td>0</td>
</tr>
<tr>
<td>1. FIR1</td>
<td>-12</td>
<td>0</td>
<td>84</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-337</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1008</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-2693</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>10142</td>
<td>0</td>
</tr>
<tr>
<td>2. FIR1</td>
<td>16384</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-12</td>
<td>0</td>
<td>84</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-337</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1008</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-2693</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>10142</td>
<td>0</td>
</tr>
<tr>
<td>1. FIR0</td>
<td>-6</td>
<td>0</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-254</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1230</td>
<td>0</td>
</tr>
<tr>
<td>2. FIR0</td>
<td>2048</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-6</td>
<td>0</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-254</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1230</td>
<td>0</td>
</tr>
</tbody>
</table>

**RF-ADC Programmable Logic Data Interface**

The data interface between the RF-ADC tiles and the PL is implemented using parallel data streams, using the AXI4-Stream protocol. These data streams are output through the gearbox FIFOs which provide a flexible interface between the user application and the RF-ADC tile. The maximum interface width is 128 bits per stream, representing up to eight 16-bit big endian words. The data streams and associated FIFOs have a configurable number of words which provide the flexibility to choose between the number of words and clock frequency to interface with the PL design. There are four streams per tile, and the naming convention is mXY_axis, where X represents the RF-ADC tile number and Y represents a stream (FIFO) output from that tile. The following figure shows the interfaces.
Interface Data Formats

The data streams represent real or I/Q data, depending on the RF-ADC tile configuration. For 4 GSPS RF-ADC tiles, a given stream is either real, I or Q. If an RF-ADC is configured with I/Q output data, then the streams with an even number represent I data and the streams with an odd number represent Q data. These 4 GSPS real and I/Q configurations are shown in RF-ADC IP Configuration.

For 2 GSPS RF-ADC tiles, a given stream is either real or I/Q interleaved. If an RF-ADC is configured with I/Q output data, then the even-numbered samples of the stream represent I data and the odd-numbered samples represent Q data. These 2 GSPS Real and I/Q configurations are illustrated in the following sections.

Related Information
RF-ADC IP Configuration

Interface Data and Clock Rates

The total data rate per channel to the PL is determined by a number of factors, RF-ADC sample rate, decimation factor, and I/Q/Real data formats. The gearbox FIFOs provide a way of interfacing this data rate to the clock frequency of the PL design, by allowing the number of words per clock to be altered. The only requirements are that the number of words and clock rate combine to match the output data rate of the RF-ADC and the decimation rate, if enabled. All RF-ADCs in a tile share a common interface clock frequency. This is shown by the following equations, where $2G_{IQMode}$ is set to 2 for a 2 GSPS RF-ADC tile and I/Q mode is enabled and set to 1 otherwise.
The core automatically calculates the data rates based on the RF-ADC sample rate and datapath settings. This is shown in the following figure. See the RF-ADC Converter Configuration for information on the settings.

*Figure 20: RF-ADC Interface Data and Clock Rates Configuration*

Because each tile has independent clocking, sample rates, clock rates, PL rates, and configurations can be specified on a per-tile basis.

**Related Information**
RF-ADC Converter Configuration

**PL Clock Interface**

The AXI4-Stream data for all four tile streams is synchronous to a clock from the PL, which has a naming convention of mX_axis_aclk, where X represents the RF-ADC tile number. This clock must be at the frequency specified by the Required AXI4-Stream clock displayed on the IP core configuration screen.

The RF-ADC tile also outputs a clock that can be used by the PL. This output clock is a divided version of the RF-ADC sample clock, and is therefore frequency locked to it. This clock has a naming convention of clk_adcX, where X represents the RF-ADC tile number.
Interface FIFO Overflow

The data rate through the interface gearbox FIFOs must be constant during runtime of the RF-ADC tile, with no frequency drift between the PL clock and RF-ADC sample clock domains. If there is a frequency mismatch between these domains, a FIFO overflow might occur. The interface FIFOs have a built-in feature to determine if FIFO overflow has occurred, which is flagged to the PL using the IP interrupt mechanism.

There are two types of overflow, actual and marginal. Actual overflow indicates that the FIFO read/write pointers are overlapping, which means data is not being transferred safely between domains, and action must be taken. Marginal overflow is a warning and indicates that the FIFO read/write pointers are close to overlapping. Overflow should not occur during normal operation, and if overflow is observed it is an indication that the clocking infrastructure of the PL/PCB/IP is incorrectly configured.

Related Information
Interrupt Handling

Synchronization

The gearbox FIFOs provide a flexible data and clock interface for the RF-ADC tiles. However, as with all dual clock FIFOs, latency can vary between one tile and another. While all channels within a tile have the same latency, some applications might require more than one RF-ADC tile to be used, and require the latencies to be matched across all RF-ADC channels. These applications can use the Multi-Converter Synchronization feature to achieve this inter-tile synchronization.

Related Information
Multi-Converter Synchronization

RF-ADC Nyquist Zone Operation

Each RF-ADC channel can sample signals in the first or second Nyquist zones. To ensure the RF-ADC performance is optimal the RF-ADC configuration settings should indicate the intended zone of operation.

- First Nyquist zone is defined as a signal between 0 and $F_s/2$
- Second Nyquist zone is defined as a signal between $F_s/2$ and $F_s$

Other Nyquist zones can also be used as long as the signal meets the RF-ADC input bandwidth requirements. Zones 1, 3, 5,... are referred to as odd zones, and zones 2, 4,... are referred to as even zones.
Related Information
RF-ADC Converter Configuration

RF-ADC IP Configuration

The RF-ADC tile can be configured in several modes. The basic configuration options are available on the IP core configuration screen in the Vivado® IDE and advanced operating modes can be configured using the RFdc driver API.

The RF-ADC is available in fixed 4x2 GSPS or 2x4 GSPS configurations depending on the device (see the Zynq UltraScale+ RFSoC Data Sheet: Overview (DS889) and the Zynq UltraScale+ RFSoC Data Sheet: DC and AC Switching Characteristics (DS926)).

4 GSPS RF-ADC Configuration Options

4 GSPS RF-ADC Real Input to Real Output

Figure 21: 4 GSPS RF-ADC Real Input to Real Output

- m03_axis (Real)
- m02_axis (Real)
- m01_axis (Real)
- m00_axis (Real)
- Decimation
- Mixer/NCO (Bypass)
- 4 GSPS ADC
- Vin0 (Real)
- Vin1 (Real)
The following figure shows a 4 GSPS RF-ADC with real data input to real data output, 1x decimation, the mixer bypassed and running at a 500 MHz AXI4-Stream clock.
4 GSPS RF-ADC Real Input to I/Q Output

Figure 24: 4 GSPS RF-ADC Real Input to I/Q Output

- m00_axis (I)
- m01_axis (Q)
- m02_axis (I)
- m03_axis (Q)

Decimation

Mixer/NCO

4 GSPS ADC

Vin0 (Real)

Vin1 (Real)
The following figure shows a 4 GSPS RF-ADC with real data input to I/Q data output, 1x decimation, the mixer enabled, and running at a 500 MHz AXI4-Stream clock.
4 GSPS RF-ADC I/Q Input to I/Q Output

For I/Q input to I/Q output, the RF-ADCs are paired.
Figure 28: 4 GSPS RF-ADC I/Q Input to I/Q Output IP Core Configuration

The following figure shows a 4 GSPS RF-ADC with I/Q input to I/Q output, 1x decimation, the mixer enabled, and running at a 500 MHz AXI4-Stream clock.

The following figure shows a 4 GSPS RF-ADC with I/Q input to I/Q output, 1x decimation, the mixer enabled, and running at a 500 MHz AXI4-Stream clock.
2 GSPS RF-ADC Configuration Options

This option is only available on selected devices (see the Zynq UltraScale+ RFSoC Data Sheet: Overview (DS889) for device information).

2 GSPS RF-ADC Real Input to Real Output
The following figure shows a 2 GSPS RF-ADC real data input to real data output, 1x decimation, the mixer bypassed, and running at a 500 MHz AXI4-Stream clock.
Figure 32: 2 GSPS RF-ADC Real Input to Real Output Timing

Figure 33: 2 GSPS RF-ADC Real Input to I/Q Output
Figure 34: 2 GSPS RF-ADC Real Input to I/Q Output IP Core Configuration

The following figure shows a 2 GSPS RF-ADC real input data to I/Q output data, 1x decimation, the mixer enabled, and running at a 500 MHz AXI4-Stream clock.
2 GSPS RF-ADC I/Q Input to I/Q Output

For I/Q input to I/Q output, the RF-ADCs are paired.
The following figure shows a 2 GSPS RF-ADC I/Q to I/Q, 1x decimation, the mixer enabled, and running at a 500 MHz AXI4-Stream clock.

**Figure 38: 2 GSPS RF-ADC I/Q Input to I/Q Output Timing**
RF-DAC

Each RF-DAC tile includes a complete clocking support structure with a PLL and the necessary synchronization logic. Every RF-DAC in a tile has a highly configurable FIFO allowing the internal interconnect logic to have direct high-speed access to the RF-DAC (see figure below).

![Simplified RF-DAC Functionality Block Diagram](image)

Certain functions can only be executed when the RF-DACs in a tile are paired. Even numbered RF-DACs are used for I datapaths and the odd numbered RF-DACs are used for Q datapaths.

All of the available built-in functionality of a tile and each of the RF-DACs in a tile are user programmable. The Zynq® UltraScale+™ RFSoC RF Data Converter core configuration screen in the Vivado® IDE and the RFdc driver API can be used to configure the digital and analog functionality of the RF-DACs.

Related Information
Quadrature Modulator Correction

RF-DAC Analog Outputs

Each RF-DAC in a tile has its own differential analog current output buffer/driver. The output currents are complementary; the sum of the two currents always equals the full-scale current of the RF-DAC. The digital input code determines the effective differential current delivered to the load. The differential RF-DAC outputs are typically AC coupled out using capacitors. RF-DAC output DC coupling is also supported, assuming that the correct common mode biasing and load impedance requirements are met. See UltraScale Architecture PCB Design User Guide (UG583) for more details.
**Transmit Transfer Function**

The differential output provides the maximum output current when all digital input bits are High. The output current (using binary format) is shown in the following equations.

\[ I_{VOUTP} = \frac{BINDATAIN}{2^{N-1}} \times I_{OUTFS} \]

\[ I_{VOUTN} = I_{OUTFS} - I_{VOUTP} \]

Where:
1. \( BInDataIn \) = 14-bit digital input
2. \( I_{OUTFS} \) = Full-scale output current

The output current (using a twos complement) is shown in the following equations.

\[ I_{VOUTP} = \frac{TwosDataIN + 2^{N-1}}{2^{N-1}} \times I_{OUTFS} \]

\[ I_{VOUTN} = I_{OUTFS} - I_{VOUTP} \]

Where:
1. \( TwosDataIN \) = 14-bit digital twos complement input ranging from \((-2^{N-1})\) to \((2^{N-1}-1)\)
2. \( I_{OUTFS} \) = Full-scale output current.

**RF-DAC Output Current Mode**

The RF-DAC output current is configurable, with an option of operating in either 20 mA or 32 mA modes. In the 20 mA mode, set the DAC_AVTT to 2.5V and in the 32 mA mode, set it to 3.0V to maintain the linearity performance.

**CAUTION:** A 3.0V DAC_AVTT should not be used in the 20 mA mode. This risks exceeding the maximum ratings of the device and also risks affecting device reliability.

The output current can be set in the Advanced tab in the IP core configuration screen (see the following figure) or using the RFdc driver API.
Figure 40: RF-DAC Output Current Configuration

RFdc Driver API Commands

```c
// Set output current of RF-DAC Tile1, Block2 to 32mA
XRFdc_GetOutputCurr(ptr, 1, 2, XRFDC_OUTPUT_CURRENT_32MA);
```

Related Information

XRFdc_GetOutputCurr

RF-DAC Nyquist Zone Operation

Each RF-DAC can optimize its output response in the second Nyquist zone by using the mix-mode feature. This feature mixes the RF-DAC data with the sample clock, and, as a result, increases the output power in the second Nyquist zone, while attenuating it in the first Nyquist zone. This is shown in the following figure.
For normal (non-mix mode) operation, the blue line represents the ideal RF-DAC output roll-off sinc response. As can be seen, an output image in the second Nyquist zone in this mode would be severely attenuated. An inverse sinc filter is available to compensate for the roll-off in the first Nyquist zone for this mode. See RF-DAC Inverse Sinc Filter for more information.

In RF-DAC mix-mode, the red line represents the ideal RF-DAC output response. In this mode, the output power of the image in the second Nyquist zone is significantly increased, and it also has an approximately flat response across the majority of the zone. The Nyquist zone can be set in the Vivado® IDE.

Related Information
RF-DAC Inverse Sinc Filter

**RF-DAC Inverse Sinc Filter**

The analog output response of the RF-DAC follows a characteristic $\text{sinc}(x)/x$, or sinc shape. For applications that require flat-output response over a wide bandwidth, an inverse sinc filter is available to achieve this. The inverse sinc filter is an 11-tap FIR, which provides a flat-response with less than ±0.05 dB of ripple up to 89% of Nyquist, or ±0.033 dB of ripple up to 80% of Nyquist. The following figure shows the inverse sinc performance.
The line in blue represents the frequency response of the filter itself. As can be seen, it increases with frequency to compensate for the sinc response of the output, as shown by the red line. The composite output response is given by the yellow trace, and shows a flat pass-band up to 89% of Nyquist. The following figure shows the detail of the corrected response, with ripple specification levels highlighted in red.
**Inverse Sinc Filter Use**

The filter can be enabled or disabled per RF-DAC in the Vivado® IDE by using the Inverse Sinc Filter checkbox, as shown in the following figure.

**Figure 44: Inverse Sinc Configuration**

![Inverse Sinc Configuration](image)

**RFdc Driver API Commands**

The RFdc driver API can be used to get the inverse sinc enable state using the following code.

```c
// Get Inverse Sinc State for Tile0, DAC Block1
int Tile = 0;
u32 Block = 1;
u32 isinc_enabled;
isinc_enabled = XRFdc_GetInverseSincFilter (ptr, Tile, Block);
xil_printf("DAC%d,%d Inverse Sinc Enabled: %d\n\r", Tile, Block, isinc_enabled);
```
Related Information
XRFdc_GetInverseSincFilter

Overflow

As seen in the Inverse Sinc Frequency response, the filter gain is >1, and increases with frequency up to approximately 2.4 dB at 80% of Nyquist. To ensure that the filter output does not overflow, the input signal amplitude must be backed-off to account for this gain factor. The inverse sinc block has automatic overflow detection and saturation, and if overflow is detected, it is flagged using the interrupt mechanism to the RFdc driver API, using the XRFDC_DAC_IXR_INVNSC_OF_MASK interrupt. For more details see Interrupt Handling.

Related Information
Interrupt Handling

Inverse Sinc Filter Details

| Inverse Sinc FIR | -1 | 3 | -6 | 15 | -52 | 594 | -52 | 15 | -6 | 3 | -1 |

RF-DAC Digital Datapath

Each RF-DAC in a tile has a number of optimized DSP features that can be used to implement Digital Up Conversion (DUC) and transmit signal filtering. These features are:

- Signal interpolation function - interpolation by x1 (bypass), ×2, ×4, or ×8 is supported
- Coarse mixing (quarter and half rate) or fine mixing with a 48-bit frequency resolution numerically controlled oscillator (NCO)
- Compensation functionality containing a quadrature modulator correction (QMC) block with coarse delay adjustment block
- Signal conditioning containing an inverse sinc FIR filter

Single, multiple, or all DSP functions can be used or bypassed, using the Vivado® IDE.

The following figure shows the available functions in a RF-DAC. Each function is described in the following sections.
RF-DAC Interpolation Filters

Interpolation filters are required to implement the up-sampling and filtering portions of the DUC process. The implemented filter operation consists of three low pass FIR filters, each with a predetermined, fixed set of coefficients, shown in the following figure. Each filter block can be bypassed and the output of each filter block can be routed to the output of the filter. This allows the creation of the following:

- 1x—all filter stages are bypassed
- 2x—interpolation using a single stage
- 4x—interpolation using two stages
- 8x—interpolation using all three available stages
The output of an interpolated signal is the exact representation of the original signal presented at a higher sampling rate. Up-sampling of a signal sampled at F1 to a higher sample rate (N*F1) results in N replications of the original spectrum. Low pass filtering of the result helps to remove unwanted replications resulting in the preservation of the original signal at the required higher sample rate.

Table 41: **Interpolation Filter Operating Modes**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFF</td>
<td>Filter is disabled, RF-DAC is not available.</td>
</tr>
<tr>
<td>1x</td>
<td>Filter is bypassed.</td>
</tr>
<tr>
<td>2x</td>
<td>2x interpolation, 80% Nyquist passband.</td>
</tr>
<tr>
<td>4x</td>
<td>4x interpolation, 80% Nyquist passband.</td>
</tr>
<tr>
<td>8x</td>
<td>8x interpolation, 80% Nyquist passband.</td>
</tr>
</tbody>
</table>

Each of the filter stages can overflow—given the step-response of a FIR filter, especially when full-scale data is on the input. Each filter stage has an overflow status signal and output saturation to detect an overflow and protect the datapath. When a filter stage is not used, the overflow flag is pulled Low. These flags are connected to the datapath interrupts mechanism.

**Related Information**

**Interrupt Handling**

**Interpolation Filter Use**

The IP core is used to set the interpolation rate. This is an IP core setting because changing the interpolation rate directly affects the physical interface as the bandwidth to the PL changes. The filters that are enabled are shown in the following figure.
**Figure 47: RF-DAC Interpolation Filter Configuration**

**Note:** I/Q input data and mixing requires a minimum of 2x interpolation. If full bandwidth I/Q data is required, two RF-DAC data converters can be operated as independent real streams, containing the I and Q data respectively.

**Related Information**

**RF-DAC Converter Configuration**
RFdc Driver API Commands

The RFdc driver API can be used to get the interpolation rate set in the Vivado® IDE using the following code.

```c
// Get Interpolation factor for Tile0, DDC Block1
int Tile = 0;
uint32 Block = 1;
uint32 Interpolation_Factor;
if( XRFdc_GetInterpolationFactor (ptr, Tile, Block, &Interpolation_Factor) == XST_SUCCESS) {
    xil_printf("DAC Tile%1d,%1d Interpolation Factor is: %d", Tile, Block, Interpolation_Factor);
}
```

Related Information
XRFdc_GetInterpolationFactor

Interpolation Filter Details

The interpolation filter chain consists of three FIR filters, FIR2, FIR1 and FIR0, which can be enabled to give successive interpolation by a factor of two per stage. The filter transfer functions are shown in the following figures.

*Figure 48: FIR2 and FIR1 Frequency Response*
The filter coefficients for the interpolation filters are shown in the following tables.

<table>
<thead>
<tr>
<th>FIR 2</th>
<th>1st Half</th>
<th>Centre Tap</th>
<th>2nd Half</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>0</td>
<td>-17</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1401</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-17</td>
<td>0</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-2112</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>44</td>
<td>3145</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-96</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-96</td>
<td>0</td>
<td>187</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-4723</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-4723</td>
<td>0</td>
<td>7415</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-13331</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-13331</td>
<td>0</td>
<td>41526</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>65536</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>41526</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-906</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-906</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FIR1</th>
<th>1st Half</th>
<th>Centre Tap</th>
<th>2nd Half</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-12</td>
<td>16384</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td></td>
<td>84</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-337</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-337</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1008</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1008</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-2693</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-2693</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>10142</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10142</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
The interpolation filter chains can operate in real or I/Q modes, depending on the data type selected for the RF-DAC interface.

**Related Information**
RF-DAC Programmable Logic Data Interface

**RF-DAC Numerical Controlled Oscillator and Mixer**

The mixer function has three modes, bypass (no mixing), coarse mixing or fine mixing. Fine mixing automatically enables the NCO which is used to generate the carrier frequency. The mixer supports full quadrature mixing, with both real to I/Q and I/Q to I/Q modes supported.

*Figure 51: RF-DAC Mixer with NCO DSP Block*
Coarse Mixer:

- The coarse mixer allows the data to be mixed with a carrier of 0, $F_s/2$, $F_s/4$, or $-F_s/4$.
- Mixing with a 0 carrier bypasses the mixer component.

Fine Mixer:

- The fine mixer allows the data to be shifted up or down in frequency by an arbitrary amount.
- The frequency shift amount is obtained by programming the mixer frequency generated in the NCO. The fine mixer also supports 18-bit phase adjustment.
- The NCO phase can be synchronized within a tile using `XRFdc_UpdateEvent`.
- The NCO phase can be synchronized across tiles using an external event signal (SYSREF or MARKER).
- To manage potential overflow, the fine mixer output includes 3 dBV attenuation, as shown in the figure above. This attenuation is set to -3 dBV (multiplication factor = 0.707) regardless of the RF-DAC setting in the RFdc driver API. A manual selection is also possible, allowing 0 dBV or -3 dBV.

The mixer settings can be configured in the core, or by using the RFdc driver API. The core is used to set the initial mixer settings (for example, mixer type and mixer mode), and the RFdc driver API is used to adjust the settings at runtime. Both the RFdc driver API and the core compute the required register settings based on the supplied sample rates and desired frequencies. See the RF-DAC Converter Configuration section for information on the settings.

Related Information
RF-DAC Converter Configuration

**RF-DAC Multi-band Operation**

Multi-band operation is where two or more baseband signals are up-converted (mixed) to individual carriers and then combined to generate a single composite analog output. In the RF-DAC tiles, this involves combining multiple DUC blocks together to drive the analog outputs.

The RF-DAC multi-band feature supports the following configurations:

- 2x multi-band real data per pair. One RF-DAC output of the pair is enabled; the other is off.
- 2x multi-band I/Q data per pair. Both RF-DACs in the pair are enabled, one for I and one for Q.
- 4x multi-band real data per tile. Four input streams are combined to drive the DAC0 output in real mode. All other RF-DACs are off.
- 4x multi-band I/Q data per tile. Four input streams are combined to drive the DAC0 output as I and DAC1 output as Q. All other RF-DACs are off.
Latency between outputs in a multi-band pair is matched and latency between pairs is matched, irrespective of the mode.

When multi-band is off, the I and Q datapaths pass straight through the multi-band logic block (as shown below). When multi-band is on, I and Q of each datapath are combined and passed to the next DSP block in the chain in front of the RF-DAC. Multi-band can be turned on per RF-DAC, for only the I datapath or the Q datapath, or for both I and Q.

**Figure 52: Multi-band Logic**

RF-DAC multi-band is implemented by connecting multiple RF-DAC DUC blocks to an RF-DAC analog block. Each DUC block handles one band of data, and can mix this up to any carrier frequency. The output is then summed before being sent to the analog datapath and RF-DAC output. This is shown in the following figure.
Two example multi-band configurations are shown. The RF-DAC tile consists of four DUC blocks (digital datapaths) and four RF-DAC-analog blocks (analog datapaths). In the left hand example, the lower RF-DAC pair in the tile is configured as 2x real multi-band, while the top pair are independent RF-DACs. Because the bottom pair uses two DUC blocks, DAC1 is off, and DAC0 outputs the dual-band signal. In the right hand example, the lower pair is configured as 2x I/Q multi-band, while the top pair is off.

**RF-DAC Programmable Logic Data Interface**

The data interface between the RF-DAC tiles and the PL is implemented using parallel data streams, using the AXI4-Stream protocol. These data streams are input to gearbox FIFOs that provide a flexible interface between the user application and the RF-DAC tile. The maximum interface width is 256 bits per stream, representing up to 16 16-bit big endian words. The data streams and associated FIFOs have a configurable number of words which provide the flexibility to choose between the number of words and the clock frequency to interface with the PL design. There are four streams per tile, and the naming convention is \( s_{XY\_axis} \), where \( X \) represents the RF-DAC tile number, and \( Y \) represents a stream input to the FIFO in that tile. The following figure shows the interfaces.
**Interface Data Formats**

The data streams represent real or I/Q data, depending on the RF-DAC tile configuration. For RF-DAC tiles, a given stream is either real or I/Q interleaved. If an RF-DAC is configured with I/Q input data then the even numbered samples of the stream represent I data and the odd-numbered samples represent Q data. These real and I/Q configurations are shown in RF-DAC IP Configuration.

In each configuration all the enabled RF-DAC FIFOs in a tile begin to accept data when the power-up sequence has completed. This is indicated by the assertion of the $s_{XY\_axis\_tready}$ outputs. If the data for any stream is not valid at this point, logic to suppress it should be included in the PL.

**Related Information**

RF-DAC IP Configuration

**Interface Data and Clock Rates**

The total data rate per channel to the PL is determined by a number of factors, RF-DAC sample rate, interpolation factor, and I/Q and Real data formats. The gearbox FIFOs provide a way of interfacing this data rate to the clock frequency of the PL design, by allowing the number of words per clock to be chosen. The only requirements are that the interface number of words and clock rate combine to match the data rate required by the RF-DAC channel, and all RF-DACs in a tile share a common interface clock frequency. This is shown by the following equations:

$$PL\_Data\_Rate = \frac{DAC\_Data\_Rate \times IQ\_Mode}{Interpolation\_Rate}$$
The IP core automatically calculates the data rates based on the RF-DAC sample rate and datapath settings. This is shown in the following figure.

**Figure 55: RF-DAC Data Interface Data and Clock Configuration**

Each tile has independent clocking; sample rates, clock rates, PL rates, and configurations can be specified on a per-tile basis.

**Related Information**

RF-DAC Converter Configuration

**PL Clock Interface**

The AXI4-Stream data for all four tile streams is synchronous to a clock from the PL, which has a naming convention of `sX_axis_aclk`, where X represents the RF-DAC tile number. This clock must be at the frequency specified by the Required AXI4-Stream clock displayed on the IP core configuration screen.

The RF-DAC tile also outputs a clock that can be used by the PL. This output clock is a divided version of the RF-DAC sample clock and therefore is frequency locked to it. This clock has a naming convention of `clk_dacX`, where X represents the RF-DAC tile number.

**Related Information**

Clocking
**Interface FIFO Overflow**

The data rate through the interface gearbox FIFOs must be constant during runtime of the RF-DAC tile, with no frequency drift between the PL clock and RF-DAC analog sample clock domains. If there is a frequency mismatch between these domains, a FIFO overflow can occur. The interface FIFOs have a built-in feature to determine if FIFO overflow has occurred, which is flagged to the PL through the IP interrupt mechanism.

There are two types of overflow: actual and marginal. Actual overflow indicates that the FIFO read/write pointers are overlapping, which means data is not being transferred safely between domains, and action must be taken. Marginal overflow is a warning and indicates that the FIFO read/write pointers are close to overlapping. Overflow should not occur during normal operation, and if overflow is seen it is an indication that the clocking infrastructure of the PL/PCB/IP core is incorrectly configured.

Related Information
Interrupt Handling

**Synchronization**

The gearbox FIFOs provide a flexible data and clock interface for the RF-DAC tiles. However, as with all dual-clock FIFOs, latency might vary between one tile and another. While all channels within a tile have the same latency, some applications might require more than one RF-DAC tile to be used, and require the latencies to be matched across all RF-DAC channels. These applications can use the Multi-Converter Synchronization feature to achieve this inter-tile synchronization.

Related Information
Multi-Converter Synchronization

**RF-DAC IP Configuration**

The RF-DAC can be configured in several modes. The basic configuration options are available on the IP core configuration screen in the Vivado® IDE and advanced operating modes can be configured using the RFdc driver API.
RF-DAC Real Input to Real Output

Figure 56: RF-DAC Real Input to Real Output

- s03_axis (Real) → Interpolation → Mixer/NCO (Bypass) → 6.5 GSPS DAC → Vout3 (Real)
- s02_axis (Real) → Interpolation → Mixer/NCO (Bypass) → 6.5 GSPS DAC → Vout2 (Real)
- s01_axis (Real) → Interpolation → Mixer/NCO (Bypass) → 6.5 GSPS DAC → Vout1 (Real)
- s00_axis (Real) → Interpolation → Mixer/NCO (Bypass) → 6.5 GSPS DAC → Vout0 (Real)
Figure 57: RF-DAC Real Input to Real Output IP Configuration

The following figure shows a 6.5 GSPS RF-DAC with real input to real output, 1x interpolation, the mixer bypassed, and running at a 400 MHz AXI4-Stream clock.
**Figure 58: RF-DAC Real Input to Real Output Data Timing**

**RF-DAC I/Q Input to Real Output**

**Figure 59: RF-DAC I/Q Input to Real Output**

- **s03_axis (I & Q)** → I/Q Interpolation → Mixer/NCO → 6.5 GSPS DAC → Vout3 (Real)
- **s02_axis (I & Q)** → I/Q Interpolation → Mixer/NCO → 6.5 GSPS DAC → Vout2 (Real)
- **s01_axis (I & Q)** → I/Q Interpolation → Mixer/NCO → 6.5 GSPS DAC → Vout1 (Real)
- **s00_axis (I & Q)** → I/Q Interpolation → Mixer/NCO → 6.5 GSPS DAC → Vout0 (Real)
The following figure shows a 6.5 GSPS RF-DAC with I/Q input to real output, 2x interpolation, the mixer bypassed, and running at a 400 MHz AXI4-Stream clock.
**Figure 61: RF-DAC I/Q Input to Real Output Timing**

*Note:* Interpolation is x2 because the bandwidth available on the AXI4-Stream interface is limited.

**RF-DAC I/Q Input to I/Q Output**

For I/Q input to I/Q output, the RF-DACs are paired. An I/Q output signal requires a pair of RF-DACs, one for I data and one for Q data. As shown in the following figure, a single DUC block outputs I and Q data and these signals are sent to the two RF-DACs.

*Figure 62: RF-DAC I/Q Input to I/Q Output*
The following figure shows a 6.5 GSPS RF-DAC with I/Q input data to I/Q output data with 2x interpolation, the mixer bypassed, and with a 400 MHz AXI4-Stream clock.
Figure 64: RF-DAC I/Q Input to I/Q Output Data Timing

Note: Interpolation is x2 because the bandwidth available on the AXI4-Stream interface is limited.
Quadrature Modulator Correction

When using an external analog quadrature mixer device, a pair of converters (either RF-ADC or RF-DAC) must be used to handle the I and Q datapaths after conversion. Due to external events or circumstances, errors or imbalances can be introduced in the analog I and Q signal paths which, if not corrected, can lead to system performance degradation. The necessary corrections to restore any system from degrading are accomplished using the quadrature modulator correction (QMC) circuit.

As shown in the following figure, the QMC circuit is for correction only. Error and/or imbalance detection must be done by application-specific code in the internal interconnect logic.

The QMC circuit can compensate for the following:

- Gain correction is done by multiplying the signal by a gain factor. This factor has a range of 0 to 2.0, and individual factors can be applied to the I and Q datapaths. The output resolution of the block is 16 bits.
• Phase correction is achieved by adding a scaled fraction of Q to the I value. The result of this addition can result in a gain error that must be corrected by the gain error correction block. Phase correction has a range of approximately ±26 degrees.

• Offset correction is done by adding a fixed LSB value to the sampled signal.

For full QMC functionality, a pair of converters need to be set up and used in I/Q mode. When disabling the phase correction factor, the QMC block can supply gain and offset correction to a converter used in real mode. In this case, the converters do not need to be paired to make use of the QMC block functions.

## Update QMC Settings

The gain, phase, and offset correction factor values can be set using the RFdc driver API. An example of using the API is as follows.

```c
// Initial Setup
XRFdc_QMC_Settings QMC_Settings_I, QMC_Settings_Q; // RF-ADC block0 is I, RF-ADC block1 is Q
QMC_Settings_I.EventSource = XRFDC_EVNT_SRC_TILE; // QMC Settings are updated with a tile event
QMC_Settings_Q.EventSource = XRFDC_EVNT_SRC_TILE;

// Update Gain/Phase/Offset for I/Q RF-DACs in tile0
QMC_Settings_I.GainCorrectionFactor = 0.9; // Set Gain for I
QMC_Settings_I.PhaseCorrectionFactor = -5.0; // I/Q imbalance factor applied to I side, approx in degrees.
QMC_Settings_I.EnableGain = 1;
QMC_Settings_I.EnablePhase = 1;
QMC_Settings_Q.GainCorrectionFactor = 0.95;
QMC_Settings_Q.EnableGain = 1;
XRFdc_SetQMCSettings(ptr, XRFDC_ADC_TILE, 0, 0, &QMC_Settings_I); // Write settings for ADC0,0 - I ADC
XRFdc_SetQMCSettings(ptr, XRFDC_ADC_TILE, 0, 1, &QMC_Settings_Q); // Write settings for ADC0,1 - Q ADC
XRFdc_UpdateEvent(ptr, XRFDC_ADC_TILE, 0, 0, XRFDC_EVENT_QMC); // Generate a Tile Update Event - applies all QMC Settings at once
```

It is also possible to read back the QMC settings from any converter, using the XRFdc_GetQMCSettings RFdc driver API command. This populates the QMC_Settings structure with the values from the hardware.
Related Information
XRFdc_SetQMCSettings
XRFdc_UpdateEvent

I/Q RF-ADC Naming Convention

When converters are used in an I/Q pair, the I channel always uses the even block numbers, and the Q channel always uses the odd block numbers. For example, in a 4 GSPS RF-ADC tile, the I RF-ADC is block0, and the Q RF-ADC is block1. For 2 GSPS RF-ADC tiles, or RF-DAC tiles, the I channels are block0 and block2, while the related Q channels are block1 and block3, respectively.

QMC Overflow

Setting excessive gain, phase or offset correction factors relative to the input signal can cause the datapath to overflow. To aid system debug, the QMC block contains built-in overflow detection and saturation blocks. These blocks generate flags which are connected to the IP interrupt mechanism using the datapath interrupt. The related interrupt flags for the QMC block are:

XRFDC_IXR_QMC_GAIN_PHASE_MASK
XRFDC_IXR_QMC_OFFST_MASK

These interrupts can be enabled per converter channel.

- The gain correction multiplier range is between 0 and 2.0.
- The range of the phase correction multiplier is approximately ±26 degrees, or ±0.5 in terms of magnitude.

Related Information
Interrupt Handling
Dynamic Update Events

Certain datapath features in the RF-ADC and RF-DAC tiles can be updated during runtime. Depending on the application, it might be required to update a setting immediately or to wait for a certain event to apply the update. For example, when resetting the NCO phase, using the SYSREF event allows the phase reset action to be applied at the same time across multiple tiles. The following table shows the available events types.

Table 45: Event Types

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Function</th>
<th>Event Source</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>Updates the settings of a given feature immediately after writing.</td>
<td>RFdc Driver</td>
<td>Does not apply for 4 GSPS RF-ADC tiles</td>
</tr>
<tr>
<td>Block</td>
<td>Updates when the block event register is written. Allows features of a given block to update at the same time.</td>
<td>RFdc Driver</td>
<td>Does not apply for 4 GSPS RF-ADC tiles</td>
</tr>
<tr>
<td>Tile</td>
<td>Updates when the tile event register is written. Allows features in all blocks in a tile to update at the same time.</td>
<td>RFdc Driver</td>
<td></td>
</tr>
<tr>
<td>PL</td>
<td>Updates when the PL_event input on a tile is asserted. Allows features across multiple blocks/tiles to update at the same time.</td>
<td>dacXY_pl_event</td>
<td>Asserted from user design in the PL</td>
</tr>
<tr>
<td>SYSREF</td>
<td>Updates when a SYSREF is received through the SYSREF_p/n input pins. Allows features across multiple blocks/tiles/devices to update at the same time.</td>
<td>SYSREF_p/n input</td>
<td>Asserted from the SYSREF generator on the PCB</td>
</tr>
<tr>
<td>PL Marker</td>
<td>Updates when the user_sysref_dac signal is asserted. This signal is synchronous and aligned to the RF-DAC data.</td>
<td>user_sysref_dac IP input</td>
<td>Applies only to RF-DAC tiles</td>
</tr>
</tbody>
</table>

Notes:
1. These signals are synchronous to the converter output clocks, clk_dacX and clk_adcX. These clocks are on local routing. A small amount of logic can be clocked on the output clocks and used to drive the PL event signals. This logic should be placed near the converter to facilitate timing closure. Alternatively the PL event signals can be generated on a global clock and then synchronized to the output clock of the relevant converter.

Features that are intended to be dynamically updated must be programmed to be sensitive to one of these update events. The event source for each feature is flexible and can be individually set. The following features support this dynamic update:

- Fine Mixer / NCO
- Quadrature Modulator Correction (QMC)
- Coarse Delay
Update Event Use

The RFdc driver API uses the EventSource parameter of a particular function to set up the trigger event for that function. It can also issue a subset of the events. The following code shows an example of the setup of a tile event source followed by the tile event.

```c
// Initial Setup - Mixer settings for 2 DDC blocks
XRFdc_Mixer_Settings Mixer_Settings_0;
XRFdc_Mixer_Settings Mixer_Settings_1;

// Mixer Settings for both DDC blocks are updated with a tile event
Mixer_Settings_0.EventSource = XRFDC_EVNT_SRC_TILE;
Mixer_Settings_1.EventSource = XRFDC_EVNT_SRC_TILE;

// Make changes to mixer settings and write them to both DDCs
Mixer_Settings_0.freq = 1.23;
Mixer_Settings_1.freq = 1.43;
XRFdc_SetMixerSettings (ptr, XRFDC_ADC_TILE, 0, 0, &Mixer_Settings_0);
XRFdc_SetMixerSettings (ptr, XRFDC_ADC_TILE, 0, 1, &Mixer_Settings_1);

// Generate the event for DDC block 0
// This maps to the Tile event, which updates both DDCs at the same moment
XRFdc_UpdateEvent(ptr, XRFDC_ADC_TILE, 0, 0, XRFDC_EVENT_MIXER);
```

Related Information

- XRFdc_SetMixerSettings
- XRFdc_UpdateEvent

PLL

Each RF-ADC and RF-DAC tile includes a clocking system with an input clock divider, a PLL, and an output divider. When used with the SYSREF input, the clocking system can be synchronized in multi-tile or multi-chip designs. When the PLL is used, the output sampling clock is fed to the multiplexer, which can also be used by a direct input clock before it is routed into the clock network of the tile (as shown below). The clock from the PLL is used in both the analog and digital portions of the RF-ADC or RF-DAC in a tile. Each RF-ADC or RF-DAC in a tile has a local divider to divide and distribute the very low jitter clock from the PLL or a direct input to the different functions in each RF-ADC or RF-DAC function.
Figure 66: Tile PLL Clocking Structure

The operating parameters of the PLL are set using the Vivado® IDE to specify the default PLL configuration or by the RFdc driver API if runtime adjustment of the PLL is required. The PLL must first be enabled in the Vivado IDE if runtime adjustment using the API is required.

The internal PLL block diagram is shown in the following figure. The API function dedicated to configuring the PLL system sets a reference divider (an integer from 1 to 4), feedback divider (an integer from 13 to 160) and output divider (an integer from 2 to 64) to achieve the best performance of the PLL using the VCO in the correct range. In any frequency configuration, the best performance of phase noise is achieved when the PLL system is able to select a reference divider of 1.
# Interrupt Handling

The RF-ADCs and RF-DACs and tiles can generate interrupts during operation which can help debug or avoid potential issues. The interrupts are shown in the following table.

## Table 46: Interrupts

<table>
<thead>
<tr>
<th>Block</th>
<th>Interrupt</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RF-ADC</td>
<td>Analog - OverVoltage</td>
<td>Indicates the analog input is exceeding the safe input range of the RF-ADC input buffer and the buffer has been shut down. The input signal amplitude and common mode should be brought within range. Related RFdc Driver API constraints: Enable All: XRFDC_ADC_IXR_OVRVOLT_MASK</td>
</tr>
<tr>
<td>RF-ADC/RF-DAC</td>
<td>Datapath QMC</td>
<td>Indicates the QMC gain/phase/offset correction has overflowed/saturated. Data amplitude, or correction factors are too high and should be reduced. Related RFdc Driver API constraints: Gain, Phase: XRFDC_IXR_QMC_GAIN_PHASE_MASK Offset: XRFDC_IXR_QMC_OFFST_MASK</td>
</tr>
<tr>
<td>RF-ADC</td>
<td>Datapath Decimation</td>
<td>Indicates one of the decimation stages has overflowed/saturated. Flags are per-stage and I/Q paths to indicate where the overflow has occurred. Data amplitude is too high. Related RFdc Driver API constraints: Enable All: XRFDC_ADC_IXR_DMON_STG_MASK I-Datapath, Stage 0, 1, 2: XRFDC_ADC_IXR_DMON_I_STG&lt;0,1,2&gt;_MASK Q-Datapath, Stage 0, 1, 2: XRFDC_ADC_IXR_DMON_Q_STG&lt;0,1,2&gt;_MASK</td>
</tr>
<tr>
<td>RF-ADC</td>
<td>Datapath Interpolation</td>
<td>Indicates one of the interpolation stages has overflowed/saturated. Flags are per-stage and I/Q paths to indicate where the overflow has occurred. Data amplitude is too high. Related RFdc Driver API constraints: Enable All: XRFDC_ADC_IXR_INTP_STG_MASK I-Datapath, Stage 0, 1, 2: XRFDC_ADC_IXR_INTP_I_STG&lt;0,1,2&gt;_MASK Q-Datapath, Stage 0, 1, 2: XRFDC_ADC_IXR_INTP_Q_STG&lt;0,1,2&gt;_MASK</td>
</tr>
<tr>
<td>RF-ADC</td>
<td>Analog - OverRange</td>
<td>Indicates the analog input is exceeding the full-scale range of the RF-ADC. The input signal amplitude should be reduced. Related RFdc Driver API constraints: Enable All: XRFDC_SUBADC_IXR_DCDR_MASK</td>
</tr>
<tr>
<td>RF-ADC/RF-DAC</td>
<td>FIFO Overflow</td>
<td>Indicates that the FIFO interface is incorrectly setup, clocks/data throughput mismatch. Related RFdc Driver API constraints: Enable All: XRFDC_IXR_FIFOUSRDAT_MASK Actual Overflow: XRFDC_IXR_FIFOUSRDAT_OF_MASK</td>
</tr>
<tr>
<td>RF-DAC</td>
<td>Datapath Inverse Sinc</td>
<td>Indicates the inverse sinc filter has overflowed/saturated. Data amplitude is too high and should be reduced. Related RFdc Driver API constraints: Inverse Sinc: XRFDC_DAC_IXR_INVSNC_OF_MASK</td>
</tr>
</tbody>
</table>

The following figure shows the interrupt hierarchy within the IP core.
Datapath overflow indicates that a sub-block in the signal chain has detected that the output signal amplitude has exceeded full scale, and has been saturated. If overflow is detected it indicates that either the signal amplitude is too high, or the block settings are incorrect for the signal amplitude. For example, using a large gain correction factor on a signal that is close to full-scale causes a QMC gain overflow.

All of the interrupts shown in the previous figure are enabled and handled by the interrupt functions of the RFdc driver API. The default interrupt status handler (XRFdc_IntrHandler) automatically follows the interrupt hierarchy in the previous figure to determine the source of the interrupt, and then passes this information to the user-defined status handler to respond.

**Related Information**
- RF-DAC Interpolation Filters
- Quadrature Modulator Correction
- RF-ADC Decimation Filters
- Clocking
Interrupt Setup and Handling Example

The following code shows an interrupt setup and handling example.

```c
// Handler
// User-Defined interrupt status handler
void RFdcHandler (void *CallBackRef, u32 Type, int Tile, u32 Block, u32 Event) {
    // Check the type of interrupt event
    if (Type == XRFDC_DAC_TILE) {
        xil_printf("Interrupt occurred for ADC%d,%d :", Tile_Id, Block_Id);
        if (Event & (XRFDC_IXR_FIFOUSRDAT_OF_MASK | XRFDC_IXR_FIFOUSRDAT_UF_MASK)) {
            xil_printf("FIFO Actual Overflow\n");
        }
        if (Event & (XRFDC_IXR_FIFOMRGNIND_OF_MASK | XRFDC_IXR_FIFOMRGNIND_UF_MASK)) {
            xil_printf("FIFO Marginal Overflow\n");
        }
        if (Event & XRFDC_DAC_IXR_INTP_STG_MASK) {
            xil_printf("Interpolation Stages Overflow\n");
        }
        // ... Other handling code ...
    } else {
        xil_printf("Interrupt occurred for ADC%d,%d :\n", Tile_Id, Block_Id);
        // ... ADC handling code ...
    }
}

// Setup
// Register the user-defined interrupt handler
XRFdc_SetStatusHandler(ptr, ptr, (XRFdc_StatusHandler) RFdcHandler);
// Setup for RF-DAC tile 0, block 0 - FIFO Interrupt
XRFdc_IntrEnable(ptr, XRFDC_DAC_TILE, 0, 0, XRFDC_IXR_FIFOUSRDAT_MASK);
// Test
// Force a FIFO interrupt by setting the FIFO write words to a different value from the design requirements.
// Note: Assumes a value of 1 is invalid for this design.
XRFdc_SetFabRdVldWords(ptr, 0, 0, 1);
```
This example shows the RFdc driver-specific interrupt handling and the forcing of an interrupt by changing the FIFO parameters. The setup and registration of the RF Data Converter irq output with the PS or MicroBlaze™ interrupt controller is not shown. For more details, see the examples provided with the RFdc driver API (see the Zynq UltraScale+ Device Technical Reference Manual (UG1085)).

Related Information
XRFdc_IntrEnable
XRFdc_SetFabRdVldWords

---

## Clocking

The following table describes the clocks associated with the Zynq® UltraScale+™ RFSoC RF Data Converter IP core.

**Table 47: IP Core Clocks**

<table>
<thead>
<tr>
<th>Clock</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s_axi_aclk</td>
<td>AXI4-Lite clock. This clock is common to all tiles in the core and should be driven by the system CPU used for the AXI4-Stream interconnect connected to the AXI4-Lite interface. This clock should be between 10 MHz and 200 MHz.</td>
</tr>
<tr>
<td>dac[3:0]_clk_p/n</td>
<td>Sampling Clock or Reference Clock to internal PLL for the RF-DAC tiles. There are up to four differential clock inputs on the core because there is a maximum of four RF-DAC tiles available. This clock must be driven with the frequency chosen on the IP core configuration screen.</td>
</tr>
<tr>
<td>adc[3:0]_clk_p/n</td>
<td>Sampling Clock or Reference Clock to internal PLL for the RF-ADC tiles. There are up to four differential clock inputs on the core because there is a maximum of four RF-ADC tiles available. This clock must be driven with the frequency chosen on the IP core configuration screen.</td>
</tr>
<tr>
<td>clk_dac[3:0]</td>
<td>Output clock to interconnect logic from the RF-DAC tiles. This is a user-configurable output clock from the core to the user logic. The use of this clock is optional. It can be used to drive the PL design if another frequency-locked RF-DAC sample clock is not available in the PL. The frequency of this clock output can be chosen from a list of values, related to the sample rate, in the IP core configuration screen.</td>
</tr>
<tr>
<td>clk_adc[3:0]</td>
<td>Output clock to interconnect logic from the RF-ADC tiles. This is a user-configurable output clock from the core to the user logic. It can be used to drive the PL design if another frequency-locked RF-ADC sample clock is not available in the PL. The frequency of this clock output can be chosen from a list of values, related to the sample rate, in the IP core configuration screen.</td>
</tr>
<tr>
<td>s[3:0]_axis_aclk</td>
<td>Slave AXI4-Stream clocks for RF-DAC data interfaces. There are up to 16 AXI4-Stream interfaces on the core, one per RF-DAC. Each RF-DAC tile has a single clock common to up to four AXI4-Stream data interfaces. All AXI4-Stream data interfaces on a tile must operate at the same clock rate but the bandwidth of each interface can be different because each AXI4-Stream interface can have a different width as specified in the IP core configuration screen. This clock must be frequency-locked to the RF-DAC sample clock as outlined in Interface FIFO Overflow.</td>
</tr>
</tbody>
</table>
Table 47: IP Core Clocks (cont’d)

<table>
<thead>
<tr>
<th>Clock</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m[3:0]_axis_aclk</td>
<td>Master AXI4-Stream clocks for RF-ADC data interfaces. There are up to eight or 16 AXI4-Stream interfaces on the core depending on the chosen device, one per RF-ADC. Each RF-ADC tile has a single clock common to up to two or four AXI4-Stream data interfaces. All AXI4-Stream data interfaces on a tile must operate at the same clock rate; however the bandwidth of each interface can be different because each AXI4-Stream interface can have a different width as specified in the IP core configuration screen. This clock must be frequency-locked to the RF-ADC sample clock as outlined in Interface FIFO Overflow.</td>
</tr>
</tbody>
</table>

Related Information

Interface FIFO Overflow

Clock Inputs

Each RF-ADC or RF-DAC tile has its own clock input. There is no need to instantiate a clock input buffer in your design because the current-mode logic (CML) clock input buffer is implemented in the tile architecture. The clock input buffer can be used as a sampling clock for the tile or as a reference for the tile PLL. The CML clock input has an on-die differential termination of 100 Ω.

The clock input buffer has a detection circuit for measuring activity at the external clock inputs. When a clock is present, the input buffer is activated. When no clock is present, the outputs of the clock buffer are forced into a steady state. If the clock is lost, the core shuts the tile down but will restart the tile without user intervention when the clock returns.

When the PLL is not used, the clock buffer output is passed through a multiplexer into the RF-ADC or RF-DAC tile clock network. When the PLL is used, the output of the clock buffer is driven to the PLL to serve as the reference clock.

In each Zynq® UltraScale+™ RFSoC, there is a dedicated input SYSREF pin pair per package. The SYSREF clocks the multi-tile and multi-chip synchronization. For multi-tile designs, the SYSREF connects into a master tile and the signal must be distributed from inside that tile to all other tiles used in the design.

The presence of a clock at the input pins can be checked by using the RFdc software API. If no clock is detected the RF-ADC and RF-DAC blocks affected do not start up and stay in the Clock Detection state.

Related Information

Power-up Sequence
Tile Clocking Structure

The following figures show the tile high-level clocking structure. The clocking structure in a tile starts from a single clock source coming from the tile PLL or straight from the clock input pins.

For a tile to keep a low-jitter and high-speed clocking structure, a single input clock is routed to each RF-ADC or RF-DAC in the tile and divided for the dedicated signal conditioning functions within the RF-ADC or RF-DAC.

Figure 69: RF-ADC Clock Structure in a Tile
The clock coming out from the divider of the master tile can be used to drive the PL. A BUFG_GT buffer is automatically instantiated by the IP core when the output clock is used.

The following figure shows the clock connections for a typical Zynq® UltraScale+™ RFSoC RF Data Converter IP core. The IP core automatically instantiates and connects the constituent RF-ADC and RF-DAC tiles based on the selections in the Vivado® IDE.
In most applications the AXI4-Stream clock is common to multiple converters and is supplied from an external clock as shown in the previous figure. This clock must be derived from the same master clock source as the sampling and/or reference clocks to the RF-ADC and RF-DAC tiles. Alternatively, it is possible to use the output of one of the RF-DAC or RF-ADC tiles to generate the AXI4-Stream clocks as shown in the following figure.
Resets

Standard AXI4-Stream resets are provided for each clock domain. Reset of the RF-ADC and RF-DAC is done using the AXI4-Lite interface. A full tile reset is carried out by asserting bit 0 of the Master Reset Register. Individual tiles can be reset and disabled by writing to the Restart registers. Disabling the tile is achieved in a similar way. Follow the individual tile reset procedure to restart the tile after it has been disabled.

Related Information
- Master Reset Register (0x0004)
- Restart All Tiles
- Restart Tile
- Power-Down Tile
Power-up Sequence

Tiles consist of different independent blocks, powered from different power supplies and clocked by different versions of the main clock. Tiles must be brought up in a known sequence for the converters to function correctly. The power-up state machine is run automatically when the device is configured or reconfigured with a bitstream but it can also be rerun at any time under software control.

Figure 73: POR Finite State Machine

Power-on Sequence Steps

<table>
<thead>
<tr>
<th>Sequence Number</th>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-2</td>
<td>Device Power up and Configuration</td>
<td>The configuration parameters set in the Vivado® IDE are programmed into the converters. The state machine then waits for the external supplies to be powered up. In hardware this can take up to 25 ms. However this is reduced to 200 μs in behavioral simulations.</td>
</tr>
<tr>
<td>3-5</td>
<td>Power Supply Adjustment</td>
<td>The configuration settings are propagated to the analog sections of the converters. In addition the regulators, bias settings in the RF-DAC and the common-mode output buffer in the RF-ADC are enabled.</td>
</tr>
<tr>
<td>6-10</td>
<td>Clock Configuration</td>
<td>The state machine first detects the presence of a good clock into the converter. Then, if the PLL is enabled, it checks for PLL lock. The clocks are then released to the digital section of the converters.</td>
</tr>
<tr>
<td>11-13</td>
<td>Converter Calibration (ADC only)</td>
<td>Calibration is carried out in the RF-ADC. In hardware this can take approximately 10 ms, however this is reduced to 10 μs in behavioral simulations.</td>
</tr>
<tr>
<td>15</td>
<td>Done</td>
<td>The state machine has completed the power-up sequence.</td>
</tr>
</tbody>
</table>

Related Information

Interrupt Handling

RF-DAC/RF-ADC Tile <n> FIFO Disable Register (0x0230)
Restart All Tiles

To restart all tiles and perform the complete power-on sequence after initialization under software control, perform the following steps:

1. Write 0x0000_0001 to the Master Reset register to restart all tiles (start and end states revert to their default so there is no need to write 0x0000_000F to the individual tile Restart State register).

2. Poll the tile Restart Power-On State Machine register for each individual tile to check operation is complete. The power-on sequence is complete when this register reads all zeros.

Related Information
Master Reset Register (0x0004)
Restart State Register (0x0008)
Restart Power-On State Machine Register (0x0004)

Restart Tile

To restart a tile and perform the complete power-on sequence after initialization under software control, perform the following steps:

1. Write 0x0000_000F to the tile<n> Restart State register.

2. Write 0x0000_0001 to the tile<n> Restart Power-On State Machine register to restart the tile.

3. Poll the tile<n> Restart Power-On State Machine register to check operation is complete; the power-on sequence is complete when this register reads all zeros.

Related Information
Restart State Register (0x0008)
Restart Power-On State Machine Register (0x0004)

Power-Down Tile

To power down a tile, the power-on sequence is rerun in the same way as restarting a tile but with the End State set to 3 rather than F. Setting the End State to 3 makes the power-on sequencer clear all the registers without performing a full power-on sequence. Perform the following steps to power down a tile:

1. Write 0x0000_0003 to the individual tile Restart State register.

2. Write 0x0000_0001 to the tile<n> Restart Power-On State Machine register to restart the tile.
3. Poll the tile Restart Power-On State Machine register to check operation is complete; the power-down sequence is complete when this register reads all zeros.

Related Information
Restart State Register (0x0008)
Restart Power-On State Machine Register (0x0004)

Interfacing to the AXI4-Stream Interface

The number of AXI4-Stream interfaces and the width of each bus depends on the bandwidth required by the mode of operation selected in the Vivado® IDE. There is a 256-bit (DAC) or 128-bit (ADC) AXI4-Stream interface per converter (four per tile). The interface can run at up to 500 MHz so the maximum bandwidth available is 8 GSPS per RF-DAC and 4 GSPS per RF-ADC. Interpolation and decimation allow the AXI4-Stream bandwidth requirements to be reduced and the width of each AXI4-Stream interface can be selected on the IP core configuration screen in conjunction with the clock rate and interpolation/decimation settings. The following figure shows a single RF-DAC data input at 4 GSPS with 8x interpolation with I and Q on a single AXI4-Stream interface running at 500 MHz.

Figure 74: Single RF-DAC Input - 32-bit

The following figure shows an RF-ADC at 4 GSPS with 4x decimation with I and Q on a single AXI4-Stream interface running at 500 MHz.

Figure 75: Single RF-ADC Output

The following figure shows 2x32-bit RF-DACs at 4 GSPS with real data, 8x interpolation, and running at a 250 MHz AXI4-Stream clock.
Applications

This section gives an overview of how the Zynq UltraScale+™ RFSoC RF Data Converter features can be used in a system.

Multi-Converter Synchronization

RFSoC devices have very flexible clocking and data interfaces to enable a multitude of different applications within the same device. Each RF-ADC/RF-DAC tile has its own independent clocking and data infrastructure, which allows users to drive each tile with individual sample rates, and PL data-word widths. The converters within a single tile share the clocking and data infrastructure, so the sample rates and latency are fixed. However, certain applications might require more than one tile, or even more than one RFSoC device to be used together. For these applications, matching converter latency across tiles is critical. The multi-converter synchronization feature can be used to achieve this multi-tile and device alignment.

Related Information

Clocking

Detailed Description

In any system with multiple independent converters and clocking structures, there are several potential sources of latency uncertainty, such as the clock divider phase, FIFO latencies, clock skew, and data skew. Discrete converters have standardized the use of the JESD204B SYSREF scheme for synchronization, and as a result the RFSoC devices have implemented a complementary, simplified scheme using SYSREF.
The integration of the converters with the RFSoC results in the elimination of the serial transceiver links for data communication. However, to provide a flexible clocking and number of data words for the PL design, each RF-ADC and RF-DAC incorporates independent gearbox FIFOs. These FIFOs allow data to be transferred between the PL clock domain to the converter sample clock domain, but also result in a non-deterministic latency that must also be measured and corrected for by using SYSREF.
The Zynq® UltraScale+™ RFSoC RF Data Converter and RFdc driver API solution provide an easy to use and integrated way to synchronize each of the tiles in a given device. The previous figure shows an example system with multiple tiles which are to be synchronized. The following applies to the previous figure:

- Two RF-DAC tiles are shown for the purposes of illustration, but the method applies equally to any number of RF-ADC or RF-DAC tiles.
- A single analog SYSREF input per device is distributed internally to all RF-ADCs and RF-DACs.
- There is an internal analog SYSREF delay adjust per tile, to ease PCB requirements by allowing SYSREF setup/hold time to be optimized on-chip.
- There is an analog clock input per tile; these are shown as DAC0_CLK and DAC1_CLK.
- The common clock for the PL user design and tiles is shown as PL Clock (PL_clock).
- The SYSREF for the PL is shown as PL SYSREF (PL_SysRef). There is a separate PL SYSREF for RF-ADCs and RF-DACs if both are in the synchronization group.
- The synchronization state machine is contained within the core, which, in conjunction with the RFdc driver API, implements the synchronization solution.

**Synchronization Steps**

At a high level, the synchronization steps that are carried out are as follows:

1. Enable all clocks and SYSREF generators
   a. Both analog and digital clocks must be running and locked before synchronization begins
   b. Any change to the clocks requires resynchronization
2. SYSREF analog capture
   a. Ensures SYSREF is safely captured by auto-adjusting the internal SYSREF programmable delay for setup/hold
   b. This is done for each tile and requires a number of periodic SYSREF pulses so that the optimal delay value can be determined
3. Clock divider reset
   - When all tiles are safely capturing SYSREF, a subsequent SYSREF edge is used to synchronize all divider phases
4. FIFO latency measure and adjust
   a. Use the PL_SysRef signal to measure the latency through each FIFO
   b. Use the measurements across all tiles to adjust the latencies so that they match

Steps 2, 3, and 4 are handled automatically by the IP core and RFdc driver API.
RFdc Driver API Use

The RFdc driver API is used to initiate the synchronization procedure and check the status. The following example applies to the RF-DAC synchronization group. The same procedure should be run on the RF-ADC synchronization group.

```c
int status_dac;
XRFdc_MultiConverter_Sync_Config DAC_Sync_Config;        // Declare DAC MTS Settings structure
XRFdc_Multiconverter_Init (&DAC_Sync_Config, 0, 0);          // Initialize DAC MTS Settings
DAC_Sync_Config.Tiles = 0x3;                                             // Tiles to sync bit-mask: DAC tiles 0 and 1

// Run Multi-Tile-Sync for the DAC Group
status_dac = XRFdc_Multiconverter_Sync(RFdcInstPtr, XRFDC_DAC_TILE, &DAC_Sync_Config);
```

After synchronization, the synchronization status can be reported:

```c
if(status_dac == XRFDC_MTS_OK)
   xil_printf("INFO : DAC Multi-Tile-Sync completed successfully\n");
else
   xil_printf("ERROR : DAC Multi-Tile-Sync Issue - check metal_log for error information\n");

// Report Overall Latency in Sample Clocks and delay adjustments made in terms of PL words added to each Tile/FIFO
// xil_printf("\n\nMulti-Tile Sync Report ==\n");
for(i=0; i<4; i++) {
   if((1<<i)&DAC_Sync_Config.Tiles) {
      XRFdc_GetInterpolationFactor(RFdcInstPtr, i, 0, &factor);
      xil_printf(" DAC%d: Latency(Tl) =%3d, Adjusted Delay (T%d) =%3d \n", i, DAC_Sync_Config.Latency[i],
                  factor, DAC_Sync_Config.Offset[i]);
   }
}
```

**Note:** The apparent latency value reported must not be confused with the absolute FIFO or tile data latency; it is a relative measure of the alignment between tiles.
**Advanced Multi-Converter Sync API Usage**

The multi-converter API usage outlined in the previous section guarantees alignment between tiles in each group. However, the total latency through the tiles can vary by a number of words related to the FIFO read-clock period divided by the number of read-words. For applications where the total latency must be constant, the Target_Latency setting in the API can be enabled. This setting adjusts all delays less than the target to the target value; any measured values greater than the target result in an error. To prevent this error, the Target_Latency value must first be determined for the user FIFO and tile configuration by running XRFdc_MultiConverter.Sync with the target set to zero. The returned latency value (plus some margin) must be set as the target value for future runs.

The margin value to be applied is specified in terms of sample clocks. For the RF-ADC tiles, this value must be a multiple of the number of FIFO read-words times the decimation factor, and for RF-DAC tiles this value can be a constant of 16. An example of setting the target latency is as follows:

```c
XRFdc_MultiConverter_Init (&DAC_Sync_Config, 0, 0); // Initialize DAC MTS Settings
DAC_Sync_Config.Tiles = 0x3; // Tiles to sync bit-mask: DAC tiles 0 and 1
DAC_Sync_Config.Target_Latency = 296; // Target latency - measured value (280 + 16 margin)
```

**SYSREF Signal Requirements**

The SYSREF signal is the timing reference for the system and must therefore be handled correctly to ensure it does not degrade the synchronization. This signal has the following requirements.

1. The SYSREF signal must be a high-quality, free-running, low-jitter square wave, to allow it to be captured consistently by the analog sample clock.

2. The SYSREF frequency must meet the following requirements:
   a. If synchronizing RF-ADC and RF-DAC tiles with different sample frequencies, the frequency must be an integer submultiple of:
      
      \[
      \text{GCD}(\text{DAC\_Sample\_Rate}/16, \text{ADC\_Sample\_Rate}/16)
      \]

   b. SYSREF must also be an integer submultiple of all PL clocks that sample it. This is to ensure the periodic SYSREF is always sampled synchronously.

   c. Less than 10 MHz.

3. SYSREF must be safely captured by the PL, before passing to the core:
   a. Setup/hold of PL_SysRef to PL_clock must be handled as part of the user design.
b. Clock generation on the PCB can be used to facilitate this, if supported.

Related Information
Example SYSREF Frequency Calculation
PL SYSREF Capture

Example SYSREF Frequency Calculation

\[
\text{ADC}_{\text{SampleRate}} = 3.93216 \text{ GHz} \quad \text{DAC}_{\text{SampleRate}} = 4.91520 \text{ GHz}
\]

\[
\text{GCD}\left(\frac{\text{ADC}_{\text{SampleRate}}}{16}, \frac{\text{DAC}_{\text{SampleRate}}}{16}\right) = 61.44 \text{ MHz}
\]

\[
\text{Integer Sub – Multiple AND < 10 MHz} = 7.68 \text{ MHz}
\]

These equations generate a SYSREF frequency that satisfies the requirements of the RF-ADC/RF-DAC internal clocking. The SYSREF signal is also being captured synchronously by the PL; because of this, the SYSREF frequency must also be a submultiple of the PL clock driving the IP interfaces.

PL SYSREF Capture

To guarantee effective synchronization between tiles, it is important to reliably capture the PL SYSREF input in the device fabric. The PL SYSREF and PL Clock inputs must be differential signals that obey the clock and banking rules for the device. Dedicated clock inputs must be used for these inputs. The AXI4-Stream clocks for the Zynq® UltraScale+® RF Data Converter core must be generated from the PL Clock input and not from the clock outputs from the core itself. The following figure shows an example PL SYSREF capture circuit where the RF-ADC and RF-DAC are operating at the same AXI4-Stream clock frequency.
The `report_datasheet` command can be used in the Vivado® IDE to calculate the required setup and hold times for the PL SYSREF input at the device pins. The following figure shows an example calculation for a -1 speed grade device with a 500 MHz PL Clock.

In the example above, the PL SYSREF must be delayed to arrive at least 0.285 ns after the rising edge of PL Clock to ensure the hold requirement is met. The delay must be added externally to the device. For each new design, or when the design is changed, the `report_datasheet` step should be run to reevaluate the setup and hold requirements. If the RF-ADC and RF-DAC are operating at different AXI4-Stream clock frequencies, the circuit shown in the following figure can be used to capture PL SYSREF.
Here, clock network deskew MMCMs are used to divide down the frequency of the PL Clock to the required rates. The RF-ADC and RF-DAC AXI4-Stream clocks must be common submultiples of PL Clock. In addition, the PL SYSREF frequency must be a common submultiple of all the clocks in the system.

**Digital Feature Synchronization**

After the synchronization is complete, a common state for the clocking is shared between all tiles. The same synchronization infrastructure can also be used by the digital datapath features using Dynamic Update Events.

One example of this is the NCOs that make up the fine mixer features in the DDC and DUC chains. The NCO settings can be applied at the exact same moment across multiple tiles using SYSREF. Each NCO supports the dynamic setting of frequency and phase with the changes being applied on a SYSREF or other update event. Similarly, there is an option to reset the phase of the NCO on a SYSREF or other update event. For a single-tile synchronization, the tile event can be used to synchronize the NCOs within a tile. For multiple tile synchronization, either the MARKER event or SYSREF events can be used.

**Related Information**

*Dynamic Update Events*
Automatic Gain Control Systems

Automatic gain control (AGC) is commonly used in RF-ADC applications where the dynamic range of the input can vary considerably. It provides a way of using the input range of the RF-ADC and maximizing Signal-to-Noise ratio (SNR), while at the same time offering the flexibility to respond to varying signal amplitudes.

AGC systems consist of the following components:

- Variable Gain Amplifier (VGA)
- RF-ADC
- Signal amplitude monitoring
- AGC Algorithm / Decision logic
- Digital gain compensation
The RFSoC RF-ADC channels allow the implementation of custom AGC solutions by integrating the signal amplitude monitoring and compensation features into the RF-ADC tiles. These features can be used in conjunction with an external VGA and AGC logic embedded in the FPGA PL. This is shown in the following figure.

**Figure 81: Automatic Gain Control**

This figure shows an example AGC application. The signal amplitude monitoring is implemented within each RF-ADC channel using the threshold feature. This feature provides two thresholds that can be programmed per RF-ADC channel. When a threshold level is violated, this is indicated directly in the PL, thereby bypassing any latency within the datapath.

A sample high level operation of the AGC is as follows:

1. At system initialization, the threshold levels and modes are set by the RFdc driver API.
2. If a threshold level is violated:
   a. `adcXY_over_threshold` output flags assert.
   b. The PL-based AGC algorithm makes a decision and computes the new VGA gain and compensation gain
   c. Gain values are programmed into the VGA and digital compensation logic
d. `adcXY_pl_event` is asserted by the AGC logic.

e. Thresholds are cleared.

Related Information
Threshold Applications
Threshold Settings
Threshold RFdc Driver API Commands
Digital Gain Compensation
Clearing Threshold Flags

**Digital Gain Compensation**

As the VGA adjusts the analog signal level, it might be required to compensate this digitally after the RF-ADC to keep the overall signal-level constant. To minimize the disturbance on the received signal, it is possible to align the gain adjustments so that the digital compensation is applied at the same point in time relative to the analog gain being applied. This application moment must account for relative latencies between the propagation of the analog gain change through the VGA, RF-ADC and digital logic. There are a number of ways this gain adjustment and synchronized application can be achieved.

**Compensation Using QMC Gain**

The QMC Gain features are built into the RF-ADC tiles and offer a programmable gain between 0 and 2.0, with 16-bit output resolution. Using this, the 12-bit RF-ADC output signal can be placed to give a flexible dynamic range via the digital datapath. The application moment of the gain value in the QMC block can be controlled by the `adcXY_pl_event` signal. This signal is controlled directly from the user design in the PL and should be delayed to account for the latency required by the external VGA adjustment and RF-ADC conversion. When using this option, the sticky threshold outputs can be auto-cleared when the gain is applied.

Related Information
Clearing Threshold Flags

**Compensation Using PL Gain**

For wider dynamic range digital gain adjustment, the FPGA PL can be used. Similarly to the QMC Gain method, the latency through the VGA, RF-ADC and datapath should be used to delay the application of the gain change in the PL. After the PL gain change, the thresholds can be cleared by using the RFdc driver API, or by issuing a `adcXY_pl_event`. 
Determining Gain-Change Latency

To apply the digital gain compensation at the correct moment, the relative latency between the propagation of the gain change through the analog path must be aligned with the propagation of the digital gain update code through the digital path. Because the external VGA communication and response time is specific to the user application, these relative delays must be measured in-system. For the digital path, each tile has a dedicated input called `adcXY_pl_event` that can be used to apply a setting, such as gain, at a deterministic time. To determine the relative delay between the digital and analog paths, a test signal of constant amplitude should be applied during the design phase, and digital and analog gain adjustments applied. Looking at the output data, the delta time between the digital and analog gain application moments can be observed as a hump/through in the output signal amplitude. This delta time should be added to the `adcXY_pl_event` assertion in by counting periods of the RF-ADC output clock (`clk_adcX`). Using this predetermined delay during normal operation of the design ensures that the gain application moments are aligned.

The following figure illustrates the stages of responding to a threshold event for an example AGC application. The middle region of the diagram represents signals related to the analog/VGA path, while the lower region represents signals related to the digital path.

![Example AGC Threshold Event Response](image)

As shown in this figure, the AGC update is initiated by the assertion of a threshold. In response to this the AGC algorithm in the PL calculates gain adjustments to be made for both the analog and digital paths. The propagation of the delay through the analog path is shown by the VGA update write, VGA settling response, and RF-ADC latency. The propagation delay through the digital path is shown by the API write of the new gain value, denoted by `TRFdc_API`, and the `adcXY_pl_event` assertion, denoted by `TDelta_Latency`. `TDelta_Latency` is chosen by the user application to match the Digital and Analog latencies. After the digital gain update has been applied, the threshold can be automatically deasserted, if the Auto-Clear function is enabled.
It should be noted that the application of the QMC gain using the RFdc driver API might incur some driver overhead. In applications where this latency is undesirable, the PL Digital Gain Compensation method can be used.

Related Information
Digital Gain Compensation
Chapter 5

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 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.
Basic Tab

Figure 83: Basic Tab

Presets

The Vivado® IDE IP core configuration screen provides a method of saving and applying preset configurations. From the Presets menu, two fixed configurations are available.

- **4x4**: ADC:R2C DAC C2R: 4 RF-ADCs and 4 RF-DACs enabled. The RF-ADCs are configured in real to I/Q mode. The RF-DACs are configured in I/Q to real mode.
• **8x8:** ADC:R2C DAC C2R: 8 RF-ADCs and 8 RF-DACs enabled. The RF-ADCs are configured in real to I/Q mode. The RF-DACs are configured in I/Q to real mode.

In addition to these fixed presets, the current configuration can be saved by selecting **Save Configuration**... from the Presets menu. After the configuration is saved it can be reloaded at any time by selecting **Apply Configuration**....

**Component Name**

The component name is used as the base name of the output files generated for the core. Names must begin with a letter and must be composed from these characters: a through z, 0 through 9 and "_" (underscore).

**System Configuration**

Configuration Preset: This menu allows the user to select from a list of predefined configurations to avoid having to enter all settings manually.

**AXI4-Lite Interface Configuration**

AXI Clock Frequency (MHz): The core requires information on the frequency of the AXI4-Lite clock input to ensure the correct timing of the power-on sequence of the RF-ADC and RF-DAC blocks. The speed of the clock should be entered in MHz and must be between 10 MHz and 200 MHz.

**Converter Setup**

The converter setup menu provides advanced and simple options for setting up the IP configuration.

• **Advanced:** All tiles are independently configurable.

• **Simple:** The user is presented with the configuration options for a single RF-ADC and RF-DAC tile. After the tile is set up the RF-ADC configuration can be applied to any RF-ADC in the device by selecting the tile from the given tile list. Similarly the RF-DAC configuration can be applied to any RF-DAC tile.

It is possible to change from simple to advanced setup without losing configuration information. However, it is not possible to change from simple to advanced.
### RF-ADC Tab

**Figure 84: RF-ADC Tab**

<table>
<thead>
<tr>
<th>ADC Tile 224</th>
<th>ADC Tile 225</th>
<th>ADC Tile 226</th>
<th>ADC Tile 227</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PLL and Clocking Configuration</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enable PLL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sampling Rate (GSPS)</td>
<td>2.0</td>
<td>1.0 - 4.118</td>
<td></td>
</tr>
<tr>
<td>Reference Clock (MHz)</td>
<td>2000.000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clock Out (MHz)</td>
<td>15.625</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AXI4-Stream Clock (MHz)</td>
<td>250.000</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Multi Tile Sync</strong></td>
<td><strong>Link Coupling</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enable Multi Tile Sync</td>
<td>Link Coupling</td>
<td>AC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Converter Configuration</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADC 0</td>
<td>ADC 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enable ADC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Invert Q Output</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dither</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Data Settings</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Digital Output Data</td>
<td>Real</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decimation Mode</td>
<td>1x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Samples per AXI4-Stream Cycle</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Required AXI4-Stream clock: 250.000 MHz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Mixer Settings</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mixer Type</td>
<td>Bypassed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mixer Mode</td>
<td>Real-&gt;Real</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Analog Settings</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nyquist Zone</td>
<td>Zone 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Calibration Mode</td>
<td>Mode2</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Mixer Settings</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mixer Type</td>
<td>Bypassed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mixer Mode</td>
<td>Real-&gt;Real</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Analog Settings</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nyquist Zone</td>
<td>Zone 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Calibration Mode</td>
<td>Mode2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RF-ADC PLL and Clocking Configuration

These parameters are common to every RF-ADC in a tile. All the converters in a single tile share the same sampling rate and reference clock. Each tile also has a single output clock (ADC0 Clk Out in this case) which can be used to drive the AXI4-Stream clocks. The AXI4-Stream aclk frequency is calculated internally and is not configurable.

- **Enable PLL**: Selects whether the PLL within the RF-ADC tile (in the active RF-ADC Tile tab) is being used or bypassed. It is configurable when at least one of the converters in the tile is enabled. If it is required to dynamically reconfigure the PLL using the RFdc driver API, the PLL must be enabled in the Vivado® IDE. Valid values are True or False.

- **Sampling Rate**: Sets the sampling rate for the RF-ADC tile. It is configurable when at least one of the converters in the tile is enabled. Valid values are between 0.5 and 4.0 Gb/s depending on the device package.

- **Reference Clock (MHz)**: Sets the frequency of the clock input for the tile. It is configurable when the PLL and at least one of the converters in the tile is enabled. Its values depend on the sampling rate of the tile. A drop-down list of values based on the sample rate selected is presented. It is recommended to have reference frequencies above 250 MHz for optimum phase noise performance.

- **ADC Clock Out Frequency (MHz)**: Sets the frequency of the output clock on the tile. This clock can be used to drive the AXI4-Stream clock inputs. It is configurable when at least one of the converters in the tile is enabled. The values depend on the sampling rate of the tile. A drop-down list of values based on the sample rate selected is presented.

- **AXI4-Stream Clock (MHz)**: This is the frequency of the clock to be supplied on the \texttt{m_axis_aclk} input for the selected tile. The required frequency is determined by the settings in the RF-ADC Converter Configuration. Because all AXI4-Stream ports on a tile share a common AXI4-Stream clock all converter configurations on a tile must require the same clock frequency.

- **Link Coupling**: This determines if the input signal to the RF-ADC is AC or DC coupled. Typical applications use AC coupling, but for certain applications such as Zero IF, DC coupling is also supported. When DC link coupling mode is selected, the common mode voltage of the external driving circuit needs to be aligned with the RF-ADC internal common mode voltage by using the externally driven VCM pin (see Figure 8: RF-ADC Analog Input). The VCM pin is only enabled for DC coupling mode, and can be left floating if AC coupling mode is chosen.

- **Multi Tile Sync**: When enabled, the tile is included in a multi-converter synchronization group. Tile 224 must be enabled and present in the group. See Multi-converter synchronization for details.

Related Information
Multi-Converter Synchronization
**RF-ADC Converter Configuration**

- **Enable ADC:** Select if the selected converter within the selected tile is enabled. Valid values are True and False.
- **Invert Q Output:** This parameter is configurable only when I/Q output data is selected and the fine mixer is enabled. When set, the quadrature output of the mixer is negated. This allows -Q data to be generated.
- **Dither:** Selects if dither is enabled for the selected tile. Dither should be enabled unless the sample rate is under 3 GSPS for the 4G ADC (under 1.5 GSPS for the 2 GSPS variant).

**Data Settings**

- **Digital Output Data:** Sets the data type of the selected converter within the selected tile. The parameter is only configurable when the converter is enabled. Valid values are Real and I/Q. When converter 0 is set to I/Q, converter 1 must also be enabled and when converter 2 is set to I/Q, converter 3 must also be enabled; otherwise the configuration is invalid.
- **Decimation Mode:** Sets the decimation values of the selected converter within the selected tile. The parameter is only configurable when the converter is enabled. Values 1x, 2x, 4x, 8x are selected from a drop-down menu. When the converter is not enabled, the value is Off.
- **Samples per AXI4-Stream Word:** Sets the number of words per cycle. This parameter is configurable when the specific converter has been enabled. Valid values are between 1 and 8 and can be selected using a drop-down list. The range of values is limited depending on the selected sample rate to keep the AXI4-Stream clock less than 500 MHz. The required AXI4-Stream clock is an input to the IP core and its value, based on the selected bus width, is displayed.

**Mixer Settings**

- **Mixer Type:** Set the type of Mixer to be used. Valid options are bypassed, coarse, and fine. Selectable options depend on the selection in the converter Digital Output Data field.
- **Mixer Mode:** Set the mixer mode of the selected converter within the selected tile. The parameter is only configurable when the converter is enabled. The choice of mixer modes depends on the mixer type and the format of the digital output data selected. When real data is output the mixer is bypassed. When I/Q data is output the mixer can be set to Real to I/Q or I/Q to I/Q.
- **Coarse Mixer Frequency:** Sets the frequency of the coarse mixer. The parameter is configurable only when coarse is selected as the Mixer Type. Valid options are Fs/2, Fs/4 and -Fs/4.
- **Fine Mixer Frequency:** Sets the frequency of the fine mixer. The parameter is only available when Fine is selected as the Mixer Type. The valid range of frequencies is -10 GHz to 10 GHz.
- **Fine Mixer Phase:** Sets the phase of the fine mixer. The parameter is only available when fine is selected as the Mixer Type. Valid range is -180 to 180.
Analog Settings

- **Nyquist Zone**: Selects between even and odd Nyquist zone operation.
- **Calibration Mode**: Selects between different calibration optimization schemes depending on the features of the input signals. Mode 1 is optimal for input frequencies $F_{samp}/2(Nyquist) +/- 30\%$. Otherwise, use Mode 2.

**RF-DAC Tab**

*Figure 85: RF-DAC Tab*
RF-DAC PLL and Clocking Configuration

These parameters are common to every RF-DAC in a tile. All the converters in a single tile share the same sampling rate and reference clock. Each tile also has a single output clock which can be used to drive the AXI4-Stream clocks. The AXI4-Stream aclk frequency is calculated internally and is not configurable.

- **Enable PLL**: Selects whether the PLL within RF-DAC tile (in the active RF-DAC Tile Tab) is being used or bypassed. It is configurable when at least one of the converters in the tile is enabled. If it is required to dynamically reconfigure the PLL using the RFdc driver API, the PLL must be enabled in the Vivado® IDE. Valid values are True or False.

- **Sampling Rate (GSPS)**: Sets the sampling rate for the RF-DAC tile. It is configurable when at least one of the converters in the tile is enabled. Valid values are between 0.1 and 6.5 GSPS depending on the device package.

- **Reference Clock (MHz)**: Sets the frequency of the clock input for the tile. It is configurable when the tile PLL is enabled and at least one of the converters in the tile is enabled. Its values depend on the sample rate of the tile. A drop-down list of values based on the sample rate selected is presented. It is recommended to have reference frequencies above 250 MHz for optimum phase noise performance.

- **DAC Clk Out Frequency (MHz)**: Sets the frequency of the output clock on the tile. This clock can be used to drive the AXI4-Stream clock inputs. It is configurable when at least one of the converters in the tile is enabled. Its values depend on the sampling rate of the tile. A drop-down list of values based on the sample rate selected is presented.

- **AXI4-Stream Clock (MHz)**: This is the frequency of the clock to be supplied on the s_axis_aclk input for the selected tile. The required frequency is determined by the settings in RF-DAC Converter Configuration. Because all AXI4-Stream ports on a tile share a common AXI4-Stream clock, all converter configurations on a tile require the same clock frequency.

- **Multi Tile Sync**: When enabled the tile is included in a multi-converter synchronization group. Tile 228 must be enabled and present in the group. See Multi-converter Synchronization for details.

Related Information
- Multi-Converter Synchronization

RF-DAC Converter Configuration

- **Enable DAC**: Select if the selected converter within the selected tile is enabled. Valid values are True and False.

- **Invert Q Output**: When enabled, the Q output of the converter is inverted. This parameter is configurable only when I/Q output data is selected. A follow-on analog mixer must implement a subtraction, I x cos(f) - Q x sin(f) so the option to provide a -Q signal (negative quadrature) is provided so that the external mixer can perform the addition, I x cos(f) + (-Q)sin(f)
**Inverse Sinc Filter:** Select whether the Inverse Sinc Filter is enabled or not. The parameter is only configurable when the converter is enabled. Valid values are True and False.

**Data Settings**

**Analog Output Data:** Sets the data type of the selected converter within the selected tile. The parameter is only configurable when the converter is enabled. Valid values are Real and I/Q. When converter 0 is set to I/Q, converter 1 must also be enabled and when converter 2 is set to I/Q, converter 3 must also be enabled; otherwise the configuration is invalid.

**Interpolation Mode:** Sets the interpolation values of the current converter. The parameter is only configurable when the converter is enabled. Values 1x, 2x, 4x, 8x are selected from a drop-down menu. When the converter is not enabled, the value is Off.

**Samples per AXI4-Stream Word:** Sets the number of words per cycle. This parameter is configurable when at least one converter in the tile is enabled. Valid values are between 1 and 16 and can be selected using a drop-down list. The range of values is limited depending on the selected sample rate in order to keep the AXI4-Stream clock less than 500 MHz. The required AXI4-Stream clock is an input to the IP core and its value, based on the selected bus width, is displayed.

**Mixer Settings**

**Mixer Type:** Sets the type of Mixer to be used. Valid options are bypassed, coarse, and fine. Selectable options depend on the selection in the converter Analog Output Data field.

**Mixer Mode:** Sets the mixer mode of the selected converter within the selected tile. The parameter is only configurable when the converter is enabled. The choice of mixer modes depends on the selected Mixer Type and the format of the analog output data selected. When real data is output the mixer can be set to I/Q -> Real or Bypass and when I/Q data is output the mixer can be set to I/Q to I/Q.

**Coarse Mixer Frequency:** Sets the frequency of the coarse mixer. The parameter is configurable only when coarse is selected as the Mixer Type. Valid options are: Fs/2, Fs/4 and -Fs/4.

**Fine Mixer Frequency:** Sets the frequency of the fine mixer. This parameter is only available when fine is selected as the Mixer Type. The valid range of frequencies is -10 GHz to 10 GHz.

**Fine Mixer Phase:** Sets the phase of the fine mixer. The parameter is only available when fine is selected as the Mixer Type. Valid range is -180 to 180.

**Analog Settings**

**Nyquist Zone:** Selects between even and odd Nyquist zone operation.

**Decoder Mode:** The RF-DAC decoder can be optimized for low signal-to-noise ratio or for high linearity.
Advanced Tab

Figure 86: Advanced Tab

- **Analog Clock Detection**: Not available for customer use.

**RF-ADC**

- **Enable Debug Ports**: Not available for customer use.
- **Enable Real Time Ports**: Enables the RF-ADC Over Threshold, Over Voltage, Over Range and `pl_event` ports.
- **Enable Cal Freeze Ports**: Enables the calibration freeze logic. Calibration is frozen when the analog input is over voltage or the relevant `int_cal_freeze` input is asserted.
- **Auto Cal Freeze**: If this option is selected, the calibration is frozen when the analog input is over range, in addition to the manual calibration freeze process.
- **Calibration Startup**: Not available for customer use.

**RF-DAC**

- **Enable Debug Ports**: Not available for customer use.
- **Enable Real Time Ports**: Enables the RF-DAC Fast Shutdown and `pl_event` ports.
- **Output Current**: Select between 20 mA and 32 mA DAC output current. In 20 mA mode, `DAC_AVTT` is equal to 2.5V. In 32 mA mode, `DAC_AVTT` is equal to 3.0V.
Related Information
Real-Time Signal Interface Ports for RF-DACs
Real-Time Signal Interface Ports for 4 GSPS RF-ADCs
Real-Time Signal Interface Ports for 2 GSPS RF-ADCs
RF-DAC Output Current Mode

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 49: User Parameters

<table>
<thead>
<tr>
<th>Vivado IDE Parameter/Value</th>
<th>User Parameter/Value</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RF-ADC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADC Tile n Enable¹</td>
<td>C_ADCn_Enable</td>
<td>0</td>
</tr>
<tr>
<td>ADC Tile n Feedback Divider</td>
<td>C_ADCn_FBDiv</td>
<td>10</td>
</tr>
<tr>
<td>ADC Tile n Fabric Frequency</td>
<td>C_ADCn_Fabric_Freq</td>
<td>0</td>
</tr>
<tr>
<td>ADC Tile n Output Divider</td>
<td>C_ADCn_OutDiv</td>
<td>2</td>
</tr>
<tr>
<td>ADC Tile n Outclk Frequency</td>
<td>C_ADCn_Outclk_Freq</td>
<td>250</td>
</tr>
<tr>
<td>ADC Tile n PLL Enable</td>
<td>C_ADCn_PLL_Enable</td>
<td>FALSE</td>
</tr>
<tr>
<td>ADC Tile n Refclk Frequency</td>
<td>C_ADCn_Refclk_Freq</td>
<td>2000</td>
</tr>
<tr>
<td>ADC Tile n Sampling Rate</td>
<td>C_ADCn_Sampling_Rate</td>
<td>2</td>
</tr>
<tr>
<td>ADC Tile n Link Coupling</td>
<td>C_ADCn_Link_Coupling</td>
<td>0</td>
</tr>
<tr>
<td>ADC Tile n MultTile Sync</td>
<td>C_ADCn_Multi_Tile_Sync</td>
<td>FALSE</td>
</tr>
<tr>
<td>ADC Tile y Converter z Data Type²</td>
<td>C_ADC_Data_Typeyz</td>
<td>0</td>
</tr>
<tr>
<td>ADC Tile y Converter z Data Width</td>
<td>C_ADC_Data_Widthyz</td>
<td>8</td>
</tr>
<tr>
<td>ADC Tile y Converter z Decimation Mode</td>
<td>C_ADC_Decimation_Modeyz</td>
<td>1</td>
</tr>
<tr>
<td>ADC Tile y Converter z Mixer Mode</td>
<td>C_ADC_Mixer_Modeyz</td>
<td>2</td>
</tr>
<tr>
<td>ADC Tile y Converter z Nyquist Zone</td>
<td>C_ADC_Nyquistyz</td>
<td>1</td>
</tr>
<tr>
<td>ADC Tile y Converter z Calibration Optimization</td>
<td>C_ADC_CalOpt_Modeyz</td>
<td>0</td>
</tr>
<tr>
<td>ADC Tile y Converter z Enable</td>
<td>C_ADC_Sliceyz_Enable</td>
<td>TRUE</td>
</tr>
<tr>
<td>ADC Tile y Converter z Mixer Type</td>
<td>C_ADC_Mixer_Typeyz</td>
<td>0</td>
</tr>
<tr>
<td>ADC Tile y Converter z Fine Mixer Frequency</td>
<td>C_ADC_NCO_Freqyz</td>
<td>0.0</td>
</tr>
<tr>
<td>ADC Tile y Converter z Fine Mixer Phase</td>
<td>C_ADC_NCO_Phaseyz</td>
<td>0</td>
</tr>
<tr>
<td>ADC Tile y Converter z Coarse Mixer Phase</td>
<td>C_ADC_Coarse_Mixer_Freqyz</td>
<td>0</td>
</tr>
<tr>
<td>ADC Tile y Converter z Invert Q Output</td>
<td>C_ADC_Neg_Quadratureyz</td>
<td>FALSE</td>
</tr>
<tr>
<td>ADC Tile y Converter z Dither</td>
<td>C_ADC_Ditheryz</td>
<td>TRUE</td>
</tr>
<tr>
<td>Common</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AXI Clock Frequency</td>
<td>C_Axiclk_Freq</td>
<td>100</td>
</tr>
</tbody>
</table>
Table 49:  User Parameters (cont’d)

<table>
<thead>
<tr>
<th>Vivado IDE Parameter/Value</th>
<th>User Parameter/Value</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component Name</td>
<td>C_COMPONENT_NAME</td>
<td>usp_rf_data_converter_0</td>
</tr>
<tr>
<td>Enable RF-ADC Debug Pins</td>
<td>C_ADC_Debug</td>
<td>FALSE</td>
</tr>
<tr>
<td>Enable RF-DAC Debug Pins</td>
<td>C_DAC_Debug</td>
<td>FALSE</td>
</tr>
<tr>
<td>Enable RF-ADC Real-Time Signals</td>
<td>C_ADC_RTS</td>
<td>FALSE</td>
</tr>
<tr>
<td>Enable RF-DAC Real-Time Signals</td>
<td>C_DAC_RTS</td>
<td>FALSE</td>
</tr>
<tr>
<td>Enable RF-ADC Calibration Freeze</td>
<td>C_Calibration_Freeze</td>
<td>FALSE</td>
</tr>
<tr>
<td>Enable RF-ADC Auto Calibration Freeze</td>
<td>C_Auto_Calibration_Freeze</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

**RF-DAC**

| DAC Tile n Enable                   | C_DACn_Enable                 | 0                           |
| DAC Tile n Feedback Divider         | C_DACn_FBDiv                  | 10                          |
| DAC Tile n Fabric Frequency         | C_DACn_Fabric_Freq            | 0                           |
| DAC Tile n Output Divider           | C_DACn_OutDiv                 | 2                           |
| DAC Tile n Outclk Frequency         | C_DACn_Outclk_Freq            | 400                         |
| DAC Tile n PLL Enable               | C_DACn_PLL_Enable             | FALSE                       |
| DAC Tile n Refclk Frequency         | C_DACn_Refclk_Freq            | 6400                        |
| DAC Tile n Sampling Rate            | C_DACn_Sampling_Rate          | 6.5                         |
| DAC Tile n Multi Tile Sync          | C_DACn_Multi_Tile_Sync        | FALSE                       |
| DAC Tile y Converter z Data Type    | C_DAC_Data_Typeyz             | 0                           |
| DAC Tile y Converter z Data Width   | C_DAC_Data_Widthyz            | 16                          |
| DAC Tile y Converter z Interpolation Mode | C_DAC_Interpolation_Modeyz    | 0                           |
| DAC Tile y Converter z Decoder Mode | C_DAC_Decoder_Modeyz          | 0                           |
| DAC Tile y Converter z Invsinc Ctrl| C_DAC_Invsinc_Ctrlyz          | FALSE                       |
| DAC Tile y Converter z Mixer Mode   | C_DAC_Mixer_Modeyz            | 2                           |
| DAC Tile y Converter z Nyquist Zone | C_DAC_Nyquistyz               | 1                           |
| DAC Tile y Converter z Enable       | C_DAC_Sliceyz_Enable          | FALSE                       |
| DAC Tile y Converter z Mixer Type   | C_DAC_Mixer_Typeyz            | 0                           |
| DAC Tile y Converter z Fine Mixer Frequency | C_DAC_NCO_Freqyz             | 0.0                         |
| DAC Tile y Converter z Fine Mixer Phase | C_DAC_NCO_Phaseyz             | 0                           |
| DAC Tile y Converter z Coarse Mixer Phase | C_DAC_Coarse_Mixer_Freqyz    | 0                           |
| DAC Tile y Converter z Invert Q Output | C_DAC_Neg_Quadratureyz        | FALSE                       |

Notes:
1.  n = 0 to 3.
2.  y = 0 to 3, z = 0 to 3.
Output Generation

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

---

Simulation

Simulation is supported using Mentor Graphics Questa Advanced Simulator in the Vivado® Design Suite. 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). The RF-ADC and RF-DAC blocks have bit-accurate real number simulation models. This means that each of the datapath features can be simulated and the related analog signals observed.

Related Information
Example Design

---

Synthesis and Implementation

For details about synthesis and implementation, see the Vivado Design Suite User Guide: Designing with IP (UG896).
Chapter 6

Example Design

The following figure shows the example design that is provided with the core.

Figure 87: Example Design

The complete example design can be opened as a separate project by right-clicking the core in the project hierarchy after it has been customized using the IP catalog. Right-click the <component_name>.xci file in the Design Sources hierarchy in the Sources window and select Open IP Example Design. This opens a new Vivado® IP integrator project in a new window with a complete RF Data Converter IP example design.

The Zynq® UltraScale+™ RFSoC RF Data Converter example design consists of the following:

- An instance of the Zynq® UltraScale+® RF Data Converter IP core
- Data generator for RF-DAC input
- Data sink for RF-ADC output
- AXI interconnect for AXI-4 Lite addressing of the design
- An example chip level wrapper containing BUFGs
RF-ADC Sink

The RF-ADC sink checks that the digital sample data converted by the RF-ADC(s) in the Zynq® UltraScale+™ RFSoC RF Data Converter core match the signals generated by the demo test bench.

RF-DAC Source

The RF-DAC source generates digital sample data for the RF-DAC(s) in the Zynq® UltraScale+™ RFSoC RF Data Converter core.

The register maps for the RF-DAC data source are shown below.

Table 50: RF-DAC Data Source Register Map

<table>
<thead>
<tr>
<th>Base Address</th>
<th>Offset</th>
<th>Register</th>
<th>Default</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0x0</td>
<td>Enable</td>
<td>0</td>
<td>1</td>
<td>Global enable</td>
</tr>
</tbody>
</table>

The following register map is replicated for each of the AXI4-Stream interfaces in the tile data generators. The base address for each tile is offset by 0x1000. The base address for each AXI4-Stream interface is offset by 0x100.

Table 51: RF-DAC Data Source Register Map (Base Address = 0x1000 * Tile Number)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Default</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>dg_enable</td>
<td>0</td>
<td>16</td>
<td>Per word lane enable. Writing 0xffffffff enables all 16 lanes.</td>
</tr>
<tr>
<td>0x04</td>
<td>dg_inc</td>
<td>words</td>
<td>7</td>
<td>Increment value.</td>
</tr>
<tr>
<td>0x08</td>
<td>dg_type</td>
<td>1</td>
<td>4</td>
<td>Waveform type:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• 0: Unsigned triangle</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• 1: Signed triangle</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• 2: Ramp up</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• 3: Ramp down</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• 4: Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• 5: Sine wave</td>
</tr>
</tbody>
</table>
Table 51: RF-DAC Data Source Register Map (Base Address = 0x1000 * Tile Number) (cont’d)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Register</th>
<th>Default</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0C</td>
<td>dg_control</td>
<td>0</td>
<td>8</td>
<td>Output amplitude:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• 0x01: Divide by 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• 0x02: Divide by 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• 0x04: Divide by 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• 0x08 Divide by 16</td>
</tr>
<tr>
<td>0x40</td>
<td>dg_init_0</td>
<td>0</td>
<td>16</td>
<td>Word 0 initial value</td>
</tr>
<tr>
<td>0x44</td>
<td>dg_init_1</td>
<td>0</td>
<td>16</td>
<td>Word 1 initial value</td>
</tr>
<tr>
<td>0x48</td>
<td>dg_init_2</td>
<td>0</td>
<td>16</td>
<td>Word 2 initial value</td>
</tr>
<tr>
<td>0x4C</td>
<td>dg_init_3</td>
<td>0</td>
<td>16</td>
<td>Word 3 initial value</td>
</tr>
<tr>
<td>0x50</td>
<td>dg_init_4</td>
<td>0</td>
<td>16</td>
<td>Word 4 initial value</td>
</tr>
<tr>
<td>0x54</td>
<td>dg_init_5</td>
<td>0</td>
<td>16</td>
<td>Word 5 initial value</td>
</tr>
<tr>
<td>0x58</td>
<td>dg_init_6</td>
<td>0</td>
<td>16</td>
<td>Word 6 initial value</td>
</tr>
<tr>
<td>0x5C</td>
<td>dg_init_7</td>
<td>0</td>
<td>16</td>
<td>Word 7 initial value</td>
</tr>
<tr>
<td>0x60</td>
<td>dg_init_8</td>
<td>0</td>
<td>16</td>
<td>Word 8 initial value</td>
</tr>
<tr>
<td>0x64</td>
<td>dg_init_9</td>
<td>0</td>
<td>16</td>
<td>Word 9 initial value</td>
</tr>
<tr>
<td>0x68</td>
<td>dg_init_10</td>
<td>0</td>
<td>16</td>
<td>Word 10 initial value</td>
</tr>
<tr>
<td>0x6C</td>
<td>dg_init_11</td>
<td>0</td>
<td>16</td>
<td>Word 11 initial value</td>
</tr>
<tr>
<td>0x70</td>
<td>dg_init_12</td>
<td>0</td>
<td>16</td>
<td>Word 12 initial value</td>
</tr>
<tr>
<td>0x74</td>
<td>dg_init_13</td>
<td>0</td>
<td>16</td>
<td>Word 13 initial value</td>
</tr>
<tr>
<td>0x78</td>
<td>dg_init_14</td>
<td>0</td>
<td>16</td>
<td>Word 14 initial value</td>
</tr>
<tr>
<td>0x7C</td>
<td>dg_init_15</td>
<td>0</td>
<td>16</td>
<td>Word 15 initial value</td>
</tr>
</tbody>
</table>

Setting the Initial Value

The data generator uses a counter to generate triangles and ramps. Sine waves are generated using a look up table. The look up table contains a sine wave described over 128 digital values. Each 16-bit sample word on the AXI interface has its own counter and look-up table. The starting value of each counter and the starting location of each look-up table is set by the initial value register for that data word. To generate the desired input to the RF-DAC the initial values for each data word should be set accordingly.

For example, if a ramp with an increment of 3 is required at the input of a RF-DAC operating on real input data, the initial value for word 0 is set to 0, word 1 is set to 3, word 2 is set to 6 and so on. The increment register is set to the width of the data interface in words multiplied by the required increment of 3.
When the RF-DAC is set up for I/Q input, the even words carry the I samples and the odd words carry the Q samples. If you require a ramp with an increment of 3 on both the I and Q inputs, the initial values for words 0 and 1 are both set to 0, the initial values for words 2 and 3 are both set to 3, and so on. The increment register is set to the width of the data interface in words divided by 2 and multiplied by the required increment of 3.

When the data generator is operating in sine wave mode, the initial value register sets the starting address in the look-up table. In I/Q mode, if a cosine wave with an increment of 1 is required on the I input, the initial values for the even words are set to 32, 33, 34, and so on. If a sine wave of increment 1 is required on the Q input, the initial values for the odd words are set to 0, 1, 2, and so on. This gives the required 90 degree phase shift between the I and Q inputs. The increment register is set to the width of the data interface in words divided by 2.

Digital Data Format

Each word interface to the RF-DAC and RF-ADC is 16 bits wide, even though the RF-DAC resolution is 14 bits and the RF-ADC 12 bits. The word is in twos complement format, giving the range shown in the following table.

The RF-ADC and RF-DAC hex values are displayed appropriately left shifted to align to the input most significant bit.

Table 52: RF-ADC and RF-DAC Word Interface

<table>
<thead>
<tr>
<th></th>
<th>AXI Stream IF</th>
<th>ADC resolution (&lt;&lt;4)</th>
<th>DAC resolution (&lt;&lt;2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hex</td>
<td>Dec</td>
<td>Hex</td>
</tr>
<tr>
<td>Max positive</td>
<td>0x7FF</td>
<td>32767</td>
<td>0x7FF0</td>
</tr>
<tr>
<td></td>
<td>0x0001</td>
<td>1</td>
<td>0x0010</td>
</tr>
<tr>
<td>Zero</td>
<td>0x0000</td>
<td>0</td>
<td>0x0000</td>
</tr>
<tr>
<td></td>
<td>0xFFFF</td>
<td>-1</td>
<td>0xFFFF</td>
</tr>
<tr>
<td>Min negative</td>
<td>0x8000</td>
<td>-32768</td>
<td>0x8000</td>
</tr>
</tbody>
</table>
This chapter contains information about the test bench provided in the Vivado® Design Suite.

For information on setting up and running simulations in the Vivado Design Suite, see the Vivado Design Suite User Guide - Logic Simulation (UG900). Zynq® UltraScale+™ RFSoC RF Data Converter core supports behavioral simulation along with post-implementation functional simulation.

The following figure shows the structure of the Zynq® UltraScale+™ RFSoC RF Data Converter test bench.

*Figure 88: Demonstration Test Bench*

All blocks that require control are accessed through an AXI4-Lite interface. The Sequencer block is responsible for managing these accesses and contains examples of how to access and configure the basic test harness. Output similar to that in the following figures should be visible on the analog I/O.
Analog Signaling

The RF-DAC and RF-ADC analog I/O are differential and are driven as shown in the following figure in the test bench.

The internal RF-DAC and RF-ADC model analog functions using Real Number models. In the Verilog example design these must be passed as 64-bit wires, using force and hierarchical assignments to set and read the values.
A code excerpt from the `demo_tb.v` file is shown below. This is generated when you generate the example design. This can be used as a guide to show how to drive and read data into RF-DAC and RF-ADC primitives.

```vhdl
//-------------------------------------------------------------------------
// Force the analog signals.
//-------------------------------------------------------------------------
`ifndef DO_NOT_USE_RFAMS_REAL_SIGNAL_FORCE
// RF-ADC
real adc00_p;
real adc00_n;
always @ (*) begin
   // Map the RF-ADC signals to top level
   adc00_p = $bitstoreal(adc_source.vout_00_p);
   adc00_n = $bitstoreal(adc_source.vout_00_n);

   // Force the RF-ADC analog input
   force
       DUT.i_rf_dut_block.inst.rf_dut_rf_wrapper_i.rx0_u_adc.SIP_HSADC_INST._vin0_p = adc00_p;
   force
       DUT.i_rf_dut_block.inst.rf_dut_rf_wrapper_i.rx0_u_adc.SIP_HSADC_INST._vin0_n = adc00_n;
end
// RF-DAC
real dac00_p;
real dac00_n;
always @ (*) begin
   // Map the RF-DAC signals to the top level
   dac00_p = DUT.i_rf_dut_block.inst.rf_dut_rf_wrapper_i.tx0_u_dac.SIP_HSDAC_INST._vout0_p;
   dac00_n = DUT.i_rf_dut_block.inst.rf_dut_rf_wrapper_i.tx0_u_dac.SIP_HSDAC_INST._vout0_n;

   // Force the RF-DAC output onto the RF-DAC sink
   force dac_sink.vin_00_p = $realtobits(dac00_p);
   force dac_sink.vin_00_n = $realtobits(dac00_n);
end
`endif
```
Upgrading

Changes from V1.2 to V2.0

Port Changes

Separate RF-ADC and RF-DAC versions of the `user_sysref` port have been added. This change affects cores with multi-tile synchronization enabled.

*Table 53: Port Changes in Version 2.0*

<table>
<thead>
<tr>
<th>Port</th>
<th>Direction</th>
<th>Upgrade Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>user_sysref_adc</td>
<td>In</td>
<td>Connect to user_sysref, if required</td>
</tr>
<tr>
<td>user_sysref_dac</td>
<td>In</td>
<td>Connect to user_sysref, if required</td>
</tr>
</tbody>
</table>

Changes from V1.1 to V1.2

Port Changes

When the RF-ADC real-time signals are enabled the ports in the following table have been added.

*Table 54: Port Changes in Version 1.2*

<table>
<thead>
<tr>
<th>Port</th>
<th>Direction</th>
<th>Upgrade Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 GSPS RF-ADC^1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>adcXZ_over_range</td>
<td>Out</td>
<td>Leave open</td>
</tr>
<tr>
<td>adcXZ_over_voltage</td>
<td>Out</td>
<td>Leave open</td>
</tr>
<tr>
<td>4 GSPS RF-ADC^2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>adcX ZZ_over_range</td>
<td>Out</td>
<td>Leave open</td>
</tr>
</tbody>
</table>
Table 54: Port Changes in Version 1.2 (cont’d)

<table>
<thead>
<tr>
<th>Port</th>
<th>Direction</th>
<th>Upgrade Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>adcX_ZZ_over_voltage</td>
<td>Out</td>
<td>Leave open</td>
</tr>
</tbody>
</table>

Notes:
1. X refers to the location of the tile in the converter column. Z refers to the location of the RF-ADC in the tile (0 to 3).
2. X refers to the location of the tile in the converter column. ZZ is either 01 (the lower RF-ADC in the tile) or 23 (the upper RF-ADC in the tile)

When the RF-ADC real-time signals are enabled the ports in the following table have been renamed.

Table 55: Ports Renamed in Version 1.2

<table>
<thead>
<tr>
<th>Port</th>
<th>Previous Name (v1.1)</th>
<th>Direction</th>
<th>Upgrade Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 GSPS RF-ADC1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>adcXZ_over_threshold1</td>
<td>adcXY_over_threshold1</td>
<td>Out</td>
<td>Leave open</td>
</tr>
<tr>
<td>adcXZ_over_threshold2</td>
<td>adcXY_over_threshold2</td>
<td>Out</td>
<td>Leave open</td>
</tr>
<tr>
<td>4 GSPS RF-ADC2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>adcX_ZZ_over_threshold1</td>
<td>adcXY_over_threshold1</td>
<td>Out</td>
<td>Leave open</td>
</tr>
<tr>
<td>adcX_ZZ_over_threshold2</td>
<td>adcXY_over_threshold2</td>
<td>Out</td>
<td>Leave open</td>
</tr>
</tbody>
</table>

Notes:
1. X refers to the location of the tile in the converter column. Y refers to the location of the DDC block in the tile (0 to 3). Z refers to the location of the RF-ADC in the tile (0 to 3).
2. X refers to the location of the tile in the converter column. Y refers to the location of the DDC block in the tile (0 to 3). ZZ is either 01 (the lower RF-ADC in the tile) or 23 (the upper RF-ADC in the tile).

Parameter Changes

There are no parameter changes.
Debugging

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

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.

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.

Answer Records

Answer Records include information about commonly encountered problems, helpful information on how to resolve these problems, and any known issues with a Xilinx product. Answer Records are created and maintained daily ensuring that users have access to the most accurate information available.

Answer Records for this core can be located by using the Search Support box on the main Xilinx support web page. To maximize your search results, use keywords such as:

- Product name
- Tool message(s)
- Summary of the issue encountered

A filter search is available after results are returned to further target the results.
**Master Answer Record for the Core**

AR 69907.

**Technical Support**

Xilinx provides technical support in the Xilinx Support web page 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 contact Xilinx Technical Support, navigate to the Xilinx Support web page.

---

**Debug Tools**

There are many tools available to address Zynq® UltraScale+® RF Data Converter design issues. It is important to know which tools are useful for debugging various situations.

**Vivado Design Suite Debug Feature**

The Vivado® Design Suite debug feature inserts logic analyzer and virtual I/O cores directly into your design. The debug feature also allows you to set trigger conditions to capture application and integrated block port signals in hardware. Captured signals can then be analyzed. This feature in the Vivado IDE is used for logic debugging and validation of a design running in Xilinx® devices.

The Vivado logic analyzer is used to interact with the logic debug LogiCORE IP cores, including:

- ILA 2.0 (and later versions)
- VIO 2.0 (and later versions)

See the **Vivado Design Suite User Guide: Programming and Debugging (UG908)**.
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 incorporated from the example design 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.
- If your outputs go to 0, check your licensing.

Interface Debug

AXI4-Lite Interfaces

Read from a register that does not have all 0s as a default to verify that the interface is functional. Output \texttt{s\_axi\_arready} asserts when the read address is valid, and output \texttt{s\_axi\_rvalid} asserts when the read data/response is valid. If the interface is unresponsive, ensure that the following conditions are met:

- The \texttt{s\_axi\_aclk} and \texttt{aclk} inputs are connected and toggling.
- The interface is not being held in reset, and \texttt{s\_axi\_areset} is an active-Low reset.
- The interface is enabled, and \texttt{s\_axi\_aclken} is active-High (if used).
- The main core clocks are toggling and that the enables are also asserted.
- If the simulation has been run, verify in simulation and/or a debug feature capture that the waveform is correct for accessing the AXI4-Lite interface.
**AXI4-Stream Interfaces**

If data is not being transmitted or received, check the following conditions:

- If transmit `<interface_name>_tready` is stuck Low following the `<interface_name>_tvalid` input being asserted, the core cannot send data.
- If the receive `<interface_name>_tvalid` is stuck Low, the core is not receiving data.
- Check that the aclk inputs are connected and toggling.
- Check that the AXI4-Stream waveforms are being followed.
- Check core configuration.

**Related Information**

*Interfacing to the AXI4-Stream Interface*
Overview

The RFdc driver API functions for the Zynq® UltraScale+™ RFSoC RF Data Converter are described in this chapter. The same driver is used for both bare metal and Linux. The driver for both software platforms runs on the libmetal software layer provided by Xilinx. The driver is composed of the following files:

- **API**
  - `xrfdc.c`: The user interface API functions are implemented in this file.
  - `xrfdc.h`: The user interface API prototypes are provided in this file. The file provides prototypes of the driver instance structure and all other structures used across the APIs. The file implements utility in-line functions to access various data in the driver and IP.
  - `xrfdc_mts.c`: The multi-tile synchronization API functions are implemented in this file.
  - `xrfdc_mts.h`: The multi-tile synchronization API prototypes are provided in this file.

- **Hardware Register Map**
  - `xrfdc_hw.h`: Definitions for the hardware register maps are provided in this file. The file also provides masks for various relevant fields for the Zynq UltraScale+ RFSoC register interface.

- **Interrupt Handling**
  - `xrfdc_intr.c`: Implements functions for the handling of various interrupts and errors from the IP core.
About Libmetal

Libmetal is a Xilinx developed open source software stack that provides common user APIs to access devices, handle device interrupts, and request memory across the following operating environments:

- Linux user space (based on Userspace IO and Virtual Function IO support in the kernel)
- RTOS (with and without virtual memory)
- Bare-metal environments

The libmetal I/O region abstraction provides access to memory mapped I/O and shared memory regions. This includes primitives to read and write memory with ordering constraints and the ability to translate between physical and virtual addressing on systems that support virtual memory.

Data Structures

All the data structures used in the driver are defined in the `xrfdc.h` file. Except for a few, all structures are used by the drivers only. Do not use them directly in your applications.
struct XRFdc_PLL_Settings

This structure is for internal driver use.

```
u32 Enabled;
double RefClkFreq;
double SampleRate;
u32 RefClkDivider;
u32 FeedbackDivider;
u32 OutputDivider;
u32 FractionalMode;
u64 FractionalData;
u32 FractWidth;
```

Description

- **u32 Enabled**: Indicates if the PLL is enabled (1) or disabled (0).
- **double RefClkFreq**: Reference clock frequency (MHz).
- **double SampleRate**: Sampling rate (GHz).
- **u32 RefClkDivider**: Reference clock divider.
- **u32 FeedbackDivider**: Feedback divider.
- **u32 OutputDivider**: Output divider.
- **u32 FractionalMode**: Fractional mode.
- **u64 FractionalData**: Fractional part of the feedback divider.
- **u32 FractWidth**: Fractional data width.

struct XRFdc_QMC_Settings

This structure is used to set or get the QMC settings.

```
u32 EnablePhase;
u32 EnableGain;
double GainCorrectionFactor;
double PhaseCorrectionFactor;
s32 OffsetCorrectionFactor;
u32 EventSource;
```

Description

- **u32 EnablePhase**: Indicates if phase is enabled (1) or disabled (0).
- **u32 EnableGain**: Indicates if gain is enabled (1) or disabled (0).
- **double GainCorrectionFactor**: Gain correction factor. Range: 0 to 2.0.
• **double PhaseCorrectionFactor**: Phase correction factor. Range: +/- 26.5 degrees.
• **s32 OffsetCorrectionFactor**: Offset correction factor is adding a fixed LSB value to the sampled signal.
• **u32 EventSource**: Event source for QMC settings. XRFDC_EVNT_SRC_* represents valid values.

**struct XRFdc_CoarseDelay_Settings**

This structure is used to set or get coarse delay settings.

```c
struct XRFdc_CoarseDelay_Settings {
    u32 CoarseDelay;
    u32 EventSource;
};
```

**Description**

• **u32 CoarseDelay**: Coarse delay in the number of samples. Range: 0 to 7.
• **u32 EventSource**: Event source for coarse delay settings. XRFDC_EVNT_SRC_* represents valid values.

**struct XRFdc_Mixer_Settings**

This structure is used to set or get mixer settings.

```c
struct XRFdc_Mixer_Settings {
    double Freq;
    double PhaseOffset;
    u32 EventSource;
    u32 FineMixerMode;
    u32 CoarseMixFreq;
    u8 FineMixerScale;
    u32 CoarseMixMode;
};
```

**Description**

• **double Freq**: NCO frequency. Range: -Fs/2 to Fs/2 (GHz).
• **double PhaseOffset**: NCO phase offset. Range: -180 to 180.
• **u32 EventSource**: Event source for mixer settings. XRFDC_EVNT_SRC_* represents valid values.
• **u32 FineMixerMode**: Mixer mode for fine mixer. XRFDC_FINE_MIXER_MOD_* represents valid values.
• **u32 CoarseMixFreq**: Coarse mixer frequency. XRFDC_COARSE_MIX_* represents valid values.
• **u8 FineMixerScale**: NCO output scale. `XRFDC_MIXER_SCALE_*` represents valid values.
• **u32 CoarseMixMode**: Mixer mode for coarse mixer. `XRFDC_COARSE_MIX_MODE_*` represents valid values.

### struct XRFdc_Threshold_Settings

This structure is used to set or get RF-ADC threshold settings.

```c
struct XRFdc_Threshold_Settings {
    u32 UpdateThreshold;
    u32 ThresholdMode[2];
    u32 ThresholdAvgVal[2];
    u32 ThresholdUnderVal[2];
    u32 ThresholdOverVal[2];
};
```

**Description**

• **u32 UpdateThreshold**: Selects which threshold to update. `XRFDC_UPDATE_THRESHOLD_*` represents valid values.

• **u32 ThresholdMode[2]**: Entry 0 is for Threshold0 and 1 for Threshold1. Range: 0 to 3 (0-OFF, 1-sticky-over, 2-sticky-under and 3-hysteresis)

• **u32 ThresholdAvgVal[2]**: Threshold average value. Entry 0 is for Threshold0 and 1 for Threshold1.

• **u32 ThresholdUnderVal[2]**: Under threshold value. Entry 0 is for Threshold0 and 1 for Threshold1.

• **u32 ThresholdOverVal[2]**: Over threshold value. Entry 0 is for Threshold0 and 1 for Threshold1.

### struct XRFdc_TileStatus

This structure is for internal driver use.

```c
struct XRFdc_TileStatus {
    u32 IsEnabled;
    u32 TileState;
    u8  BlockStatusMask;
    u32 PowerUpState;
    u32 PLLState;
};
```

**Description**

• **u32 IsEnabled**: Indicates tile is enabled (1) or disabled (0).

• **u32 TileState**: Indicates current tile state.

• **u8 BlockStatusMask**: Bit mask for converter status. 1 indicates converter enable.
• **u32 PowerUpState**: Indicates power-up status.
• **u32 PLLState**: Indicates if PLL is locked or unlocked.

**struct XRFdc_IPStatus**

This structure is used to get the IP core status.

```c
XRFdc_TileStatus DACTileStatus[4];
XRFdc_TileStatus ADCTileStatus[4];
u32 State;
```

**Description**

• **XRFdc_TileStatus DACTileStatus[4]**: Tile status for four RF-DAC tiles.
• **XRFdc_TileStatus ADCTileStatus[4]**: Tile status for four RF-ADC tiles.
• **u32 State**: Not currently supported.

**struct XRFdc_BlockStatus**

This structure is used to get the status of RF-DACs or RF-ADCs.

```c
double SamplingFreq;
u32 AnalogDataPathStatus;
u32 DigitalDataPathStatus;
u8 DataPathClocksStatus;
u8 IsFIFOFlagsEnabled;
u8 IsFIFOFlagsAsserted;
```

**Description**

• **double SamplingFreq**: Sampling frequency.
• **u32 AnalogDataPathStatus**:
  • RF-ADC
    • [bit[0]] Converter enable/disable.
  • RF-DAC
    • [3:0] Inverse sinc enable/disable.
    • [7:4] Decoder mode.
• **u32 DigitalDataPathStatus**:
• RF-ADC
  • [3:0] FIFO status (enable/disable).
• RF-DAC
  • [3:0] FIFO status.
  • [7:4] Interpolation factor.
• u8 DataPathClocksStatus: Indicates if all required datapath clocks are enabled; 1 if all clocks enabled, 0 otherwise.
• u8 IsFIFOFlagsEnabled: FIFO flags enabled mask; 1 is enabled, otherwise 0.
• u8 IsFIFOFlagsAsserted: FIFO flags asserted mask; 1 is enabled, otherwise 0.

**struct XRFdc_DACBlock_AnalogDataPath_Config**

This structure is for internal driver use.

```c
u32 BlockAvailable;
u32 InvSyncEnable;
u32 MixMode;
u32 DecoderMode;
```

**Description**

• **u32 BlockAvailable:** Corresponds to the C_DAC_Slice{xy}_Enable IP parameter.
• **u32 InvSyncEnable:** Corresponds to the C_DAC_Invsinc_Ctrl{xy} IP parameter.
• **u32 MixMode:** Corresponds to the C_DAC_Mixer_Mode{xy} IP parameter.
• **u32 DecoderMode:** Corresponds to the C_DAC_Decoder_Mode{xy} IP parameter.
struct XRFdc_DACBlock_DigitalDataPath_Config

This structure is for internal driver use.

```c
u32 DataType;
u32 DataWidth;
u32 InterpolationMode;
u32 FifoEnable;
u32 AdderEnable;
```

Description

- **u32 DataType**: Corresponds to the C_DAC_Data_Type{xy} IP parameter.
- **u32 DataWidth**: Corresponds to the C_DAC_Data_Width{xy} IP parameter.
- **u32 InterpolationMode**: Corresponds to the C_DAC_Interpolation_Mode{xy} IP parameter.
- **u32 FifoEnable**: Corresponds to the C_DAC_Fifo{xy} IP parameter.
- **u32 AdderEnable**: Corresponds to the C_DAC_Adder{xy} IP parameter.

struct XRFdc_ADCBlock_AnalogDataPath_Config

This structure is for internal driver use.

```c
u32 BlockAvailable;
u32 MixMode;
```

Description

- **u32 BlockAvailable**: Corresponds to the C_ADC_Slice{xy}_Enable IP parameter.
- **u32 MixMode**: Corresponds to the C_ADC_Mixer_Mode{xy} IP parameter.

struct XRFdc_ADCBlock_DigitalDataPath_Config

This structure is for internal driver use.

```c
u32 DataType;
u32 DataWidth;
u32 DecimationMode;
u32 FifoEnable;
```
Description

- **u32 DataType**: Corresponds to the C_ADC_Data_Type{xy} IP parameter.
- **u32 DataWidth**: Corresponds to the C_ADC_Data_Width{xy} IP parameter.
- **u32 DecimationMode**: Corresponds to the C_ADC_Decimation_Mode{xy} IP parameter.
- **u32 FifoEnable**: Corresponds to the C_ADC_Fifo{xy} IP parameter.

```c
struct XRFdc_DACTile_Config

This structure is for internal driver use.
```

```c
u32 Enable;
u32 PLLEnable;
double SamplingRate;
double RefClkFreq;
double FabClkFreq;
XRFdc_DACBlock_AnalogDataPath_Config DACBlock_Analog_Config[4];
XRFdc_DACBlock_DigitalDataPath_Config DACBlock_Digital_Config[4];
```

Description

- **u32 Enable**: Corresponds to the C_DAC{x}_Enable IP parameter.
- **u32 PLLEnable**: Corresponds to the C_DAC{x}_PLL_Enable IP parameter.
- **double SamplingRate**: Corresponds to the C_DAC{x}_Sampling_Rate IP parameter.
- **double RefClkFreq**: Corresponds to the C_DAC{x}_RefClk_Freq IP parameter.
- **double FabClkFreq**: Corresponds to the C_DAC{x}_FabClk_Freq IP parameter.

```c
struct XRFdc_ADCTile_Config

This structure is for internal driver use.
```

```c
u32 Enable;
u32 PLLEnable;
double SamplingRate;
double RefClkFreq;
double FabClkFreq;
XRFdc_ADCBlock_AnalogDataPath_Config ADCBlock_AnalogConfig[4];
XRFdc_ADCBlock_DigitalDataPath_Config ADCBlock_Digital_Config[4];
```

Description

- **u32 Enable**: Corresponds to the C_ADC{x}_Enable IP parameter.
- **u32 PLLEnable**: Corresponds to the C_ADC{x}_PLL_Enable IP parameter.
• double SamplingRate: Corresponds to the C_ADC{x}_Sampling_Rate IP parameter.
• double RefClkFreq: Corresponds to the C_ADC{x}_RefClk_Freq IP parameter.
• double FabClkFreq: Corresponds to the C_ADC{x}_FabClk_Freq IP parameter.

**struct XRFdc_Config**

This structure is for internal driver use (metal_phys_addr_t represents unsigned long).

```c
struct XRFdc_Config {
    u32 DeviceId;
    #ifdef _MICROBLAZE_
        metal_phys_addr_t BaseAddr;
    #else
        u32 BaseAddr;
    #endif /* IP Base Address */
    u32 ADCType;
    u32 MasterADCTile;
    u32 MasterDACTile;
    u32 ADCSysRefSource;
    u32 DACSysRefSource;
    XRFdc_DAConfig DACTile_Config[4];
    XRFdc_ADConfig ADCTile_Config[4];
};
```

**Description**

• u32 ADCType: 2 GSPS RF-ADC part or 4 GSPS RF-ADC part.
• u32 MasterADCTile: Corresponds to the C_ADC_Master IP parameter.
• u32 MasterDACTile: Corresponds to the C_DAC_Master IP parameter.
• u32 ADCSysRefSource: Corresponds to the C_ADC_Sysref_Source IP parameter.
• u32 DACSysRefSource: Corresponds to the C_DAC_Sysref_Source IP parameter.

**struct XRFdc_DACBlock_AnalogDataPath**

This structure is for internal driver use.

```c
struct XRFdc_DACBlock_AnalogDataPath {
    u32 Enabled;
    u32 MixedMode;
    double TerminationVoltage;
    double OutputCurrent;
    u32 InverseSincFilterEnable;
    u32 DecoderMode;
    void * FuncHandler;
    XRFdc_QMC_Settings QMC_Settings;
    XRFdc_CoarseDelay_Settings CoarseDelay_Settings;
    u32 NyquistZone;
};
```
Description
- **u32 Enabled**: RF-DAC enable (1) or disable (0).
- **u32 MixedMode**: Mixer mode.
- **double TerminationVoltage**: Termination voltage.
- **double OutputCurrent**: Output current.
- **u32 InverseSincFilterEnable**: Inverse sinc filter enable (1) or disable (0).
- **u32 DecoderMode**: Decoder mode.
- **void * FuncHandler**: Function handler (currently not used in the driver).
- **XRFdc_QMC_Settings QMC_Settings**: QMC settings structure.
- **XRFdc_CoarseDelay_Settings CoarseDelay_Settings**: CoarseDelay settings structure.
- **u32 NyquistZone**: Nyquist zone.

**struct XRFdc_DACBlock_DigitalDataPath**

This structure is for internal driver use.

```c
u32 DataType;
u32 DataWidth;
int ConnectedIData;
int ConnectedQData;
u32 InterpolationFactor;
XRFdc_Mixer_Settings Mixer_Settings;
```

Description
- **u32 DataType**: Digital input datatype.
- **u32 DataWidth**: Data width (samples per AXI4-Stream word).
- **int ConnectedIData**: Data converter connected for I datapath. Valid values are 0-3 and -1.
- **int ConnectedQData**: Data converter connected for Q datapath. Valid values are 0-3 and -1.
- **u32 InterpolationFactor**: Interpolation factor.
- **XRFdc_Mixer_Settings Mixer_Settings**: Mixer settings structure.
struct XRFdc_ADCBlock_AnalogDataPath

This structure is for internal driver use.

```
struct XRFdc_ADCBlock_AnalogDataPath {
    u32 Enabled;
    XRFdc_QMC_Settings QMC_Settings;
    XRFdc_CoarseDelay_Settings CoarseDelay_Settings;
    XRFdc_Threshold_Settings Threshold_Settings;
    u32 NyquistZone;
    u8 CalibrationMode;
}
```

Description

- **u32 Enabled**: RF-ADC Enable (1) or Disable (0).
- **XRFdc_QMC_Settings QMC_Settings**: QMC settings structure.
- **XRFdc_CoarseDelay_Settings CoarseDelay_Settings**: Coarse delay settings structure.
- **XRFdc_Threshold_Settings Threshold_Settings**: Threshold settings structure.
- **u32 NyquistZone**: Nyquist zone.
- **u8 CalibrationMode**: Calibration mode, set by the XRFdc_SetCalibrationMode API function.

struct XRFdc_ADCBlock_DigitalDataPath

This structure is for internal driver use.

```
struct XRFdc_ADCBlock_DigitalDataPath {
    u32 DataType;
    u32 DataWidth;
    u32 DecimationFactor;
    int ConnectedIData;
    int ConnectedQData;
    XRFdc_Mixer_Settings Mixer_Settings;
}
```

Description

- **u32 DataType**: Analog input data type.
- **u32 DataWidth**: Data width (samples per AXI4-Stream word).
- **u32 DecimationFactor**: Decimation factor.
- **int ConnectedIData**: Data converter connected for I datapath. Valid values are 0-3 and -1.
- **int ConnectedQData**: Data converter connected for Q datapath. Valid values are 0-3 and -1.
- **XRFdc_Mixer_Settings Mixer_Settings**: Mixer settings structure.
struct XRFdc_DAC_Tile

This structure is for internal driver use.

```c
u32 TileBaseAddr;
u32 NumOfDACBlocks;
XRFdc_PLL_Settings PLL_Settings;
XRFdc_DACBlock_AnalogDataPath DACBlock_Analog_Datapath[4];
XRFdc_DACBlock_DigitalDataPath DACBlock_Digital_Datapath[4];
```

Description

- **u32 TileBaseAddr**: Tile base address.
- **u32 NumOfDACBlocks**: Number of RF-DACs enabled.
- **XRFdc_PLL_Settings PLL_Settings**: PLL settings structure.
- **XRFdc_DACBlock_AnalogDataPath DACBlock_Analog_Datapath[4]**: DACBlock_AnalogDataPath structure for four RF-DACs.
- **XRFdc_DACBlock_DigitalDataPath DACBlock_Digital_Datapath[4]**: DACBlock_DigitalDataPath structure for four RF-DACs.

struct XRFdc_ADC_Tile

This structure is for internal driver use.

```c
u32 TileBaseAddr;
u32 NumOfADCBlocks;
XRFdc_PLL_Settings PLL_Settings;
XRFdc_ADCBlock_AnalogDataPath ADCBlock_Analog_Datapath[4];
XRFdc_ADCBlock_DigitalDataPath ADCBlock_Digital_Datapath[4];
```

Description

- **u32 TileBaseAddr**: Tile base address.
- **u32 NumOfADCBlocks**: Number of RF-ADCs enabled.
- **XRFdc_PLL_Settings PLL_Settings**: PLL settings structure.
- **XRFdc_ADCBlock_AnalogDataPath ADCBlock_Analog_Datapath[4]**: ADCBlock_AnalogDataPath structure for four RF-ADCs.
- **XRFdc_ADCBlock_DigitalDataPath ADCBlock_Digital_Datapath[4]**: ADCBlock_DigitalDataPath structure for four RF-ADCs.
struct XRFdc //Driver instance structure

This structure is for internal driver use.

XRFdc_Config RFdc_Config;
void *CallBackRef;

Description

- **XRFdc_Config RFdc_Config**: Driver structure configuration.
- **u32 IsReady**: This flag is used to indicate that the driver is ready.
- **u32 ADC4GSPS**: This flag is used to indicate if RF-ADC is 4 GSPS.
- **metal_phys_addr_t BaseAddr**: Base address.
- **struct metal_io_region**: Libmetal IO structure.
- **struct metal_device *device**: Libmetal device structure.
- **XRFdc_DAC_Tile DAC_Tile[4]**: RF-DAC tile structure for four tiles.
- **XRFdc_ADC_Tile ADC_Tile[4]**: RF-ADC tile structure for four tiles.
- **XRFds_StatusHandler StatusHandler**: Event handler function.
- **void *CallbackRef**: Callback reference for event handler.
- **u8 UpdateMixerScale**: Set to 1 to overwrite mixer scale.
struct XRFdc_MultiConverter_Sync_Config

This structure is used to configure the MTS algorithm. Indicated files can be configured in the code.

u32 RefTile;
u32 Tiles;
int Target_Latency;
int Offset[4];
int Latency[4];
int Marker_Delay;
int SysRef_Enable;
XRFdc_MTS_DTC_Settings DTC_Set_PLL;
XRFdc_MTS_DTC_Settings DTC_Set_T1;

Description

- **u32 RefTile**: Reference tile (must be 0).
- **u32 Tiles**: Bitmask indicating which tiles to align. BitX enables MTS for TileX. Tile0 must always be enabled.
- **int Target_Latency**: Sets the target relative latency. This is required to be set for multi-chip alignment, or deterministic latency use-cases. It is not required to be set for single-chip alignment.
- **int Offset[4]**: Status – indicates the value the interface data was delayed by to achieve alignment, per tile.
- **int Latency[4]**: Status – indicates the measured relative-latency value of each tile.
- **int Marker_Delay**: Marker delay is for internal use.
- **int SysRef_Enable**: Set to 1 (default) to keep SYSREF capture enabled after MTS runs. Set to 0 to disable SYSREF capture.

struct XRFdc_MTS_Marker

This structure is for internal driver use. There are no user-configurable fields.

u32 Count[4];
u32 Loc[4];
struct XRFdc_MTS_DTC_Settings

This structure is for internal driver use. There are no user-configurable fields.

u32 RefTile;
u32 IsPLL;
int Target[4];
int Scan_Mode;
int DTC_Code[4];
int Num_Windows[4];
int Max_Gap[4];
int Min_Gap[4];
int Max_Overlap[4];

User API Functions

All user API functions are implemented in the source file xrfdc.c. The prototypes for these are provided in header file xrfdc.h.

All driver API functions have InstancePtr as an argument. It is a pointer to the XRFdc structure that contains information about the base address, the structure pointing to RFdc default configurations, and structures for RF-ADC and RF-DAC tile configurations. The InstancePtr argument is initialized in the XRFdc_CfgInitialize API function. Call the XRFdc_CfgInitialize API function to initialize InstancePtr before calling any other functions.

XRFdc_CfgInitialize

Function Prototype

int XRFdc_CfgInitialize (XRFdc* InstancePtr, XRFdc_Config *Config);

Arguments

- XRFdc* InstancePtr: Pointer to the driver instance.
- XRFdc_Config *Config: Pointer to the configuration structure.

Description

This API function populates appropriate entries in the driver instance by copying relevant entries from the configuration structure. This function is required for any software interaction with the RFdc driver API and must be called first before using any other API functions.
Return Value

XRFDC_SUCCESS
XRFDC_FAILURE

XRFdc.LookupConfig

Function Prototype

XRFdc_Config *XRFdc.LookupConfig(u16 DeviceId);

Arguments

- u16 DeviceId: ID of the device whose configuration information is required.

Description

This API function looks up the device configuration based on the unique ID of the device. A table contains the configuration information for each device in the system.

Return Value

The function returns a pointer to the configuration found, or NULL if the specified device ID was not found.

XRFdc.StartUp

Function Prototype

int XRFdc.StartUp (XRFdc* InstancePtr, u32 Type, int Tile_Id);

Arguments

- XRFdc* InstancePtr: Pointer to the driver instance.
- u32 Type: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- int Tile_Id: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.

Description

This API function restarts the tile as requested through Tile_Id. If -1 is passed as Tile_Id, the function restarts all the enabled tiles. Existing register settings are not lost or altered in the process.
Return Value

XRFDC_SUCCESS

XRFDC_FAILURE (returned when the tile is not enabled or available)

XRFdc_Shutdown

Function Prototype

```c
int XRFdc_Shutdown (XRFdc* InstancePtr, u32 Type, int Tile_Id);
```

Arguments

- XRFdc* InstancePtr: Pointer to the driver instance.
- u32 Type: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- int Tile_Id: Valid values are 0-3 and -1 (for all tiles).

Description

This API function stops the tile as requested through Tile_Id. If -1 is passed as Tile_Id, the function stops all the enabled tiles. The existing register settings are not cleared.

Note: This is a common API function for RF-ADC and RF-DAC tiles.

Return Value

XRFDC_SUCCESS

XRFDC_FAILURE (returned when the tile is not enabled or available)

XRFdc_Reset

Function Prototype

```c
int XRFdc_Reset (XRFdc* InstancePtr, u32 Type, int Tile_Id);
```

Arguments

- XRFdc* InstancePtr: Pointer to the driver instance.
- u32 Type: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- int Tile_Id: Valid values are 0-3 and -1 (for all tiles).
Description

This API function resets the tile as requested through Tile_Id. If -1 is passed as Tile_Id, it resets all the enabled tiles. All existing register settings are cleared and are replaced with the settings initially configured.

Note: This is a common API function for RF-ADC and RF-DAC tiles.

Return Value

XRFDC_SUCCESS

XRFDC_FAILURE (returned when the tile is not enabled or available)

XRFdc_ReStartIPSM

Function Prototype

```c
static void XRFdc_ReStartIPSM (XRFdc* InstancePtr, u32 BaseAddress, u32 Start, u32 End);
```

Arguments

- `XRFdc* InstancePtr`: Pointer to the driver instance.
- `u32 BaseAddress`: Tile control and status register base address.
- `u32 Start`: Start state of state machine.
- `int End`: End state of state machine.

Description

This API function restarts the requested tile and ensures that it starts from a defined start state and reaches the requested or defined end state. This function is for internal driver use.

Note: This a common API function for both RF-ADC and RF-DAC tiles.

Return Value

None.
**XRFdc_GetIPStatus**

**Function Prototype**

```c
int XRFdc_GetIPStatus (XRFdc* InstancePtr, XRFdc_IPStatus* IPStatus);
```

**Arguments**

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **XRFdc_IPStatus* IPStatus**: Pointer to the XRFdc_IPStatus structure through which the status is returned.

**Description**

This API function returns the IP status.

**Return Value**

- XRFDC_SUCCESS
- XRFDC_FAILURE (IP not ready)

**XRFdc_GetBlockStatus**

**Function Prototype**

```c
int XRFdc_GetBlockStatus (XRFdc* InstancePtr, u32 Type, int Tile_Id, u32 Block_Id,
                         XRFdc_BlockStatus* BlockStatus);
```

**Arguments**

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **u32 Type**: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- **int Tile_Id**: RF-ADC/RF-DAC tile number. Valid values are 0-3.
- **u32 Block_Id**: RF-ADC/RF-DAC block number inside the tile. Valid values are 0-3.
- **XRFdc_BlockStatus * BlockStatus**: Pointer to the XRFdc_BlockStatus structure through which the RF-ADC/RF-DAC block status is returned.

**Description**

This API function returns the requested block status.
**Note:** This is a common API function for RF-ADCs/RF-DACs.

**Return Value**

XRFDC_SUCCESS

XRFDC_FAILURE (Converter not enabled)

**XRFdc_SetMixerSettings**

**Function Prototype**

```c
int XRFdc_SetMixerSettings (XRFdc* InstancePtr, u32 Type, int Tile_Id, u32 Block_Id, 
XRFdc_Mixer_Settings * Mixer_Settings):
```

**Arguments**

- `XRFdc* InstancePtr`: Pointer to the driver instance.
- `u32 Type`: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- `int Tile_Id`: RF-ADC/RF-DAC tile number. Valid values are 0-3.
- `u32 Block_Id`: RF-ADC/RF-DAC number inside the tile. Valid values are 0-3.
- `XRFdc_Mixer_Settings * Mixer_Settings`: Pointer to the `XRFdc_Mixer_Settings` structure that passes the mixer/NCO settings.

**Description**

Mixer/NCO settings passed are used to update the corresponding block level registers. The driver structure is updated with the new values.

**Note:** This is a common API function for RF-ADCs/RF-DACs.

**Return Value**

XRFDC_SUCCESS

XRFDC_FAILURE
**XRFdc_GetMixerSettings**

**Function Prototype**

```c
int XRFdc_GetMixerSettings (XRFdc* InstancePtr, u32 Type, int Tile_Id, u32 Block_Id,
XRFdc_Mixer_Settings * Mixer_Settings);
```

**Arguments**

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **u32 Type**: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- **int Tile_Id**: RF-ADC/RF-DAC tile number. Valid values are 0-3.
- **u32 Block_Id**: RF-ADC/RF-DAC number inside the tile. Valid values are 0-3.
- **XRFdc_Mixer_Settings * Mixer_Settings**: Pointer to the `XRFdc_Mixer_Settings` structure in which the existing mixer/NCO settings are returned.

**Description**

Mixer/NCO settings from corresponding registers are returned to the caller.

*Note*: This is a common API function for RF-ADCs/RF-DACs.

**Return Value**

- `XRFDC_SUCCESS`
- `XRFDC_FAILURE`

---

**XRFdc_SetQMCSettings**

**Function Prototype**

```c
int XRFdc_SetQMCSettings (XRFdc* InstancePtr, u32 Type, int Tile_Id, u32 Block_Id,
XRFdc_QMC_Settings * QMC_Settings);
```

**Arguments**

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **u32 Type**: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- **int Tile_Id**: RF-ADC/RF-DAC tile number. Valid values are 0-3.
• u32 Block_Id: RF-ADC/RF-DAC number inside the tile. Valid values are 0-3.
• XRFdc_QMC_Settings * QMC_Settings: Pointer to the XRFdc_QMC_Settings structure used to update the corresponding registers.

Description
QMC settings from the passed structure are populated into the corresponding registers. The driver structure is updated with the new values.

Note: This is a common API function for RF-ADCs/RF-DACs.

Return Value
XRFDC_SUCCESS
XRFDC_FAILURE

Related Information
Threshold RFdc Driver API Commands

XRFdc_GetQMCSettings

Function Prototype

```c
int XRFdc_GetQMCSettings (XRFdc* InstancePtr, u32 Type, int Tile_Id, u32 Block_Id, XRFdc_QMC_Settings * QMC_Settings);
```

Arguments
• XRFdc* InstancePtr: Pointer to the driver instance.
• u32 Type: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
• int Tile_Id: RF-ADC/RF-DAC tile number. Valid values are 0-3.
• u32 Block_Id: RF-ADC/RF-DAC number inside the tile. Valid values are 0-3.
• XRFdc_QMC_Settings * QMC_Settings: Pointer to the XRFdc_QMC_Settings structure that returns the corresponding register values.

Description
QMC settings from the relevant registers are returned back to the caller.

Note: This is a common API function for RF-ADCs/RF-DACs.
Return Value

XRFDC_SUCCESS
XRFDC_FAILURE

XRFdc_GetCoarseDelaySettings

Function Prototype

```c
int XRFdc_GetCoarseDelaySettings (XRFdc* InstancePtr, u32 Type, int Tile_Id, u32 Block_Id, XRFdc_CoarseDelay_Settings * CoarseDelay_Settings);
```

Arguments

- **XRFdc* InstancePtr:** Pointer to the driver instance.
- **u32 Type:** RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- **int Tile_Id:** RF-ADC/RF-DAC tile number. Valid values are 0-3.
- **u32 Block_Id:** RF-ADC/RF-DAC number inside the tile. Valid values are 0-3.
- **XRFdc_CoarseDelay_Settings * CoarseDelay_Settings:** Pointer to the `XRFdc_CoarseDelay_Settings` structure which returns the corresponding register values.

Description

Coarse delay settings from the relevant registers are returned back to the caller.

*Note:* This is a common API function for RF-ADCs/RF-DACs.

Return Value

XRFDC_SUCCESS
XRFDC_FAILURE

XRFdc_SetCoarseDelaySettings

Function Prototype

```c
int XRFdc_SetCoarseDelaySettings (XRFdc* InstancePtr, u32 Type, int Tile_Id, u32 Block_Id, XRFdc_CoarseDelay_Settings * CoarseDelay_Settings);
```
Arguments

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **u32 Type**: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- **int Tile_Id**: RF-ADC/RF-DAC tile number. Valid values are 0-3.
- **u32 Block_Id**: RF-ADC/RF-DAC number inside the tile. Valid values are 0-3.
- **XRFdc_CoarseDelay_Settings* CoarseDelay_Settings**: Pointer to the XRFdc_CoarseDelay_Settings structure through which the coarse delay settings are passed.

Description

Coarse delay settings passed from the caller are updated in the relevant registers. The driver structure is updated with the new values.

*Note*: This is a common API function for RF-ADCs/RF-DACs.
Return Value

XRFDC_SUCCESS
XRFDC_FAILURE

XRFdc_UpdateEvent

Function Prototype

```c
int XRFdc_UpdateEvent (XRFdc* InstancePtr, u32 Type, int Tile_Id, u32 Block_Id, u32 Event);
```

Arguments

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **u32 Type**: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- **int Tile_Id**: RF-ADC/RF-DAC tile number. Valid values are 0-3.
- **u32 Block_Id**: RF-ADC/RF-DAC number inside the tile. Valid values are 0-3.
- **u32 Event**: Event can be mixer, QMC, or coarse delay.

Description

Use this function to trigger the update event for an event, if the event source is Slice or Tile.

*Note*: This is a common API function for RF-ADCs/RF-DACs.

Return Value

XRFDC_SUCCESS
XRFDC_FAILURE

XRFdc_SetInterpolationFactor

Function Prototype

```c
int XRFdc_SetInterpolationFactor (XRFdc* InstancePtr, int Tile_Id, u32 Block_Id, u32 InterpolationFactor);
```

Arguments

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- `int Tile_Id`: RF-DAC tile number. Valid values are 0-3.
- `u32 Block_Id`: RF-DAC number. Valid values are 0-3.
- `u32 InterpolationFactor`: Interpolation factor to be set for the RF-DAC.

**Description**

This API function sets the interpolation factor for the requested RF-DAC and also updates the FIFO read width based on the interpolation factor.

**Return Value**

- `XRFDC_SUCCESS`
- `XRFDC_FAILURE`

**XRFdc_GetInterpolationFactor**

**Function Prototype**

```c
int XRFdc_GetInterpolationFactor (XRFdc* InstancePtr, int Tile_Id, u32 Block_Id, u32 * InterpolationFactor);
```

**Arguments**

- `XRFdc* InstancePtr`: Pointer to the driver instance.
- `int Tile_Id`: RF-DAC tile number. Valid values are 0-3.
- `u32 Block_Id`: RF-DAC number inside the tile. Valid values are 0-3.
- `u32 * InterpolationFactor`: Pointer to return the interpolation factor for the RF-DAC.

**Description**

The interpolation factor for the requested RF-DAC is returned back to the caller.

*Note*: This API function is only applicable for RF-DACs.

**Return Value**

- `XRFDC_SUCCESS`
- `XRFDC_FAILURE`
XRFdc_SetDecimationFactor

Function Prototype

```c
int XRFdc_SetDecimationFactor (XRFdc* InstancePtr, int Tile_Id, u32 Block_Id, u32 DecimationFactor);
```

Arguments

- `XRFdc* InstancePtr`: Pointer to the driver instance.
- `int Tile_Id`: RF-ADC tile number. Valid values are 0-3.
- `u32 Block_Id`: RF-ADC number. Valid values are 0-3.
- `u32 DecimationFactor`: Decimation factor to be set for the RF-ADC.

Description

This API function sets the decimation factor for the requested RF-ADC and also updates the FIFO write width based on the decimation factor.

*Note*: This function is only applicable for RF-ADCs.

Return Value

- `XRFDC_SUCCESS`
- `XRFDC_FAILURE`

XRFdc_GetDecimationFactor

Function Prototype

```c
int XRFdc_GetDecimationFactor (XRFdc* InstancePtr, int Tile_Id, u32 Block_Id, u32 * DecimationFactor);
```

Arguments

- `XRFdc* InstancePtr`: Pointer to the driver instance.
- `int Tile_Id`: RF-ADC tile number. Valid values are 0-3.
- `u32 Block_Id`: RF-ADC number inside the tile. Valid values are 0-3.
- `u32 * DecimationFactor`: Pointer to return the decimation factor for the RF-ADC.
Description

Decimation factor for the requested RF-ADC is returned back to the caller.

Note: This API function is only applicable for RF-ADCs.

Return Value

XRFDC_SUCCESS
XRFDC_FAILURE

XRFdc_SetFabClkOutDiv

Function Prototype

```
u32 XRFdc_SetFabClkOutDiv(XRFdc *InstancePtr, u32 Type, int Tile_Id, u16 FabClkDiv);
```

Arguments

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **u32 Type**: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- **int Tile_Id**: RF-ADC/RF-DAC tile number. Valid values are 0-3.
- **u16 FabClkDiv**: Fabric clock divider to be set for a tile.

Description

Use this function to set the divider for clock fabric out.

Note: This is a common function for RF-ADCs and RF-DACs.

Return Value

XRFDC_SUCCESS
XRFDC_FAILURE

XRFdc_SetFabWrVldWords

Function Prototype

```
int XRFdc_SetFabWrVldWords (XRFdc* InstancePtr, int Tile_Id, u32 Block_Id, u32 FabricWrVldWords);
```
Arguments

- `XRFdc* InstancePtr`: Pointer to the driver instance.
- `int Tile_Id`: RF-DAC tile number. Valid values are 0-3.
- `u32 Block_Id`: RF-DAC number inside the tile. Valid values are 0-3.
- `u32 FabricWrVldWords`: Write fabric data rate to be set for RF-DAC.

Description

This API function sets the write fabric data rate for the requested RF-DAC by writing to the corresponding register.

*Note:* This API function is only applicable for RF-DACs.

Return Value

- `XRFDC_SUCCESS`
- `XRFDC_FAILURE`

**XRFdc_GetFabWrVldWords**

Function Prototype

```c
int XRFdc_GetFabWrVldWords (XRFdc* InstancePtr, u32 Type, int Tile_Id, u32 Block_Id, u32 * FabricWrVldWords);
```

Arguments

- `XRFdc* InstancePtr`: Pointer to the driver instance.
- `u32 Type`: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- `int Tile_Id`: RF-ADC/RF-DAC tile number. Valid values are 0-3.
- `u32 Block_Id`: RF-ADC/RF-DAC number inside the tile. Valid values are 0-3.
- `u32 * FabricWrVldWords`: Pointer to return the write fabric data rate for the RF-ADCs/RF-DACs.

Description

Write fabric data rate for the requested RF-ADC/RF-DAC is returned back to the caller.

*Note:* This is a common API function for RF-ADCs and RF-DACs.
Return Value

XRFDC_SUCCESS
XRFDC_FAILURE

XRFdc_SetFabRdVldWords

Function Prototype

```c
int XRFdc_SetFabRdVldWords (XRFdc* InstancePtr, int Tile_Id, u32 Block_Id, u32 FabricRdVldWords);
```

Arguments

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **int Tile_Id**: RF-ADC tile number. Valid values are 0-3.
- **u32 Block_Id**: RF-ADC number inside the tile. Valid values are 0-3.
- **u32 FabricRdVldWords**: Read fabric data rate to be set for RF-ADC.

Description

This API function sets the read fabric data rate for the requested RF-ADC by writing to the corresponding register.

Note: This API function is only applicable for RF-ADCs.

Return Value

XRFDC_SUCCESS
XRFDC_FAILURE

XRFdc_GetFabRdVldWords

Function Prototype

```c
int XRFdc_GetFabRdVldWords (XRFdc* InstancePtr, u32 Type, int Tile_Id, u32 Block_Id, u32 * FabricRdVldWords);
```
Arguments

- XRFdc* InstancePtr: Pointer to the driver instance.
- u32 Type: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- int Tile_Id: RF-ADC/RF-DAC tile number. Valid values are 0-3.
- u32 Block_Id: RF-ADC/RF-DAC number inside the tile. Valid values are 0-3.
- u32 * FabricRdVldWords: Pointer to return the read fabric data rate for the RF-ADC/RF-DACs.

Description

Read fabric data rate for the requested RF-ADC/RF-DAC is returned back to the caller.

Note: This is a common API function for RF-ADCs and RF-DACs.

Return Value

XRFDC_SUCCESS
XRFDC_FAILURE

XRFdc_ThresholdStickyClear

Function Prototype

```c
int XRFdc_ThresholdStickyClear (XRFdc* InstancePtr, int Tile_Id, u32 Block_Id, u32 ThresholdToUpdate);
```

Arguments

- XRFdc* InstancePtr: Pointer to the driver instance.
- int Tile_Id: RF-ADC tile number. Valid values are 0-3.
- u32 Block_Id: RF-ADC number inside the tile. Valid values are 0-3.
- u32 ThresholdToUpdate: Select which Threshold (Threshold0 or Threshold1 or both) to update.

Description

This API function clears the sticky bit in threshold configuration registers based on the ThresholdToUpdate parameter.

Note: This API function is applicable only for RF-ADCs.
Return Value

XRFDC_SUCCESS
XRFDC_FAILURE

Related Information

Threshold RFdc Driver API Commands

XRFdc_SetThresholdClrMode

Function Prototype

```c
int XRFdc_SetThresholdClrMode (XRFdc* InstancePtr, int Tile_Id, u32 Block_Id, u32 ThresholdToUpdate, u32 ClrMode);
```

Arguments

- **XRFdc* InstancePtr**: Pointer to the driver instance
- **int Tile_Id**: RF-ADC tile number. Valid values are 0-3.
- **u32 Block_Id**: RF-ADC number inside the tile. Valid values are 0-3.
- **u32 ThresholdToUpdate**: Select which Threshold (Threshold0 or Threshold1 or both) to update.
- **u32 ClrMode**: Clear mode can be DRP access (manual) or auto clear (QMC gain update event).

Description

This API function sets the threshold clear mode.

*Note*: This API function is only applicable for RF-ADCs.

Return Value

XRFDC_SUCCESS
XRFDC_FAILURE

Related Information

Threshold RFdc Driver API Commands
XRFdc_GetThresholdSettings

Function Prototype

```c
int XRFdc_GetThresholdSettings (XRFdc* InstancePtr, int Tile_Id, u32 Block_Id,
                                   XRFdc_Threshold_Settings * Threshold_Settings);
```

Arguments

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **int Tile_Id**: RF-ADC tile number. Valid values are 0-3.
- **u32 Block_Id**: RF-ADC number inside the tile. Valid values are 0-3.
- **XRFdc_Threshold_Settings * Threshold_Settings**: Pointer through which the register settings for thresholds are passed back.

Description

This API function reads threshold settings from the corresponding registers.

*Note*: This API function is only applicable for RF-ADCs.

Return Value

- XRFDC_SUCCESS
- XRFDC_FAILURE

XRFdc_SetThresholdSettings

Function Prototype

```c
int XRFdc_SetThresholdSettings (XRFdc* InstancePtr, int Tile_Id, u32 Block_Id,
                                   XRFdc_Threshold_Settings * Threshold_Settings);
```

Arguments

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **int Tile_Id**: RF-ADC tile number. Valid values are 0-3.
- **u32 Block_Id**: RF-ADC number inside the tile. Valid values are 0-3.
- **XRFdc_Threshold_Settings**: 
  - **Threshold_Settings**: Pointer through which the register settings for thresholds are passed to the API.

**Description**

This API function writes the threshold settings to the relevant registers. The driver structure is updated with the new values.

**Note**: This API function is only applicable for RF-ADCs.

**Return Value**

- XRFDC_SUCCESS
- XRFDC_FAILURE

**Related Information**

Threshold RFdc Driver API Commands

**XRFdc_SetDecoderMode**

**Function Prototype**

```c
int XRFdc_SetDecoderMode (XRFdc* InstancePtr, int Tile_Id, u32 Block_Id, u32 DecoderMode);
```

**Arguments**

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **int Tile_Id**: RF-DAC tile number. Valid values are 0-3.
- **u32 Block_Id**: RF-DAC number inside the tile. Valid values are 0-3.
- **u32 DecoderMode**: Decoder mode settings. Valid values are:
  - Maximum linearity, for randomized decoder
  - Maximum SNR, for non-randomized decoder

**Description**

This API function writes the decoder mode to the relevant registers. The driver structure is updated with the new values.

**Note**: This API function is used only for the RF-DACs.
Return Value

XRFDC_SUCCESS
XRFDC_FAILURE

XRFdc_GetDecoderMode

Function Prototype

```c
int XRFdc_GetDecoderMode (XRFdc* InstancePtr, int Tile_Id, u32 Block_Id, u32 *DecoderMode);
```

Arguments

- `XRFdc* InstancePtr`: Pointer to the driver instance.
- `int Tile_Id`: RF-DAC tile number. Valid values are 0-3.
- `u32 Block_Id`: RF-DAC number inside the tile. Valid values are 0-3.
- `u32 DecoderMode`: Pointer in which decoder mode settings to be returned back to the caller. Valid values are:
  - Maximum linearity, for randomized decoder
  - Maximum SNR, for non-randomized decoder

Description

This API function reads the decoder mode from the relevant registers.

Note: This API function is used only for the RF-DACs.

Return Value

XRFDC_SUCCESS
XRFDC_FAILURE

XRFdc_ResetNCOPhase

Function Prototype

```c
int XRFdc_ResetNCOPhase (XRFdc* InstancePtr, u32 Type, int Tile_Id, u32 Block_Id);
```
Arguments

- **XRFdc** InstancePtr: Pointer to the driver instance.
- **u32 Type**: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- **int Tile_Id**: RF-ADC/RF-DAC tile number. Valid values are 0-3.
- **u32 Block_Id**: RF-ADC/RF-DAC number inside the tile. Valid values are 0-3.

Description

This API function resets the NCO phase of the current block phase accumulator.

Return Value

XRFDC_SUCCESS
XRFDC_FAILURE

XRFdc_SetupFIFO

Function Prototype

```c
int XRFdc_SetupFIFO(XRFdc* InstancePtr, u32 Type, int Tile_Id, u8 Enable);
```

Arguments

- **XRFdc** InstancePtr: Pointer to the driver instance.
- **u32 Type**: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- **int Tile_Id**: RF-ADC/RF-DAC tile number. Valid values are 0-3.
- **u8 Enable**: Valid values are 1 (FIFO enable) and 0 (FIFO Disable).

Description

This API function enables and disables the RF-ADC/RF-DAC FIFO.

Return Value

XRFDC_SUCCESS
XRFDC_FAILURE
**XRFdc_GetOutputCurr**

**Function Prototype**

```c
int XRFdc_GetOutputCurr (XRFdc* InstancePtr, int Tile_Id, u32 Block_Id, int *OutputCurr);
```

**Arguments**

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **int Tile_Id**: RF-DAC tile number. Valid values are 0-3.
- **u32 Block_Id**: RF-DAC number inside the tile. Valid values are 0-3.
- **int *OutputCurr**: OutputCurr pointer to return the output current.

**Description**

This API function gets the output current.

*Note*: This API function is used only for the RF-DACs.

**Return Value**

- XRFDC_SUCCESS
- XRFDC_FAILURE

**XRFdc_GetFIFOStatus**

**Function Prototype**

```c
int XRFdc_GetFIFOStatus(XRFdc* InstancePtr, u32 Type, int Tile_Id, u8 *Enable);
```

**Arguments**

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **u32 Type**: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- **int Tile_Id**: RF-ADC/RF-DAC tile number. Valid values are 0-3.
- **u8 *Enable**: Valid values are 1 (FIFO enable) and 0 (FIFO Disable).
**Description**

This API function gets the current status of the RF-ADC/RF-DAC FIFO.

**Return Value**

- XRFDC_SUCCESS
- XRFDC_FAILURE

**XRFdc_DumpRegs**

**Function Prototype**

```c
void XRFdc_DumpRegs (XRFdc* InstancePtr, u32 Type, int Tile_Id);
```

**Arguments**

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **u32 Type**: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- **int Tile_Id**: RF-ADC/RF-DAC tile number. Valid values are 0-3 and -1.

**Description**

This function is intended for debug purposes. It prints the contents of registers to the console for the passed Tile_Id. If -1 is passed, it prints the contents of the registers for all the tiles for the respective RF-ADC or RF-DAC. It prints the offset of the register as well as the content.

**Return Value**

None.

**XRFdc_SetNyquistZone**

**Function Prototype**

```c
int XRFdc_SetNyquistZone(XRFdc* InstancePtr, u32 Type, int Tile_Id, u32 Block_Id, u32 NyquistZone);
```

**Arguments**

- **XRFdc* InstancePtr**: Pointer to the driver instance.
• **u32 Type:** RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
• **int Tile_Id:** RF-ADC/RF-DAC tile number. Valid values are 0-3.
• **u32 Block_Id:** RF-DAC number inside the tile. Valid values are 0-3.
• **u32 NyquistZone:** Valid values are 1 (Odd), 2 (Even).

**Description**
This API function sets the Nyquist zone for the RF-ADC/RF-DACs.

**Return Value**
- XRFDC_SUCCESS
- XRFDC_FAILURE

**XRFdc_GetNyquistZone**

**Function Prototype**

```c
int XRFdc_GetNyquistZone(XRFdc* InstancePtr, u32 Type, int Tile_Id, u32 Block_Id, u32 *NyquistZone);
```

**Arguments**
• **XRFdc* InstancePtr:** Pointer to the driver instance.
• **u32 Type:** RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
• **int Tile_Id:** RF-ADC/RF-DAC tile number. Valid values are 0-3.
• **u32 Block_Id:** RF-DAC number inside the tile. Valid values are 0-3.
• **u32 *NyquistZone:** Pointer to return Nyquist Zone.

**Description**
This API function gets the Nyquist zone for the RF-ADCs/RF-DACs.

**Return Value**
- XRFDC_SUCCESS
- XRFDC_FAILURE
XRFdc_SetInvSincFIR

Function Prototype

```c
u32 XRFdc_SetInvSincFIR(XRFdc *InstancePtr, u32 Tile_Id, u32 Block_Id, u16 Enable);
```

Arguments

- `XRFdc* InstancePtr`: Pointer to the driver instance.
- `u32 Tile_Id`: RF-DAC tile number. Valid values are 0-3.
- `u32 Block_Id`: RF-DAC number. Valid values are 0-3.
- `u16 Enable`: Valid values are 0 (disable) and 1 (enable).

Description

This API function is used to enable or disable the inverse sinc filter.

Note: This API function is used only for the RF-DACs.

Return Value

- `XRFDC_SUCCESS`
- `XRFDC_FAILURE`

XRFdc_GetInvSincFIR

Function Prototype

```c
u32 XRFdc_SetInvSincFIR(XRFdc *InstancePtr, u32 Tile_Id, u32 Block_Id, u16 *Enable);
```

Arguments

- `XRFdc* InstancePtr`: Pointer to the driver instance.
- `u32 Tile_Id`: RF-DAC tile number. Valid values are 0-3.
- `u32 Block_Id`: RF-DAC tile number. Valid values are 0-3.
- `u16 *Enable`: Enable pointer is used to get the status. valid values are 0 (disable) and 1 (enable).
Description
This API function is used to get the inverse sinc filter status.

*Note:* This API function is used only for the RF-DACs.

Return Value
- XRFDC_SUCCESS
- XRFDC_FAILURE

**XRFdc_SetCalibrationMode**

Function Prototype

```
uint32_t XRFdc_SetCalibrationMode(XRFdc *InstancePtr, int Tile_Id, uint32_t Block_Id, uint8_t CalibrationMode);
```

Arguments
- **InstancePtr:** Pointer to the driver instance.
- **Tile_Id:** RF-ADC tile number. Valid values are 0-3.
- **Block_Id:** RF-ADC number. Valid values are 0-3.
- **CalibrationMode:** Valid values are 1 and 2.

Description
This API function sets the calibration mode for the RF-ADC.

*Note:* This API function is used only for the RF-ADCs.

Return Value
- XRFDC_SUCCESS
- XRFDC_FAILURE

**XRFdc_GetCalibrationMode**

Function Prototype

```
uint32_t XRFdc_GetCalibrationMode(XRFdc *InstancePtr, int Tile_Id, uint32_t Block_Id, uint8_t *CalibrationMode);
```

*Note:* This API function is used only for the RF-DACs.
Arguments

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **u32 Tile_Id**: RF-ADC tile number. Valid values are 0-3.
- **u32 Block_Id**: RF-ADC block number. Valid values are 0-3.
- **u8 *CalibrationMode**: Pointer to get the calibration mode.

Description

This API function sets the calibration mode for the RF-ADCs.

*Note*: This API function is used only for RF-ADCs.

Return Value

- **XRFDC_SUCCESS**
- **XRFDC_FAILURE**

**XRFdc_MultiBand**

Function Prototype

```c
u32 XRFdc_MultiBand(XRFdc* InstancePtr, u32 Type, u32 Tile_Id, u8 DigitalDataPathMask, u32 DataType, u32 DataConverterMask);
```

Arguments

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **u32 Type**: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- **u32 Tile_Id**: RF-ADC/RF-DAC tile number. Valid values are 0-3.
- **u8 DigitalDataPathMask**: Represents the datapath mask participating in multiband. The first four bits represent four datapaths; 1 means enabled and 0 means disabled.
- **u32 DataType**: DataType is the mixer data type; valid values are XRFDC_MB_DATATYPE_.*.
- **u32 DataConverterMask**: DataConverterMask is a block-enabled mask (input/output driving blocks); 1 means enabled and 0 means disabled.

Description

This API function is used to set up single-band and multi-band configuration.
Return Value

XRFDC_SUCCESS
XRFDC_FAILURE

XRFdc_MultiConverter_Init

Function Prototype

void XRFdc_MultiConverter_Init (XRFdc_MultiConverter_Sync_Config* Config, int *PLL_Codes, int *T1_Codes);

Arguments

• XRFdc_MultiConverter_Sync_Config* Config: Multi-tile synchronization configuration structure.
• int *PLL_Codes: Optional target codes for PLL analog SYSREF capture. Set to 0 (NULL) when not used.
• int *T1_Codes: Optional target codes for T1 analog SYSREF capture. Set to 0 (NULL) when not used.

Description

This API function initializes the multi-tile synchronization configuration structures; it must be called before XRFdc_MultiConverter_Sync. Optionally, it allows previously determined target codes to be provided for the PLL/T1 analog SYSREF capture.

Return Value

None.

XRFdc_MultiConverter_Sync

Function Prototype

u32 XRFdc_MultiConverter_Sync (XRFdc* InstancePtr, u32 Type, XRFdc_MultiConverter_Sync_Config* Config);

Arguments

• XRFdc* InstancePtr: Pointer to the driver instance.
• u32 Type: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
• XRFdc_MultiConverter_Sync_Config* Config: Multi-tile sync config structure.
Description
This is the top-level API function used for multi-tile synchronization.

Return Value
XRFDC_MTS_OK
XRFDC_MTS_TIMEOUT
XRFDC_MTS_MARKER_RUN
XRFDC_MTS_MARKER_MISM
XRFDC_MTS_NOT_SUPPORTED

XRFdc_DynamicPLLConfig

Function Prototype
```
u32 XRFdc_DynamicPLLConfig(XRFdc* InstancePtr, u32 Type, u32 Tile_Id, u8 Source, double RefClkFreq, double SamplingRate);
```

Arguments
- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **u32 Type**: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- **u32 Tile_Id**: RF-ADC/RF-DAC tile number. Valid values are 0-3.
- **u8 Source**: Internal PLL or external clock source.
- **double RefClkFreq**: Reference clock frequency in MHz (50–1200).
- **double SamplingRate**: Sampling rate frequency in MHz (500–6400).

Description
This API function is used for dynamic switching between the internal PLL and the external clock sources, and for configuring the internal PLL for the RF-ADCs/RF-DACs.

Return Value
XRFDC_SUCCESS
XRFDC_FAILURE
XRFdc_GetClockSource

Function Prototype

```c
u32 XRFdc_GetClockSource(XRFdc* InstancePtr, u32 Type, u32 Tile_Id, u32 *ClockSource);
```

Arguments

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **u32 Type**: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- **u32 Tile_Id**: RF-ADC/RF-DAC tile number. Valid values are 0-3.
- **u32 *ClockSource**: Pointer to return the clock source.

Description

This API function gets the clock source for the RF-ADCs/RF-DACs.

Return Value

**XRFDC_SUCCESS**

**XRFDC_FAILURE**

XRFdc_GetPLLLockStatus

Function Prototype

```c
u32 XRFdc_GetPLLLockStatus(XRFdc* InstancePtr, u32 Type, u32 Tile_Id, u32 *LockStatus);
```

Arguments

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **u32 Type**: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- **u32 Tile_Id**: RF-ADC/RF-DAC tile number. Valid values are 0-3.
- **u32 *LockStatus**: Pointer to return the PLL lock status.

Description

This API function gets the PLL lock status for the RF-ADCs/RF-DACs.
Return Value

XRFDC_SUCCESS
XRFDC_FAILURE

Interrupt Handling

All interrupt handling functions are implemented in the xrfdc_intr.c file. The prototypes for these are provided through xrfdc.h.

XRFdc_IntrEnable

Function Prototype

```c
void XRFdc_IntrEnable (XRFdc* InstancePtr, u32 Type, int Tile_Id, u32 Block_Id, u32 IntrMask);
```

Arguments

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **u32 Type**: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- **int Tile_Id**: RF-ADC or RF-DAC tile number. Valid values are 0-3.
- **u32 Block_Id**: RF-ADC or RF-DAC number inside the tile. Valid values are 0-3.
- **u32 IntrMask**: Interrupts to be enabled. The valid masks are:
  - XRFDC_IXR_FIFOUSRDat_Mask 0x0000000FU
  - XRFDC_IXR_FIFOUSRDat_OF_Mask 0x00000001U
  - XRFDC_IXR_FIFOUSRDat_UF_Mask 0x00000002U
  - XRFDC_IXR_FIFOMRGNIND_OF_Mask 0x00000004U
  - XRFDC_IXR_FIFOMRGNIND_UF_Mask 0x00000008U
  - XRFDC_ADC_IXR_DATAPATH_Mask 0x000000FF0U
  - XRFDC_ADC_IXR_DMON_STG_Mask 0x000003F0U
  - XRFDC_DAC_IXR_DATAPATH_Mask 0x00001FF0U
  - XRFDC_DAC_IXR_INTP_STG_Mask 0x000003F0U
  - XRFDC_DAC_IXR_INTP_I_STG0_Mask 0x00000010U
  - XRFDC_DAC_IXR_INTP_I_STG1_Mask 0x00000010U
```
• XRFDC_DAC_IXR_INTP_I_STG1_MASK 0x00000020U
• XRFDC_DAC_IXR_INTP_I_STG2_MASK 0x00000040U
• XRFDC_DAC_IXR_INTP_Q_STG0_MASK 0x00000080U
• XRFDC_DAC_IXR_INTP_Q_STG1_MASK 0x00000100U
• XRFDC_DAC_IXR_INTP_Q_STG2_MASK 0x00000200U
• XRFDC_ADC_IXR_DMON_I_STG0_MASK 0x00000010U
• XRFDC_ADC_IXR_DMON_I_STG1_MASK 0x00000020U
• XRFDC_ADC_IXR_DMON_I_STG2_MASK 0x00000040U
• XRFDC_ADC_IXR_DMON_Q_STG0_MASK 0x00000080U
• XRFDC_ADC_IXR_DMON_Q_STG1_MASK 0x00000100U
• XRFDC_ADC_IXR_DMON_Q_STG2_MASK 0x00000200U
• XRFDC_IXR_QMC_GAIN_PHASE_MASK 0x00000400U
• XRFDC_IXR_QMC_OFFST_MASK 0x00000800U
• XRFDC_DAC_IXR_INVSNC_OF_MASK 0x00001000U
• XRFDC_SUBADC_IXR_DCDR_MASK 0x00FF0000U
• XRFDC_SUBADC0_IXR_DCDR_OF_MASK 0x00010000U
• XRFDC_SUBADC0_IXR_DCDR_UF_MASK 0x00020000U
• XRFDC_SUBADC1_IXR_DCDR_OF_MASK 0x00040000U
• XRFDC_SUBADC1_IXR_DCDR_UF_MASK 0x00080000U
• XRFDC_SUBADC2_IXR_DCDR_OF_MASK 0x00100000U
• XRFDC_SUBADC2_IXR_DCDR_UF_MASK 0x00200000U
• XRFDC_SUBADC3_IXR_DCDR_OF_MASK 0x00400000U
• XRFDC_SUBADC3_IXR_DCDR_UF_MASK 0x00800000U
• XRFDC_ADC_OVR_VOLTAGE_MASK 0x04000000U
• XRFDC_ADC_OVR_RANGE_MASK 0x08000000U
• XRFDC_ADC_DAT_OVR_MASK 0x40000000U
• XRFDC_ADC_FIFO_OVR_MASK 0x80000000U

Description

This API function enables the interrupt for the corresponding converter by taking the IntrMask as an input and writing to the corresponding register bit.
Return Value
None.

**XRFdc_IntrDisable**

**Function Prototype**

```c
void XRFdc_IntrDisable (XRFdc* InstancePtr, u32 Type, int Tile_Id, u32 Block_Id, u32 IntrMask);
```

**Arguments**

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **u32 Type**: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- **int Tile_Id**: RF-ADC or RF-DAC tile number. Valid values are 0-3.
- **u32 Block_Id**: RF-ADC or RF-DAC number inside the tile. Valid values are 0-3.
- **u32 IntrMask**: Interrupts to be disabled. The valid masks are described in the API for **XRFdc_IntrEnable**.

**Description**

This function disables the interrupt for the corresponding converter by taking the **IntrMask** as an input and writing to the corresponding register bit.

**Return Value**
None.

**Related Information**

*XRFdc_IntrEnable*

**XRFdc_SetStatusHandler**

**Function Prototype**

```c
void XRFdc_SetStatusHandler (XRFdc *InstancePtr, void *CallBackRef, XRFdc_StatusHandler FunctionPtr);
```
Arguments

- `XRFdc* InstancePtr`: Pointer to the driver instance.
- `void *CallBackRef`: Upper layer callback reference passed back when the callback function is invoked.
- `XRFdc_StatusHandler FunctionPtr`: Pointer to the callback function.

Description

This function sets the status callback function, the status handler, which the driver calls when it encounters conditions that must be reported to the higher layer software. The handler executes in an interrupt context to minimize the amount of processing.

Return Value

None.

**XRFdc_IntrClr**

Function Prototype

```
void XRFdc_IntrClr(XRFdc* InstancePtr, u32 Type, int Tile_Id, u32 Block_Id, u32 IntrMask);
```

Arguments

- `XRFdc* InstancePtr`: Pointer to the driver instance.
- `u32 Type`: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- `int Tile_Id`: RF-ADC or RF-DAC tile number. Valid values are 0-3.
- `u32 Block_Id`: RF-ADC or RF-DAC number inside the tile. Valid values are 0-3
- `u32 IntrMask`: Interrupts to be cleared. The valid masks are described in the API for `XRFdc_IntrEnable`.

Description

This function clears the interrupt for the corresponding converter by taking the `IntrMask` as an input and writing to the corresponding register bit.

Return Value

None.
**XRFdc_GetIntrStatus**

**Function Prototype**

```c
u32 XRFdc_GetIntrStatus (XRFdc* InstancePtr, u32 Type, int Tile_Id, u32 Block_Id);
```

**Arguments**

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **u32 Type**: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- **int Tile_Id**: RF-ADC or RF-DAC tile number. Valid values are 0-3.
- **u32 Block_Id**: RF-ADC or RF-DAC number inside the tile. Valid values are 0-3.

**Description**

This function returns the status of the interrupts through the masks. Valid masks are described in the API for `XRFdc_IntrEnable`.

**Return Value**

The interrupt status through the masks.

**Related Information**

- `XRFdc_IntrEnable`

---

**XRFdc_IntrHandler**

**Function Prototype**

```c
int XRFdc_IntrHandler (int Vector, void * XRFdcPtr);
```

**Arguments**

- **Int Vector**: Interrupt Vector number.
- **void* XRFdcPtr**: Pointer to the driver instance that caused the interrupt.

**Description**

This function is used internally by the driver. This is the master interrupt handler for the RFSoC driver. This routine must be connected to an interrupt controller using OS/BSP-specific APIs. It clears the source of the interrupt and prints the cause of the interrupt.
Return Value

METAL_IRQ_HANDLED (to inform Libmetal library, IRQ is handled).

In-line Functions

All in-line functions are defined in the `xrfdc.h` file. These are available to programmers and are also used internally by other driver API functions. These in-line functions provide a set of utility APIs.

**XRFdc_Get_IPBaseAddr**

Function Prototype

```c
static inline u32 XRFdc_Get_IPBaseAddr (XRFdc* InstancePtr);
```

Arguments

- **XRFdc* InstancePtr**: Pointer to the driver instance.

Description

None.

Return Value

Base address of the IP.

**XRFdc_Get_TileBaseAddr**

Function Prototype

```c
static inline u32 XRFdc_Get_TileBaseAddr (XRFdc* InstancePtr, u32 Type, int Tile_Id);
```

Arguments

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **u32 Type**: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- **int Tile_Id**: Valid values, 0-3.
**Description**
None.

**Return Value**
Base address of the requested tile.

**XRFdc_Get_BlockBaseAddr**

**Function Prototype**

```c
static inline u32 XRFdc_Get_BlockBaseAddr (XRFdc* InstancePtr, u32 Type, int Tile_Id, u32 Block_Id);
```

**Arguments**

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **u32 Type**: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- **int Tile_Id**: Valid values are 0-3.
- **int Block_Id**: Valid values are 0-3.

**Description**
None.

**Return Value**
Base address of the requested converter.

**XRFdc_GetNoOfDACBlock**

**Function Prototype**

```c
static inline u32 XRFdc_GetNoOfDACBlock (XRFdc* InstancePtr, int Tile_Id);
```

**Arguments**

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **int Tile_Id**: Valid values are 0-3.
Description
None.

Return Value
This function returns the number of RF-DACs enabled in the tile.

XRFdc_GetNoOfADCBlocks

Function Prototype

```c
static inline u32 XRFdc_GetNoOfADCBlocks (XRFdc* InstancePtr, int Tile_Id);
```

Arguments
- `XRFdc* InstancePtr`: Pointer to the driver instance.
- `int Tile_Id`: Valid values are 0-3.

Description
None.

Return Value
This function returns the number of RF-ADCs enabled in the tile.

XRFdc_IsDACBlockEnabled

Function Prototype

```c
static inline u32 XRFdc_IsDACBlockEnabled (XRFdc* InstancePtr, int Tile_Id, u32 Block_Id);
```

Arguments
- `XRFdc* InstancePtr`: Pointer to the driver instance.
- `int Tile_Id`: Valid values are 0-3.
- `u32 Block_Id`: Valid values are 0-3.
Description
None.

Return Value
If the requested RF-DAC is enabled, the function returns 1; otherwise, it returns 0.

**XRFdc_IsADCBlockEnabled**

Function Prototype

```c
static inline u32 XRFdc_IsADCBlockEnabled (XRFdc* InstancePtr, int Tile_Id, u32 Block_Id);
```

Arguments

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **int Tile_Id**: Valid values are 0-3.
- **u32 Block_Id**: Valid values are 0-3.

Description
None.

Return Value
If the requested RF-ADC is enabled, the function returns 1; otherwise, it returns 0.

**XRFdc_IsADC4GSPS**

Function Prototype

```c
static inline u32 XRFdc_IsADC4GSPS (XRFdc* InstancePtr);
```

Arguments

- **XRFdc* InstancePtr**: Pointer to the driver instance.

Description
None.
Return Value

If the RF-ADC part is 4 GSPS, the function returns 1; otherwise, it returns 0.

**XRFdc_GetDataType**

**Function Prototype**

```c
static inline u32 XRFdc_GetDataType (XRFdc* InstancePtr, u32 Type, int Tile_Id, u32 Block_Id);
```

**Arguments**

- XRFdc* InstancePtr: Pointer to the driver instance.
- u32 Type: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- int Tile_Id: Valid values are 0-3.
- u32 Block_Id: Valid values are 0-3.

**Description**

None.

**Return Value**

If the data type is real, the function returns 0; otherwise, it returns 1.

**XRFdc_GetDataWidth**

**Function Prototype**

```c
static inline u32 XRFdc_GetDataWidth (XRFdc* InstancePtr, u32 Type, int Tile_Id, u32 Block_Id);
```

**Arguments**

- XRFdc* InstancePtr: Pointer to the driver instance.
- u32 Type: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- int Tile_Id: Valid values are 0-3.
- u32 Block_Id: Valid values are 0-3.
Description
None.

Return Value
This function returns the programmed data width for the RF-ADC.

**XRFdc_GetInverseSincFilter**

Function Prototype

```c
static inline u32 XRFdc_GetInverseSincFilter (XRFdc* InstancePtr, int Tile_Id, u32 Block_Id);
```

Arguments
- `XRFdc* InstancePtr`: Pointer to the driver instance.
- `int Tile_Id`: Valid values are 0-3.
- `u32 Block_Id`: Valid values are 0-3.

Description
None.

Return Value
If the inverse sinc filter is enabled for the RF-DAC, the function returns 1; otherwise, it returns 0.

**XRFdc_GetMixedMode**

Function Prototype

```c
static inline u32 XRFdc_GetMixedMode (XRFdc* InstancePtr, int Tile_Id, u32 Block_Id);
```

Arguments
- `XRFdc* InstancePtr`: Pointer to the driver instance.
- `int Tile_Id`: Valid values are 0-3.
- `u32 Block_Id`: Valid values are 0-3.
Description
None.

Return Value
This function returns the mixed mode setting for the RF-DAC.

XRFdc_GetMasterTile

Function Prototype

static inline u32 XRFdc_GetMasterTile (XRFdc* InstancePtr, u32 Type);

Arguments

- XRFdc* InstancePtr: Pointer to the driver instance.
- u32 Type: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.

Description
None.

Return Value
The function returns the master tile ID.

XRFdc_GetSysRefSource

Function Prototype

static inline u32 XRFdc_GetSysRefSource (XRFdc* InstancePtr, u32 Type);

Arguments

- XRFdc* InstancePtr: Pointer to the driver instance.
- u32 Type: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.

Description
None.
Return Value

The function returns the source of the SYSREF (internal or external).

XRFdc_GetFabClkFreq

Function Prototype

```c
static inline double XRFdc_GetFabClkFreq (XRFdc* InstancePtr, u32 Type, int Tile_Id);
```

Arguments

- XRFdc* InstancePtr: Pointer to the driver instance.
- u32 Type: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- int Tile_Id: Valid values are 0-3.

Description

None.

Return Value

The function returns the programmed fabric clock frequency.

XRFdc_IsFifoEnabled

Function Prototype

```c
static inline u32 XRFdc_IsFifoEnabled (XRFdc* InstancePtr, u32 Type, int Tile_Id, u32 Block_Id);
```

Arguments

- XRFdc* InstancePtr: Pointer to the driver instance.
- u32 Type: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- int Tile_Id: Valid values are 0-3.
- u32 Block_Id: Valid values are 0-3.
Description
None.

Return Value
If the FIFO is enabled, the function returns 1; otherwise, it returns 0.

XRFdc_GetDriverVersion

Function Prototype

```c
static inline double XRFdc_GetDriverVersion();
```

Arguments
None.

Description
Gets the driver version.

Return Value
Driver version number.

XRFdc_GetConnectedIData

Function Prototype

```c
static inline int XRFdc_GetConnectedIData(XRFdc* InstancePtr, u32 Type, int Tile_Id, u32 Block_Id);
```

Arguments
- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **u32 Type**: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- **int Tile_Id**: RF-ADC/RF-DAC tile number. Valid values are 0-3.
- **u32 Block_Id**: RF-ADC/RF-DAC number. Valid values are 0-3.

Description
Get converter connected for I digital data path.
Return Value
Converter number.

XRFdc_GetConnectedQData

Function Prototype

```c
static inline int XRFdc_GetConnectedQData(XRFdc* InstancePtr, u32 Type, int Tile_Id, u32 Block_Id);
```

Arguments

- **XRFdc* InstancePtr**: Pointer to the driver instance.
- **u32 Type**: RF-ADC or RF-DAC; 0 for RF-ADC and 1 for RF-DAC.
- **int Tile_Id**: RF-ADC/RF-DAC tile number. Valid values are 0-3.
- **u32 Block_Id**: RF-ADC/RF-DAC number. Valid values are 0-3.

Description

Get converter connected for Q digital data path.

Return Value
Converter number.
Appendix D

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 provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open the Xilinx Documentation Navigator (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 the Xilinx Documentation Navigator, click the Design Hubs View tab.
- On the Xilinx website, see the Design Hubs page.

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

These documents provide supplemental material useful with this product guide:

1. Zynq UltraScale+ RFSoC Data Sheet: Overview (DS889)
2. Zynq UltraScale+ RFSoC Data Sheet: DC and AC Switching Characteristics (DS926)

Training Resources

1. Vivado Design Suite Hands-on Introductory Workshop
2. Vivado Design Suite Tool Flow

Revision History

The following table shows the revision history for this document.

<table>
<thead>
<tr>
<th>Section</th>
<th>Revision Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>04/17/2018 v2.0</td>
<td></td>
</tr>
<tr>
<td>Data Structures</td>
<td></td>
</tr>
<tr>
<td>User API Functions</td>
<td>Added and updated driver commands.</td>
</tr>
<tr>
<td>In-line Functions</td>
<td></td>
</tr>
<tr>
<td>04/04/2018 v2.0</td>
<td></td>
</tr>
<tr>
<td>Port Descriptions</td>
<td>Added clarification to the notes in the Port Description</td>
</tr>
<tr>
<td>Multi Converter Synchronization</td>
<td>Added Multi Converter Synchronization Ports table.</td>
</tr>
<tr>
<td>Ports</td>
<td></td>
</tr>
</tbody>
</table>
## Section | Revision Summary
--- | ---
Calibration Freeze Ports for 2 GSPS RF-ADCs | Added Calibration Freeze Ports tables.
Calibration Freeze Ports for 4 GSPS RF-ADCs |  
Basic Tab | Added Basic and Advanced Vivado IDE tab descriptions.
Advanced Tab |  
RF-ADC Tab | Improved description of RF-ADC and RF-DAC Vivado IDE tabs.
RF-DAC Tab |  
Chapter 6: Example Design | Added information on IP Integrator example design.
RF-DAC Source | Added information on RF-DAC source register map.

10/04/2017 v1.1
Major content addition and IP core updates.

04/05/2017 v1.0
Initial Xilinx release.

## Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx’s limited warranty, please refer to Xilinx’s Terms of Sale which can be viewed at [https://www.xilinx.com/legal.htm#tos](https://www.xilinx.com/legal.htm#tos); IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at [https://www.xilinx.com/legal.htm#tos](https://www.xilinx.com/legal.htm#tos).
AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS “XA” IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE (“SAFETY APPLICATION”) UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD (“SAFETY DESIGN”). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright

© Copyright 2017-2018 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. ARM is a registered trademark of ARM in the EU and other countries. All other trademarks are the property of their respective owners.