DUC/DDC Compiler
v3.0

LogiCORE IP Product Guide

Vivado Design Suite

PG147 November 18, 2015
# Table of Contents

**IP Facts**

**Chapter 1: Overview**  
Licensing and Ordering Information ......................................................... 7

**Chapter 2: Product Specification**  
Standards ........................................................................................................ 8  
Performance and Resource Utilization ......................................................... 8  
Port Descriptions ............................................................................................ 8  
Register Space ............................................................................................... 11

**Chapter 3: Designing with the Core**  
General Design Guidelines ............................................................................. 23  
Clocking ......................................................................................................... 25  
 Resets ........................................................................................................... 25  
Control Signals and Timing ........................................................................... 26

**Chapter 4: Design Flow Steps**  
Customizing and Generating the Core ............................................................. 34  
Constraining the Core .................................................................................... 39  
Simulation ...................................................................................................... 40  
Synthesis and Implementation ...................................................................... 40

**Chapter 5: C Model Reference**  
Unpacking and Model Contents ..................................................................... 41  
Installation ..................................................................................................... 42  
DUC/DDC C Model Interface ......................................................................... 42  
MATLAB ........................................................................................................ 52

**Appendix A: Migrating and Upgrading**  
Migrating to the Vivado Design Suite ............................................................ 57  
Upgrading in the Vivado Design Suite ............................................................ 57
Appendix B: Debugging

Finding Help on Xilinx.com ................................................................. 58
Debug Tools .................................................................................... 59

Appendix C: Additional Resources and Legal Notices

Xilinx Resources .............................................................................. 60
References ....................................................................................... 60
Revision History .............................................................................. 61
Please Read: Important Legal Notices .............................................. 61
Introduction

The Xilinx LogiCORE™ IP DUC/DDC Compiler implements high-performance, optimized digital up- and down-converter modules for use in wireless base stations and other suitable applications. In addition to a wide range of parameter options, resource trade-off options are available to tailor the core to a particular application.

Features

• Generates Digital Up-Converter modules for a range of output sample rates between 30.76 and 245.76 MHz
• Generates Digital Down-Converter modules for a range of input sample rates between 30.76 and 184.32 MHz
• Supports LTE (1.4, 3, 5, 10, 15 and 20 MHz channels), TD-SCDMA (1.6 MHz channel) and W-CDMA (5 MHz channel)
• Supports up to 30 carriers (maximum depends on wireless standard and channel bandwidth)
• Implementation options to configure clock rate, enable optional control signals, and set resource usage preferences
• Supports Fs/4 IF down-mixing in DDC mode
• Supports programmable carrier frequencies (within the limits imposed by wireless standard)
• Supports fixed carrier phase offsets between 0 and 2
• Supports selectable carrier relative gain levels
• AXI4-Stream data interfaces, allowing integration into signal processing data flows
• AMBA® 3 APB programming interface
• Resource and latency estimation, frequency and phase raster reporting

IP Facts

<table>
<thead>
<tr>
<th>LogiCORE IP Facts Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Specifics</td>
</tr>
<tr>
<td>Supported Device Family[1]</td>
</tr>
<tr>
<td>Supported User Interfaces</td>
</tr>
<tr>
<td>Resources</td>
</tr>
<tr>
<td>Provided with Core</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Tested Design Flows[2]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Support</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Notes:
1. For a complete listing of supported devices, see the IP catalog feature for this core.
2. For the supported versions of the tools, see the Xilinx Design Tools: Release Notes Guide.
Chapter 1

Overview

Digital Up-/Down-Converters (DUC/DDC) are key components in wireless communications systems, linking the baseband processing function with the radio front end.

A DUC forms part of the transmit path of digital radio front end (DFE) signal processing systems, and performs the function of filtering and up-converting the baseband signal to a higher sample rate to be passed to the radio front end through the Digital-to-Analog Converter (DAC), or to provide input to Crest Factor Reduction (CFR), Digital Pre-Distortion (DPD), I/Q offset correction, or other ancillary RF signal processing functions applied prior to the DAC. A DUC can include a multi-carrier mixing stage to combine multiple carriers into a composite passband signal.

A Digital Down-Converter (DDC) forms part of the receive path of a digital radio front-end signal processing system, following the Analog-to-Digital Converter (ADC), and Automatic Gain Control (AGC) or other ancillary RF signal processing functions. A DDC performs the function of filtering and down-converting the input RF sample rate to the baseband processing sample rate of the system (or an integer multiple thereof, for example 2x for symbol timing recovery.) The DDC can also perform frequency translation to shift each carrier of a multi-carrier system to baseband ready for de-modulation.

Channel selection filtering is normally incorporated into the filtering functions of DUC or DDC modules, and the sample rate conversion is normally performed most efficiently over multiple stages, with appropriate low-pass filtering for anti-aliasing or image rejection. The general architecture of a DUC or DDC therefore consists of multiple stages of filters and mixers, with the mixers being constructed variously from direct digital synthesizers, multipliers, and simple logic functions. This generalized architecture is illustrated in Figure 1-1 and Figure 1-2.
Chapter 1: Overview

Each core configuration has been designed to meet the requirements of the relevant air standard with a target spectral mask margin of approximately 5 to 10 dB. The generated core meets or exceeds the EVM, ACLR and Blocking/ACS requirements for the relevant specifications, and EVM is further limited to 1.6% or less (standard/bandwidth dependent) to provide maximum flexibility in other areas of the wireless digital front end (DFE).

The DUC/DDC Compiler core provides an easy-to-use programming interface to allow carrier positions and relative gain levels to be programmed, as well as to provide configuration information and status reporting.

The DUC offers the capability to mix multiple carriers into a complex composite signal centered around zero Hz, while the DDC configuration offers the option of additionally translating a real passband composite signal centered at $F_s/4$ Hz, where $F_s$ is the input sample rate (usually the ADC sample rate), to a complex composite signal at zero Hz. This two-stage mixing process is illustrated in Figure 1-3.

---

**Figure 1-1: Generalized DUC Architecture**

**Figure 1-2: Generalized DUC Architecture**
The core covers a wide range of parameter options, and automatically compiles a highly optimized filter cascade and mixer structure from the system-level specification. Advanced algorithms select appropriate mixing sample rates, filter types, datapath configurations, and other meta-parameters to create an efficient design that meets the performance requirements of the relevant wireless air interface specification and achieves the resource usage goals of the user.

The DUC/DDC Compiler core provides an easy-to-use programming interface to allow carrier positions and relative gain levels to be programmed, as well as providing a status reporting mechanism. This interface complies with the AMBA 3 APB bus specification.

Licensing and Ordering Information

This Xilinx LogiCORE IP module is provided at no additional cost with the Xilinx Vivado Design Suite under the terms of the Xilinx End User License. Information about this and 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 2

Product Specification

Standards

The DUC/DDC Compiler conforms to the following wireless standards:

- 3GPP TS 36.104, [Ref 2]
- 3GPP TR 25.105, [Ref 3]
- 3GPP TR 25.104, [Ref 4]

The programming interface is detailed in the AMPA 3 APB specification [Ref 1].

The data interfaces of the core conform to AMBA AXI4-Stream standards [Ref 5].

Performance and Resource Utilization

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

Port Descriptions

Table 2-1 defines the core port names and port functional descriptions.

<table>
<thead>
<tr>
<th>Name</th>
<th>Interface</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACLK</td>
<td>System</td>
<td>Input</td>
<td>Core clock (active rising edge.) Always present.</td>
</tr>
<tr>
<td>ARESETN</td>
<td>System</td>
<td>Input</td>
<td>Synchronous reset (active-Low.) Asserting ARESETN synchronously with ACLK resets the core control logic functions, but does not reset the filter data memory contents. ARESETN is an optional pin.</td>
</tr>
<tr>
<td>S_AXIS_DIN_TVALID</td>
<td>Data slave</td>
<td>Input</td>
<td>Indicates that the input data is valid. Data is input into the core when both S_AXIS_DIN_TVALID and S_AXIS_DIN_TREADY are asserted.</td>
</tr>
</tbody>
</table>
Chapter 2: Product Specification

Table 2-1: Core Signal Pinout (Cont’d)

<table>
<thead>
<tr>
<th>Name</th>
<th>Interface</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_AXIS_DIN_TREADY</td>
<td>Data slave</td>
<td>Output</td>
<td>Core ready to receive input data. Used by core to indicate required input sample rate. After reset, S_AXIS_DIN_TREADY is held High until the first time S_AXIS_DIN_TVALID goes High. Thereafter, S_AXIS_DIN_TREADY goes High for one cycle at a time, at the configured input sample rate. If input data is not provided at the required time (S_AXIS_DIN_TREADY is High but S_AXIS_DIN_TVALID is Low), then S_AXIS_DIN_TREADY is held High until S_AXIS_DIN_TVALID goes High. The core continues to process and output data that is in its pipeline, but does not start processing any new data until new data is provided (indicated by S_AXIS_DIN_TVALID going High.) This allows the external master to pause its supply of data. When this occurs, the core signals a missing input interrupt, see INT_MISSINPUT.</td>
</tr>
<tr>
<td>S_AXIS_DIN_TLAST</td>
<td>Data slave</td>
<td>Input</td>
<td>Indicates the last sample of an input packet. DUC input packet contains complex data for each carrier, I then Q for TDM format, in ascending numerical carrier order. DDC input packet contains one complex data sample, I then Q for TDM format. Signal is not present if the input packet is one cycle long (DDC, not TDM format.) If S_AXIS_DIN_TLAST is asserted for any sample other than the last sample of an input packet, or deasserted for the last sample of an input packet, then the core ignores S_AXIS_DIN_TLAST and signals a packet error interrupt, see INT_ERRPACKET.</td>
</tr>
<tr>
<td>S_AXIS_DIN_TDATA[M-1:0]</td>
<td>Data slave</td>
<td>Input</td>
<td>Aggregate sample input data bus; M indicates bus width. This aggregate signal is a concatenation of individual input samples for all antenna datapaths, including separate I and Q samples if appropriate.</td>
</tr>
<tr>
<td>M_AXIS_DOUT_TVALID</td>
<td>Data master</td>
<td>Output</td>
<td>Indicates that the output data is valid. Data is output from the core when both M_AXIS_DOUT_TVALID and M_AXIS_DOUT_TREADY are asserted.</td>
</tr>
<tr>
<td>M_AXIS_DOUT_TREADY</td>
<td>Data master</td>
<td>Input</td>
<td>Indicates that the external slave is ready to receive output data. The core presents valid output data until M_AXIS_DOUT_TREADY is asserted. The core cannot accept back pressure. If M_AXIS_DOUT_TVALID is High but M_AXIS_DOUT_TREADY is held Low, then any future output samples that the core generates are internally discarded. When this occurs, the core signals a lost output interrupt, see INT_LOSTOUTPUT. Slaves are recommended to tie M_AXIS_DOUT_TREADY High.</td>
</tr>
<tr>
<td>M_AXIS_DOUT_TLAST</td>
<td>Data master</td>
<td>Output</td>
<td>Indicates the last sample of an output packet. DUC output packet contains one complex data sample, I then Q for TDM format. DDC output packet contains complex data for each carrier, I then Q for TDM format, in ascending numerical carrier order. Signal is not present if the output packet is one cycle long (DUC, not TDM format.)</td>
</tr>
</tbody>
</table>
Chapter 2: Product Specification

Table 2-1: Core Signal Pinout (Cont’d)

<table>
<thead>
<tr>
<th>Name</th>
<th>Interface</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_AXIS_DOUT_TUSER</td>
<td>Data master</td>
<td>Output</td>
<td>Indicates when output data is clean, that is, calculated solely from input data, not from invalid data remaining within the datapath since initialization or reset. As output data from FIR filters is dependent on both input data and internal state, it takes several output samples following initialization or reset until all internal state at the time of initialization or reset has been flushed from the filters. This signal is Low for output samples following initialization or reset until the filter internal state has been flushed. Thereafter, this signal is High for all output samples (until the next reset.)</td>
</tr>
<tr>
<td>M_AXIS_DOUT_TDATA[N-1:0]</td>
<td>Data master</td>
<td>Output</td>
<td>Aggregate output sample data bus; M indicates bus width. This aggregate signal is a concatenation of individual output samples for all antenna datapaths, including separate I and Q samples if appropriate.</td>
</tr>
<tr>
<td>SREG_PRESETn</td>
<td>Programming</td>
<td>Input</td>
<td>Programming interface. Compliant with AMBA 3 APB protocol. See the AMBA 3 APB bus specification [Ref 1] for detailed information.</td>
</tr>
<tr>
<td>SREG_PADDR[11:0]</td>
<td>Programming</td>
<td>Input</td>
<td></td>
</tr>
<tr>
<td>SREG_PSEL</td>
<td>Programming</td>
<td>Input</td>
<td></td>
</tr>
<tr>
<td>SREG_PENABLE</td>
<td>Programming</td>
<td>Input</td>
<td></td>
</tr>
<tr>
<td>SREG_PWRITE</td>
<td>Programming</td>
<td>Input</td>
<td></td>
</tr>
<tr>
<td>SREG_PWDATA[31:0]</td>
<td>Programming</td>
<td>Input</td>
<td></td>
</tr>
<tr>
<td>SREG_PREADY</td>
<td>Programming</td>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>SREG_PRDATA[31:0]</td>
<td>Programming</td>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>SREG_PSLVERR</td>
<td>Programming</td>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>INT_MISSINPUT</td>
<td>Interrupt</td>
<td>Output</td>
<td>Interrupt flag indicating that a missing input sample condition has occurred. Valid input was not provided on the data input interface in the required clock cycle. The input interface paused until valid input was provided on the data input interface in some later clock cycle. The output data values are unaffected, but there is a corresponding pause in the output data rate after data has propagated through the core. When asserted, this interrupt signal remains High until the interrupt is cleared or disabled using the programming interface, or ARESETN is asserted.</td>
</tr>
<tr>
<td>INT_ERRPACKET</td>
<td>Interrupt</td>
<td>Output</td>
<td>Interrupt flag indicating an error in input packet length has occurred. S_AXIS_DIN_TLAST went High when not expected, so the input packet is too short, or did not go High when expected, so the input packet is too long. Output data might be incorrect. When asserted, this interrupt signal remains High until the interrupt is cleared or disabled using the programming interface, or ARESETN is asserted.</td>
</tr>
</tbody>
</table>
Register Space

For compatibility with SoCs that use APB to communicate with several peripherals, the DUC/DDC Compiler register map is limited to a 4KB (12-bit) address space, and the address bus, \texttt{SREG\_PADDR}, is 12 bits wide. The DUC/DDC Compiler register map is shown in Figure 2-1.

### Table 2-1: Core Signal Pinout (Cont’d)

<table>
<thead>
<tr>
<th>Name</th>
<th>Interface</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT_LOSTOUTPUT</td>
<td>Interrupt</td>
<td>Output</td>
<td>Interrupt flag indicating that a lost output sample condition has been detected. Valid output was not accepted on the data output interface in one of the required clock cycles. The following sample was dropped to maintain the output sample rate. When asserted, this interrupt signal remains High until the interrupt is cleared or disabled using the programming interface, or ARESETN is asserted.</td>
</tr>
<tr>
<td>INT_DUCDDC</td>
<td>Interrupt</td>
<td>Output</td>
<td>Combined interrupt output. This signal is the logical OR of all other interrupt output signals. It indicates that one or more interrupts are active.</td>
</tr>
</tbody>
</table>

![Figure 2-1: Register Map](image-url)
Register Definitions

The registers are summarized in Table 2-2, and described in detail in the following sections. All registers are 32 bits wide.

Table 2-2:    Registers

<table>
<thead>
<tr>
<th>Address</th>
<th>Label</th>
<th>Type</th>
<th>Init/Reset</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000</td>
<td>lock</td>
<td>RW</td>
<td>0x00000000</td>
<td>Lock Register</td>
</tr>
<tr>
<td>0x004</td>
<td>bank_select</td>
<td>RW</td>
<td>0x00000000</td>
<td>Programming Bank Selection Register</td>
</tr>
<tr>
<td>0x008-0x07C</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x080</td>
<td>config</td>
<td>RO</td>
<td>(2)</td>
<td>Configuration Register</td>
</tr>
<tr>
<td>0x084</td>
<td>mrm</td>
<td>RO</td>
<td>(2)</td>
<td>Mixing Rate Multiple Register</td>
</tr>
<tr>
<td>0x088</td>
<td>freq_raster</td>
<td>RO</td>
<td>(2)</td>
<td>Frequency Raster Register</td>
</tr>
<tr>
<td>0x08C-0x09C</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x0A0</td>
<td>int_en</td>
<td>RW</td>
<td>0x00000007</td>
<td>Interrupt Enable Register</td>
</tr>
<tr>
<td>0x0A4</td>
<td>int_stat_mask</td>
<td>RO</td>
<td>0x00000000</td>
<td>Masked Interrupt Status Register</td>
</tr>
<tr>
<td>0x0A8</td>
<td>int_stat_raw</td>
<td>RO</td>
<td>0x00000000</td>
<td>Raw Interrupt Status Register</td>
</tr>
<tr>
<td>0x0AC</td>
<td>int_clear</td>
<td>WO</td>
<td>-</td>
<td>Interrupt Clear Register</td>
</tr>
<tr>
<td>0x0B0-0x0FC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x100-0x174</td>
<td>a_frequency&lt;n&gt;</td>
<td>RW</td>
<td>(1)(3)</td>
<td>Frequency Programming Registers, bank A, &lt;n&gt; = 1 to Number of Carriers</td>
</tr>
<tr>
<td>0x178-0x1FC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x200-0x274</td>
<td>a_phase&lt;n&gt;</td>
<td>RO</td>
<td>(1)(4)</td>
<td>Phase Offset Programming Registers, bank A, &lt;n&gt; = 1 to Number of Carriers</td>
</tr>
<tr>
<td>0x278-0x2FC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x300-0x374</td>
<td>a_gain&lt;n&gt;</td>
<td>RW</td>
<td>0x00001000</td>
<td>Gain Control Programming Registers, bank A, &lt;n&gt; = 1 to Number of Carriers</td>
</tr>
<tr>
<td>0x378-0x4FC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x500-0x574</td>
<td>b_frequency&lt;n&gt;</td>
<td>RW</td>
<td>(1)(3)</td>
<td>Frequency Programming Registers, bank B, &lt;n&gt; = 1 to Number of Carriers</td>
</tr>
<tr>
<td>0x578-0x5FC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x600-0x674</td>
<td>b_phase&lt;n&gt;</td>
<td>RO</td>
<td>(1)(4)</td>
<td>Phase Offset Programming Registers, bank B, &lt;n&gt; = 1 to Number of Carriers</td>
</tr>
<tr>
<td>0x678-0x6FC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x700-0x774</td>
<td>b_gain&lt;n&gt;</td>
<td>RW</td>
<td>0x00010000</td>
<td>Gain Control Programming Registers, bank B, &lt;n&gt; = 1 to Number of Carriers</td>
</tr>
<tr>
<td>0x778-0xFFC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Notes:
1. Register is not reset by SREG_PRESETn input.
2. Initial value is calculated by the core based on several parameters.
3. Initial value of each register <n> is the quantized carrier frequency for carrier <n> divided by the frequency raster.
4. Initial value of each register <n> is the quantized carrier phase offset for carrier <n> divided by the phase raster.
If Programmable Carrier Frequencies is enabled, the Frequency Programming Registers are read/write; otherwise these registers are read only and an attempt to write to them results in a SLVERR response.

If Programmable Carrier Gain Control is enabled, the gain control programming registers are read/write; otherwise they are read only and an attempt to write to them results in a SLVERR response.

**Lock Register**

The Lock Register is a 32-bit read/write register at address 0x000 that enables or disables write access to all other registers accessible through the programming interface. The format of this register is shown in Figure 2-2.

The register bits are shown in Table 2-3.

**Table 2-3: Lock Register Bits**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Type</th>
<th>Init/Reset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:16] key</td>
<td>WO</td>
<td>-</td>
<td>Key: write only, Read As Zero. To lock, write the value 0xC705 to this field. To unlock, write the value 0x5E5A to this field.</td>
<td></td>
</tr>
<tr>
<td>[15:1]</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>Should Be Zero</td>
</tr>
<tr>
<td>0</td>
<td>L</td>
<td>RO</td>
<td>0</td>
<td>Lock: read only, ignored on writes. 0 = unlocked: programming interface writes to registers proceed as normal. 1 = locked: all programming interface writes except writes to the Lock Register fail with a SLVERR response.</td>
</tr>
</tbody>
</table>

Writes to the Lock Register are not affected by the L bit. Writes with an incorrect key (neither the lock nor unlock key) are silently ignored and do not affect the L bit. This register is reset by the SREG_PRESERTn reset input.

**Programming Bank Selection Register**

The Programming Bank Selection Register is a 32-bit read/write register at address 0x004 that selects the bank of programming registers to be active. The format of this register is
shown in Figure 2-3.

![Programming Bank Selection Register](image)

**Figure 2-3:** Programming Bank Selection Register

The register bits are shown in Table 2-4.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Type</th>
<th>Init</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:1]</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>Should Be Zero</td>
</tr>
<tr>
<td>0</td>
<td>B</td>
<td>RW</td>
<td>0</td>
<td>Bank selection: 0 = programming registers bank A is active, bank B is shadowing. 1 = programming registers bank B is active, bank A is shadowing.</td>
</tr>
</tbody>
</table>

There are two banks of programming registers — bank A and bank B. Each bank of registers contains a complete set of frequency, phase offset, and gain programming registers (see Frequency Programming Registers, page 19, Phase Offset Programming Registers, page 20 and Gain Control Programming Registers, page 21, respectively.) At any time, one bank of programming registers is active and the other bank is shadowing, allowing rapid, on-the-fly updating. The active bank is used by the core datapath and is read only; writes to registers in the active bank result in a SLVERR response. The shadowing bank is not used by the core datapath, and is read/write, allowing its use for programming new frequency, phase offset, and gain values. When the new values are correctly programmed into the shadowing bank, write the Programming Bank Selection Register to change the B bit, and the new values become the active bank and are used by the core datapath. This register is not reset by the SREG_PRESETn reset input.

Users wishing to update only a single register value can do so in a few register transactions:

1. Write to the programming register to switch banks.
2. Write the new register value.
3. Write to the programming register to switch banks back again.

This should only be attempted while the core is not actively transmitting or receiving (for example, during an inter-slot gap), as the core uses the configuration data in the alternate bank during the register write transaction time.

Any write to the Programming Bank Selection Register, whether the value of the B bit is changed or not, forces an internal core datapath reset, as if ARESETn had been asserted. This is required to allow the DDS and mixer to start using the new programmed frequency, phase offset and gain values in a consistent and predictable manner. The M_AXIS_DOUT_TVALID signal goes Low to indicate that the core has been reset, and remains Low until new input data has been processed and has propagated through the core.
The internal datapath reset does not clear the contents of internal sample history pipelines. See Filter Sample History Persistence, page 25, for the implications of this on output data values.

**Configuration Register**

The Configuration Register is a 32-bit, read-only register at address 0x080 that shows the value of key core parameters. The format of this register is shown in Figure 2-4.

The register bits are shown in Table 2-5.

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:28]</td>
<td>-</td>
<td>-</td>
<td>Should Be Zero</td>
</tr>
</tbody>
</table>
| [27:26]   | IF    | RO   | Digital IF:  
00: 0 Hz  
01: F_s/4  
All other values are reserved. |
| [25:20]   | BW    | RO   | Channel bandwidth option, also depends on S field. See Table 2-6 for details. |
| [19:16]   | S     | RO   | Wireless standard:  
0000: LTE  
0001: TD-SCDMA  
0010: W-CDMA  
All other values are reserved. |
| 15        | T     | RO   | Core type:  
0: DUC  
1: DDC |
| 14        | G     | RO   | Programmable gain control:  
0 = no programmable gain control  
1 = programmable gain control present |
| 13        | P     | RO   | Programmable carrier phase offsets. Reserved for future use; always reads as zero. |
| 12        | F     | RO   | Programmable carrier frequencies:  
0 = fixed carrier frequencies  
1 = programmable carrier frequencies |
| [11:6]    | R     | RO   | Number of antennas. Possible values are 1 to 8; all other values are reserved. |
| [5:0]     | C     | RO   | Number of carriers. Possible values are 1 to 30; all other values are reserved. |

The channel bandwidth is determined from the S and BW fields as shown in Table 2-6. All combinations of S and BW not shown are reserved. There are two settings for the same
bandwidth for each of TD-SCDMA and W-CDMA; the latter setting is associated with DDC cores only, and indicates a baseband sample rate at twice the chip rate.

**Table 2-6: Channel Bandwidth**

<table>
<thead>
<tr>
<th>S</th>
<th>Wireless Standard</th>
<th>BW</th>
<th>Channel Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>LTE</td>
<td>000001</td>
<td>1.4 MHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000011</td>
<td>3 MHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000101</td>
<td>5 MHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>001010</td>
<td>10 MHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>001111</td>
<td>15 MHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>010100</td>
<td>20 MHz</td>
</tr>
<tr>
<td>0001</td>
<td>TD-SCDMA</td>
<td>000100</td>
<td>1.6 MHz (2 x F_sym)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>000110</td>
<td>1.6 MHz (2 x F_sym)</td>
</tr>
<tr>
<td>0010</td>
<td>W-CDMA</td>
<td>000100</td>
<td>5 MHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>001000</td>
<td>5 MHz (2 x F_sym)</td>
</tr>
</tbody>
</table>

The C field, which reports the number of carriers, is expected to be the most used, to allow driver software to generate programmable carrier frequencies and gains. Therefore this field is in the LSBs of the register so that a register read and a single AND 0x3F software instruction can return the number of carriers.

The value of this register is constant and does not change during run time.

**Mixing Rate Multiple Register**

The Mixing Rate Multiple Register is a 32-bit read-only register at address 0x084 that indicates the ratio between the mixing sample rate and the frequency raster. It shows the number of possible carrier positions for both carrier frequency and carrier phase offset. The format of this register is shown in Figure 2-5.

![Figure 2-5: Mixing Rate Multiple Register](image)

The register bits are shown in Table 2-7.

**Table 2-7: Mixing Rate Multiple Register Bits**

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:0] mrm</td>
<td>RO</td>
<td>Mixing Rate Multiple, unsigned integer value</td>
</tr>
</tbody>
</table>

The mixing rate multiple (mrm) indicates the number of possible values for carrier frequency and carrier phase offset. Carrier frequency values in Frequency Programming Registers, page 19, are in the range –mrm/2 to mrm/2–1. Carrier phase offset values in
Phase Offset Programming Registers, page 20, are in the range 0 to mrm–1.

The phase raster can be calculated from the mixing rate multiple using the formula:

\[
\text{phase raster (radians)} = \frac{2\pi}{\text{mrm}}
\]

The sample rate at which multi-carrier mixing occurs can be calculated from the mixing rate multiple and the frequency raster (see Frequency Raster Register) using the formula:

\[
\text{mixing sample rate in Hz} = \text{frequency raster in Hz} \times \text{mrm}
\]

The value of this register is constant and does not change during run time.

**Frequency Raster Register**

The Frequency Raster Register is a 32-bit read only register at address 0x088 that indicates the frequency raster in Hz for the implemented wireless standard. The format of this register is shown in Figure 2-6.

The register bits are shown in Table 2-8.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:0]</td>
<td>freq_raster</td>
<td>RO</td>
<td>Frequency Raster in Hz, unsigned integer value</td>
</tr>
</tbody>
</table>

See Rasterized DDS: Specification and Programming, page 24, for details of the frequency raster.

The value of this register is constant and does not change during run time.

**Interrupt Registers**

There are four interrupt registers:

- Interrupt Enable Register
- Masked Interrupt Status Register
- Raw Interrupt Status Register
- Interrupt Clear Register

All four registers have the same format, which is shown in Figure 2-7.
Chapter 2: Product Specification

Each field in the interrupt registers corresponds to an interrupt type, as shown in Table 2-9.

### Interrupt Enable Register

The Interrupt Enable Register is a 32-bit read/write register at address 0x0A0 that enables or disables interrupts. The format and register bits are shown in Figure 2-7 and Table 2-9.

When a bit in the Interrupt Enable Register is set, the interrupt for that bit is enabled, and an interrupt shown in the Raw Interrupt Status Register is also signalled by the corresponding interrupt output going High. Clearing a bit disables the interrupt for that bit, and the corresponding interrupt is masked (the interrupt output is held Low) regardless of the interrupt status.

All interrupts are enabled (all bits corresponding to interrupts are 1) at reset. This register is reset by the SREG_PRESETn reset input.

### Masked Interrupt Status Register

The Masked Interrupt Status Register is a 32-bit read-only register at address 0x0A4 that provides the interrupt status taking into account interrupt enabling. This is the AND of the Raw Interrupt Status Register and the Interrupt Enable Register. The Masked Interrupt Status Register directly indicates the status of the interrupt output pins. The format and register bits are shown in Figure 2-7 and Table 2-9.

When a bit in the Masked Interrupt Status Register is High, the interrupt for that bit is triggered and enabled. When a bit is Low, the interrupt for that bit is either not triggered or is not enabled, and so has been masked.

All bits are zero initially. This register is not reset by the SREG_PRESETn reset input. Interrupts are reset (cleared) by the ARESETN reset input, and the Interrupt Enable Register is reset by the SREG_PRESETn reset input; therefore this register can change value on either reset.

### Table 2-9: Interrupt Register Bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:3]</td>
<td>-</td>
<td>Should Be Zero</td>
</tr>
<tr>
<td>2</td>
<td>L</td>
<td>Lost output sample interrupt</td>
</tr>
<tr>
<td>1</td>
<td>P</td>
<td>Input packet length error interrupt</td>
</tr>
<tr>
<td>0</td>
<td>M</td>
<td>Missing input sample interrupt</td>
</tr>
</tbody>
</table>

*Figure 2-7: Interrupt Registers*
Raw Interrupt Status Register

The Raw Interrupt Status Register is a 32-bit read-only register at address 0x0A8 that provides the interrupt status ignoring interrupt enabling. The Raw Interrupt Status Register indicates the status of interrupts from the core before masking. The Raw Interrupt Status Register can differ from the status of the interrupt output pins if one or more interrupts are disabled using the Interrupt Enable Register. The format and register bits are shown in Figure 2-7 and Table 2-9.

When a bit in the Raw Interrupt Status Register is High, the interrupt for that bit is triggered. When a bit is Low, the interrupt for that bit is not triggered.

All bits are zero initially. This register is not reset by the SREG_PRESETn reset input. Interrupts are reset (cleared) by the ARESETN reset input; therefore this register might change value on ARESETN.

Interrupt Clear Register

The Interrupt Clear Register is a 32-bit write-only register at address 0x0AC for clearing interrupts. The format and register bits are shown in Figure 2-7 and Table 2-9.

Writing one to a bit in the Interrupt Clear Register clears the corresponding bit in the Raw Interrupt Status Register, thereby clearing the interrupt and setting the corresponding interrupt output pin Low. Writing zero to a bit has no effect.

Frequency Programming Registers

The Frequency Programming Registers are two banks, bank A and bank B, of C 32-bit read/write registers, where C is the number of carriers (reported in the C field of the Configuration Register, page 15.) The registers in bank A are at sequential word addresses starting at address 0x100; the registers in bank B are at sequential word addresses starting at address 0x500. The a_frequency<n> register for carrier n is at address (0x100 + 4 × (n – 1)), and the b_frequency<n> register for carrier n is at address (0x500 + 4 × (n – 1)). Each a_frequency<n> and b_frequency<n> register holds the frequency for carrier n as a multiple of the frequency raster, given by the Frequency Raster Register, page 17.

The format of each a_frequency<n> and b_frequency<n> register is shown in Figure 2-8.

![Figure 2-8: a_frequency<n> and b_frequency<n> Registers](image-url)
The register bits are shown in **Table 2-10**.

**Table 2-10: a_frequency<n> and b_frequency<n> Register Bits**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Type</th>
<th>Init</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:0]</td>
<td>frequency</td>
<td>RW (1)</td>
<td>Carrier ( n ) frequency as a multiple of the frequency raster. 2's complement integer in the range (-mrm/2) to (mrm/2-1), where (mrm) is the mixing rate multiple, see Mixing Rate Multiple Register, page 16.</td>
<td></td>
</tr>
</tbody>
</table>

Notes:
1. Initial value of each register \( <n> \) is the quantized carrier frequency for carrier \( <n> \) divided by the frequency raster.

Software drivers can calculate the correct value to write to a Frequency Programming Register using the required frequency in Hz and the frequency raster; see Frequency Raster Register, page 17.

\[
\text{frequency programming value} = \frac{\text{required frequency in Hz}}{\text{frequency raster in Hz}}
\]

The Programming Bank Selection Register, page 14, selects which bank of Frequency Programming Registers is active (used by the core datapath) and which is shadowing (not used by the core datapath but available for programming.) Frequency Programming Registers in the active bank are read only — an attempt to write an active bank Frequency Programming Register results in a SLVERR response. Frequency Programming Registers in the shadowing bank are read/write if Programmable Carrier Frequencies is enabled, and read only otherwise. Values written to Frequency Programming Registers in the shadowing bank are not used by the core datapath until the Programming Bank Selection Register is modified to swap the active and shadowing banks.

An attempt to write a Frequency Programming Register with a value that is out of the legal range results in a SLVERR response and does not change the register value. An attempt to read or write a Frequency Programming Register that does not exist (that is, where \( n \) is greater than the number of carriers) results in a SLVERR response.

For a single carrier, the single Frequency Programming Register in each bank is read only and its value is set to zero.

These registers are *not* reset by the SREG_PRESEn reset input.

**Phase Offset Programming Registers**

The Phase Offset Programming Registers are two banks, bank A and bank B, of C 32-bit read only registers, where C is the number of carriers (reported in the C field of the Configuration Register, page 15.) The registers in bank A are at sequential word addresses starting at address 0x200; the registers in bank B are at sequential word addresses starting at address 0x600. The a_phase\(<n>\) register for carrier \( n \) is at address \(0x200 + 4 \times (n - 1)\)), and the b_phase\(<n>\) register for carrier \( n \) is at address \(0x600 + 4 \times (n - 1)\)). Each a_phase\(<n>\) and b_phase\(<n>\) register holds the phase offset for carrier \( n \) as a multiple of the phase raster, which can be calculated from the Mixing Rate Multiple Register, page 16.
The format of each `a_phase<n>` and `b_phase<n>` register is shown in Figure 2-9.

![Figure 2-9: a_phase<n> and b_phase<n> Registers](image)

The register bits are shown in Table 2-11.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Field</th>
<th>Type</th>
<th>Init</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:0]</td>
<td>phase</td>
<td>RO</td>
<td>(1)</td>
<td>Carrier n phase offset as a multiple of the phase raster. Unsigned integer in the range 0 to (mrm-1), where (mrm) is the mixing rate multiple, see Mixing Rate Multiple Register, page 16.</td>
</tr>
</tbody>
</table>

Notes:
1. Initial value of each register \(<n>\) is the quantized carrier phase offset for carrier \(<n>\) divided by the phase raster.

The value of a Phase Offset Programming Register is calculated from the corresponding carrier phase offset in radians and the phase raster, which is derived from the mixing rate multiple, see Mixing Rate Multiple Register, page 16.

Phase Offset Programming Register value = carrier phase offset in radians / phase raster in radians

All Phase Offset Programming Registers in both bank A and bank B are read only. Carrier phase offsets cannot be changed at run time.

**Gain Control Programming Registers**

The Gain Control Programming Registers are two banks, bank A and bank B, of C 32-bit read/write registers, where C is the number of carriers (reported in the C field of the Configuration Register, page 15.) The registers in bank A are at sequential word addresses starting at address 0x300; the registers in bank B are at sequential word addresses starting at address 0x700. The `a_gain<n>` register for carrier n is at address (0x300 + 4 \(\times (n-1)\)), and the `b_gain<n>` register for carrier n is at address (0x700 + 4 \(\times (n-1)\)). Each `a_gain<n>` and `b_gain<n>` register holds the gain for carrier n.

The format of each `a_gain<n>` and `b_gain<n>` register is shown in Figure 2-10.

![Figure 2-10: a_gain<n> and b_gain<n> registers](image)
The register bits are shown in Table 2-12.

### Table 2-12: \texttt{a\_gain<n>} and \texttt{b\_gain<n>} Register Bits

<table>
<thead>
<tr>
<th>Bit Field</th>
<th>Type</th>
<th>Init</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:17]</td>
<td></td>
<td>0</td>
<td>Should Be Zero</td>
</tr>
<tr>
<td>[16:8]</td>
<td>gain</td>
<td>RW 100000000 (that is, 1.0)</td>
<td>Carrier (n) gain. Unsigned fixed point value in the range 0.0 to 1.0. The binary point is fixed, between bits 16 and 15. Only power of 2 values are allowed, that is, gain field must be one-hot or all zeros.</td>
</tr>
</tbody>
</table>

Gain control allows the relative amplitude of carriers to be adjusted, for example for power management, by attenuating one or more carriers. The amplitude of each carrier is attenuated by the programmed value. For example, writing a gain value of \texttt{001000000} to carrier one Gain Control Programming Register corresponds to a gain of 0.25, and all data values for carrier one are multiplied by 0.25 prior to multi-carrier mixing.

The Programming Bank Selection Register, page 14, selects which bank of Gain Control Programming Registers is active (used by the core datapath) and which is shadowing (not used by the core datapath but available for programming). Gain Control Programming Registers in the active bank are read only — an attempt to write an active bank Gain Control Programming Register results in a \texttt{SLVERR} response. Gain Control Programming Registers in the shadowing bank are read/write if Programmable Carrier Gain Control is enabled, and read-only otherwise. Values written to Gain Control Programming Registers in the shadowing bank are not used by the core datapath until the Programming Bank Selection Register is modified to swap the active and shadowing banks.

An attempt to write a Gain Control Programming Register with an illegal value or with a value that is out of the legal range results in a \texttt{SLVERR} response and does not change the register value. An attempt to read or write a Gain Control Programming Register that does not exist (that is, where \(n\) is greater than the number of carriers) results in a \texttt{SLVERR} response.

For a single carrier, the single Gain Control Programming Register in each bank is read-only and its value is set to \texttt{100000000} (that is, 1.0).

Gain control is not available for DDCs: all Gain Control Programming registers are read-only in DDCs.

These registers are not reset by the \texttt{SREG\_PRESETr<n>} reset input.
Chapter 3

Designing with the Core

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

General Design Guidelines

The DUC/DDC Compiler provides coverage of a wide range of parameters that affect the design implementation while presenting a simple interface to the user; detailed knowledge of the internal workings of the core are generally unnecessary. The core handles configuration by querying a compiled database of highly optimal design configurations, which provides configuration information on appropriate rate change steps, filter configurations and numbers of datapaths at each stage, mixer sample rate and configuration and flow control signal handling.

The basic static configuration parameter options for the core and their effects are detailed in Chapter 4, Design Flow Steps, while the programming interface and registers define dynamic configuration options and their effects. Where some additional understanding of the core configuration and implementation is beneficial to ease integration of the core into an application system, further information is provided in this section.

The DUC/DDC Compiler makes extensive use of the LogiCORE IP FIR Compiler (v5.0) as a component core, including multiple instances of FIR filters in cascades to achieve the desired up- or down-conversion function. Familiarity with the data sheet for that core is advised to better understand some of the features of this core, particularly with regard to behavior under reset conditions. The implications of FIR filter reset behavior for overall DUC/DDC as a whole are described in Filter Sample History Persistence, page 25.

Input Range

The input data format is a binary word of user-configurable width with arbitrary scaling (nominally the range is considered as being between −1.0 and 0.99999...)

A further restriction must be considered when using an F_s/4 IF mixer stage. In such cases, the input is saturated at −0.999... to 0.999... to avoid any increase in integer bit representation, as the F_s/4 sampling operation utilizes a logical inversion stage, and therefore the data input signal must have a symmetrical range about zero. Saturation is performed within the core; however, users should be aware of this fact and might wish to
scale input data appropriately to minimize the frequency and impact of that saturation operation.

**Internal Range**

Integer bit growth may be introduced in filters and mixers along the data path, and the core accumulates these bits to avoid signal clipping (no saturation stages are inserted in the datapath processing chain.) Generally, the gain of the filters in the implemented cores is configured to maintain the dynamic range of the datapath (unity gain, or gain scaled by the rate change factor; however, some headroom is added in the initial filter stage (one extra integer bit), and DUC mixer stages (up to four extra bits) introduce bit growth due to accumulation. The Data Format information in the GUI provides feedback on the number of integer bits at the output stage.

The output of each filter stage and each mixer operation is rounded using a Convergent to Even rounding scheme throughout. Internal data bit width representation retains as much precision as is practical. The single additional integer bit inserted in the initial filtering stage is more than sufficient to restrict overflow conditions throughout the filter chain to acceptable levels. The additional integer bits inserted by the DUC mixer are calculated based on the assumption that the RMS power of a composite multi-carrier signal is approximately equal to the RMS power of the input multiplied by the square root of the number of carriers which have been combined. These calculations are based on generally similar individual carriers in terms of bandwidth and power spectral density. By using the range handling measures, overflow is unlikely, but if it occurs, it is handled by wrapping.

**Output Range and Scaling**

As described in Internal Range, growth in the number of integer bits through the various filter stages and multi-carrier mixer accumulator is controlled to maintain a balance between likelihood of overflow and achieving sufficient precision to achieve the desired performance specified by the relevant air standard. The GUI provides a text field to indicate the output integer bits utilized by the current configuration. Generally, the core adds one additional integer bit for headroom and then the filter chain maintains unity gain as far as is practical; the main exception to this pattern is the case of multi-carrier DUCs, in which case the mixer accumulation leads to additional integer bits.

**Rasterized DDS: Specification and Programming**

The multi-carrier mixing function uses an efficient form of Direct Digital Synthesizer which is commonly referred to as a rasterized DDS, in that it can only produce frequencies which fall on a raster of fixed frequency values separated by a constant frequency difference (the raster step.) Wireless air interface standards generally have a specified raster of frequencies at which carriers can be located; for LTE, the minimum raster is 100 kHz, while for TD-SCDMA and W-CDMA it is 200 kHz. There are certain special cases included in the latest revisions of the W-CDMA and TD-SCDMA standards which call for a 100 kHz offset and therefore the lower 100 kHz figure is used as the base requirement for all cases. The raster
step used in the DDS for the core depends on the desired mixing frequency, and is sometimes smaller than that specified in the wireless standard (the implementation raster step is in fact the greatest common divisor of the wireless standard raster and the mixing sample rate selected by the core.)

The frequency raster step is shown in the GUI in the Quantization section, see Carrier Specification Tab(s) in Chapter 4. All carrier frequencies are multiples of this frequency raster. Quantization of carrier frequencies to make them multiples of the frequency raster is performed automatically by the GUI.

Users are highly recommended to restrict the range of carrier frequencies such that all carriers lie within the width of the IF Passband, centered at zero Hz. The filters in the core are designed for that range of carrier frequencies, and carriers outside this range are strongly attenuated.

**Filter Sample History Persistence**

The DUC/DDC core contains several internal FIR filter stages. Each filter stage has a sample history pipeline that contributes to the overall sample latency of the core. The core does not clear the contents of these pipelines at initialization and after reset events; therefore consistent operation is only guaranteed when these pipelines have flushed with known sample values, which takes several samples equal to the total sample latency, or, in other words, the impulse response length of the full filter cascade. The `M_AXIS_DOUT_TUSER` signal denotes the period following reset and initialization during which time the sample history might be unknown, should that information be required. For further information on `M_AXIS_DOUT_TUSER` timing behavior, see Master Data Output Interface.

**Clocking**

The core uses a single clock for all functions and interfaces.

**Resets**

The core uses a single active-Low reset signal for all functions and interfaces.
Control Signals and Timing

The DUC/DDC Compiler core has five interfaces:

- **System Interface**: clock and reset signals
- **Slave Data Input Interface**: AXI4-Stream slave
- **Master Data Output Interface**: AXI4-Stream master
- **Programming Interface**: AMBA 3 APB slave
- **Interrupt Interface**: interrupt output signals

**System Interface**

The system interface consists of a single clock and a single synchronous reset.

All interfaces and all internal logic use the same single clock, ACLK.

The synchronous reset input `ARESETN` is active-Low, and applies to the datapath, the data input interface, the data output interface, and the interrupt interface. A second reset input, `SREG_PRESETn`, is provided for resetting the programming interface and registers: see **Programming Interface, page 32**. Both reset inputs are present only when **Optional Pins - Reset** is selected in the GUI.

**Slave Data Input Interface**

The slave data input interface is an AXI4 compliant data streaming interface that receives data samples that are to be input to the core. The core asserts `S_AXIS_DIN_TREADY` when it is ready to receive a data sample. `S_AXIS_DIN_TVALID` input indicates that the input data is valid. Data is input into the core when both `S_AXIS_DIN_TREADY` and `S_AXIS_DIN_TVALID` are asserted.

After reset, when the core is ready to receive the first input data sample set, the core holds `S_AXIS_DIN_TREADY` High until `S_AXIS_DIN_TVALID` goes High to indicate the first input data sample. Thereafter, the core expects input data to be provided at a steady rate, and it indicates the expected data rate by asserting `S_AXIS_DIN_TREADY`. If input data is not provided at the expected time (`S_AXIS_DIN_TREADY` is High but `S_AXIS_DIN_TVALID` is Low), then the core holds `S_AXIS_DIN_TREADY` High until `S_AXIS_DIN_TVALID` goes High, and signals a missing input interrupt on the `INT_MISSINPUT` interrupt output. The core continues to process and output data that is in its datapath while it waits for new data. There is a corresponding pause in the output data rate when data in the core datapath has been output from the core.
Chapter 3: Designing with the Core

The data bus input to the core, `S_AXIS_DIN_TDATA`, is an amalgamated bus formed by concatenation of individual data samples, covering all antennas and including both In-phase and Quadrature sample values where appropriate.

Depending on the core configuration, the data is input in one of three ways:

- **DDC cores with Digital IF set to \(F_s/4\):** Real only input data from each antenna is sign-extended to the nearest byte boundary and concatenated together from lowest antenna number in the LS position to highest antenna number in the MS position (see Figure 3-1).

- **Any DUC core, or a DDC core with Digital IF set to Zero, and parallel I and Q signal format selected:** Complex input data, with in-phase (I) and quadrature (Q) portions sign-extended separately, then concatenated with I least significant and Q most significant, concatenated together from lowest antenna number in the LS position to highest antenna number in the MS position (see Figure 3-2).

![Figure 3-1: DDC Cores with Digital IF set to \(F_s/4\)](image1)

![Figure 3-2: DUC/DDC Core with Digital IF set to Zero and Parallel I, Q Signal Format](image2)
• Any DUC core, or a DDC core with Digital IF set to Zero, and TDM mode I and Q signal format selected: Complex input data, with in-phase and quadrature portions sign-extended in a TDM (alternating) format, and concatenated together from lowest antenna number in the LS position to highest antenna number in the MS position (see Figure 3-3).

\[
\text{S_AXIS_DIN_TDATA[63:0]}
\]

\[
\begin{array}{ccc}
\text{T0} & \text{T1} & \text{T2} \\
\end{array}
\]

**Figure 3-3:** DUC/DDC Core with Digital IF set to Zero, TDM mode I, Q Signal Format

The minimum TDATA width that could occur is in the case of a single antenna, single carrier TDM input format DUC, or a single antenna DDC with either TDM input format or digital IF at Fs/4 (real data only). In such cases, using the minimum constituent data unit width of 11, sign-extension to the next byte boundary results in a data to 16. The maximum TDATA width that could occur is in the case of a full complement of 8 antennas, parallel I/Q input/output format DUC. In such cases, using the maximum constituent data unit width of 17, sign-extended to 24 bits, the output width would be 8 x 2 x 24 = 384.

\( S_{\text{AXIS_DIN_TLAST}} \) input indicates the last sample of an input packet: this is the input data sample for the last carrier (in a multi-carrier DUC), the quadrature portion if TDM format is used. The core expects input data to be provided for each carrier in ascending carrier order (in a multi-carrier DUC), and in-phase then quadrature data if TDM format is used. If \( S_{\text{AXIS_DIN_TLAST}} \) is asserted for any sample other than the last sample of an input packet, or is not asserted for the last sample of an input packet, then the core ignores \( S_{\text{AXIS_DIN_TLAST}} \) and signals a packet error interrupt on the \( \text{INT_ERRPACKET} \) interrupt output. It is not possible to change the input sample order using \( S_{\text{AXIS_DIN_TLAST}} \). A DDC or a single carrier DUC with Data Interface Format of separate I and Q signals has an input packet containing only one sample: in this configuration, \( S_{\text{AXIS_DIN_TLAST}} \) is not present.

**Timing Diagrams**

**Figure 3-4**, **Figure 3-5**, **Figure 3-6** and **Figure 3-7** show timing diagrams for four configurations of the core:

**Figure 3-4:** DDC with Digital IF of Fs/4, real data input, 1 antenna
Chapter 3: Designing with the Core

Figure 3-5: DDC with Digital IF of 0 Hz, TDM format, 1 antenna

Figure 3-6: DUC with 3 carriers, separate I and Q format, 1 antenna

Figure 3-7: DUC with 3 carriers, TDM format, 1 antenna

In each case, the input packet is eight clock cycles in length. The effects of S_AXIS_DIN_TVALID not being High when S_AXIS_DIN_TREADY is High causing a missing input interrupt, and incorrect S_AXIS_DIN_TLAST (where present) causing a packet error interrupt, are shown towards the end of each timing diagram. Interrupt signals are asserted two clock cycles after the interrupt event.
Chapter 3: Designing with the Core

Master Data Output Interface

The master data output interface is an AXI4 compliant data streaming interface that outputs data samples from the core. The core asserts M_AXIS_DOUT_TVALID when new output data is valid and presented on the interface. The core holds M_AXIS_DOUT_TVALID High until M_AXIS_DOUT_TREADY is asserted. Data is output from the core when both M_AXIS_DOUT_TVALID and M_AXIS_DOUT_TREADY are asserted.

At reset and initialization, the core holds M_AXIS_DOUT_TVALID Low until input data has been received and processed by the core datapath, and is ready to be output. Thereafter, the core attempts to output data at a steady rate, and it asserts M_AXIS_DOUT_TVALID when each output sample is available.

If input data was not provided at the slave data input interface when the core expected it, the core waits until input data is provided, but continues to process and output data that is in its datapath. When all data samples in the datapath have been output, the core holds M_AXIS_DOUT_TVALID Low until new input data has been received and processed by the core datapath, and is ready to be output. This results in a pause in the steady rate of output data. The core indicates that this happens by signalling a missing input interrupt on the INT_MISSINPUT interrupt output when the input data was not provided at the expected time.

The core cannot accept back pressure. It must produce output samples at the rate determined by the input data rate and the sample rate change performed by the core. If M_AXIS_DOUT_TVALID is High but M_AXIS_DOUT_TREADY is held Low, then any future output samples that the core generates is internally discarded, and the core signals a lost output interrupt on the INT_LOSTOUTPUT interrupt output. Slaves are recommended to tie M_AXIS_DOUT_TREADY High.

Similarly to the slave input data bus, S_AXIS_DIN_TDATA, the data bus output from the core, M_AXIS_DOUT_TDATA, is an amalgamated bus formed by concatenation of individual data samples, covering all antennas and including both in-phase and quadrature sample values where appropriate.

- Parallel I and Q signal format selected: Complex output data, with in-phase (I) and quadrature (Q) portions sign-extended separately, then concatenated with I least significant and Q most significant, concatenated together from lowest antenna number in the LS position to highest antenna number in the MS position.
- TDM mode I and Q signal format selected: Complex input data, with in-phase (I) and quadrature (Q) portions sign-extended in a TDM (alternating, I first) format, and concatenated together from lowest antenna number in the LS position to highest antenna number in the MS position.

The MDATA_LAST output indicates the last sample of an output packet: this is the output data sample for the last carrier (in a multi-carrier DDC), the quadrature portion if TDM format is used. A DUC or a single carrier DDC, with a data interface format of separate I and
Chapter 3: Designing with the Core

Q signals, has an output packet containing only one sample: in this configuration, MDATA_LAST is not present.

**Timing Diagrams**

Figure 3-8, Figure 3-9, Figure 3-10 and Figure 3-11 show timing diagrams for four configurations of the core:

- **Figure 3-8**: DUC with separate I and Q format, 1 antenna
- **Figure 3-9**: DUC with TDM format, 1 antenna
- **Figure 3-10**: DDC with 3 carriers, separate I and Q format, 1 antenna
- **Figure 3-11**: DDC with 3 carriers, TDM format, 1 antenna

In each case, the output packet is eight clock cycles in length. The effect of M_AXIS_DOUT_TREADY being held Low for long enough to cause a lost output interrupt is shown towards the end of each timing diagram. Interrupt signals are asserted two clock cycles after the interrupt event.
Chapter 3: Designing with the Core

Programming Interface

The programming interface is an AMBA 3 APB slave interface for programming carrier frequencies and carrier gain control for providing configuration, status and error information about the core. The programming interface is always present.

The programming interface complies with the AMBA 3 APB interface specification [Ref 1]. See this specification for detailed information about the interface.

The core uses SREG_PREADY to insert wait states to extend APB transactions. All transactions on the programming interface use at least three wait states. Reads and writes to Frequency Programming Registers and Gain Control Programming Registers sometimes use more wait states, up to a maximum of nine.

SREG_PRESETn is the reset signal for the programming interface and registers. This reset signal is synchronous and active-Low. SREG_PRESETn is registered internally to aid timing closure, and takes effect one cycle after it is synchronously asserted.

Interrupt Interface

The interrupt interface is a set of interrupt output pins, each corresponding to a particular interrupt type, plus one combined interrupt output pin that indicates an interrupt of any type:

- INT_MISSINPUT indicates a missing input interrupt, see Slave Data Input Interface, page 26 for details and timing diagrams.
- INT_ERRPACKET indicates an input packet length error interrupt, see Slave Data Input Interface, page 26 for details and timing diagrams.
- INT_LOSTOUTPUT indicates a lost output interrupt, see Master Data Output Interface, page 30 for details and timing diagrams.
- INT_DUCTDC indicates an interrupt of any type, and is the logical OR of the three preceding interrupt signals.

Interrupt signals are asserted two clock cycles after the corresponding interrupt event.
All interrupt outputs are always present. Each interrupt type can be independently enabled or disabled using the Interrupt Enable Register. When an interrupt is disabled, the corresponding interrupt signal is held Low at all times, whether an interrupt has occurred or not. INT_DUC/DDC cannot be independently enabled or disabled: it is the logical OR of the other three interrupt signals, and therefore takes into account the individual interrupt enables.

The current status of interrupts and interrupt signals is also available in the Raw Interrupt Status Register and Masked Interrupt Status Register respectively.

All interrupt outputs are sticky, and when they go High they stay High until disabled by writing to the Interrupt Enable Register, or cleared by writing to the Interrupt Clear Register or resetting the core by asserting ARESETN.
Chapter 4

Design Flow Steps

This chapter 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) [Ref 7]
- Vivado Design Suite User Guide: Getting Started (UG910) [Ref 8]
- Vivado Design Suite User Guide: Logic Simulation (UG900) [Ref 9]

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) [Ref 6] 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 you can 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) [Ref 7] and the Vivado Design Suite User Guide: Getting Started (UG910) [Ref 8].

The DUC/DDC Compiler core GUI has several pages with fields in which to set parameter values for the particular configuration required, while also providing some user feedback
for information about the implementation. This section provides a description of each GUI field.

**IP Symbol Tab**

The IP Symbol tab illustrates the core pinout. AXI bus ports can be expanded to show individual signals.

**Implementation Tab**

The Implementation tab displays:

- Resource estimation information
- Input data quantization and scaling details
- Estimated core latency in both clock cycles and time based on the specified clock rate

The estimated number of DSP slices is displayed in addition to an approximate count of the number of block RAM elements required to implement the design. Usage of general slice logic is not currently estimated. The results in the resource estimation are estimates only using equations that model the expected core implementation structure. The core should be implemented following generation for a more accurate report on resource usage. The final resource cost when integrated into the user application system might differ due to additional routing congestion. It is not guaranteed that the resource estimates provided in the GUI match the results of a mapped core implementation.

The latency report box provides an indication of the approximate expected delay of the core from input to output, in terms of both clock cycles and absolute time. This information is a useful metric for the core in assessing suitability for particular applications. The figure is approximate and users are encouraged to confirm the actual latency in simulation following core generation.

The Data Format box provides information on the quantization and scaling used by the core. Core inputs are assumed to range between –1.0 to +0.999..., that is, one integer bit with the remaining bits being fractional. Core outputs are scaled by the integer bit growth through the core. The Data Format information is provided to allow the user to decide how to handle the output samples. For example, the user might decide to saturate and round the sample values at the core output based on the reduced fractional width, removing some or all of the additional integer bits.

**Configuration Tab**

- **Component Name**: The name of the core component to be instantiated. The name must begin with a letter and be composed of the following characters: a to z, 0 to 9, and “_”.
- **Core Type**: Select between DUC and DDC options.
• **Wireless Standard**: Select the required wireless air interface standard, LTE, TD-SCDMA or W-CDMA.

• **Channel Bandwidth**: Select between 1.4, 3, 5, 10, 15 and 20 MHz channel bandwidth options for LTE. This field is set automatically to 1.6 MHz for TD-SCDMA and to 5 MHz for W-CDMA.

• **Baseband Sample Rate**: The baseband sample rate value is selected automatically based on the Channel Bandwidth setting. The relevant values are: 1.28 or 2.56 Ms/s for TD-SCDMA 1.6 MHz channel (normal or over-sampled); 3.84 or 7.68 Ms/s for W-CDMA 5 MHz channel (normal or over-sampled); 1.92, 3.84, 7.68, 15.36, 23.04, 30.72 Ms/s for LTE 1.4, 3, 5, 10, 15, 20 MHz channels respectively.

• **RF Sample Rate**: Select the RF sample rate value to suit the data converter sample rates in the application system, or to match up with another intermediate sample rate. For DUC implementation, the allowable RF sample rate values are: 30.76, 61.44, 76.80, 92.16, 122.88, 153.60, 184.32, and 245.76 Ms/s; while for DDCs, the range is: 30.76, 61.44, 76.80, 92.16, 122.88, 153.60, and 184.32 Ms/s.

• **Digital IF**: Select the intermediate frequency mixing option to implement, either zero IF or F_s/4. Only available for DDC configurations. The F_s/4 option adds an efficient quarter sample rate frequency translation stage to convert the RF input signal from a real passband signal centered at F_s/4, where F_s is the RF Sample Rate value, to a complex passband signal centered at 0 Hz ready for extraction of individual carrier sample streams from the composite multi-carrier signal.

• **Number of Carriers**: Select the required number of carriers. Support for carrier options is dependent upon the Wireless Standard and Channel Bandwidth already selected. Table 4-1 shows the carrier options that are supported in the core. Some RF Sample Rate values reduce the number of carriers supported.

**Table 4-1: Carrier Support Options**

<table>
<thead>
<tr>
<th>Wireless Standard</th>
<th>Channel Bandwidth</th>
<th>Supported Number of Carriers</th>
</tr>
</thead>
<tbody>
<tr>
<td>LTE</td>
<td>1.4</td>
<td>1, 2, 3, 4, 5, 6, 8, 10, 12, 16</td>
</tr>
<tr>
<td>LTE</td>
<td>3</td>
<td>1, 2, 3, 4, 5, 6, 8, 10, 12, 16</td>
</tr>
<tr>
<td>LTE</td>
<td>5</td>
<td>1, 2, 3, 4, 5, 6, 8, 10</td>
</tr>
<tr>
<td>LTE</td>
<td>10</td>
<td>1, 2, 3, 4, 5, 6, 8</td>
</tr>
<tr>
<td>LTE</td>
<td>15</td>
<td>1, 2, 3, 4</td>
</tr>
<tr>
<td>LTE</td>
<td>20</td>
<td>1, 2, 3, 4</td>
</tr>
<tr>
<td>TD-SCDMA</td>
<td>1.6</td>
<td>1, 3, 6, 9, 12, 15, 18, 24, 30</td>
</tr>
<tr>
<td>W-CDMA</td>
<td>5</td>
<td>1, 2, 3, 4, 5, 6, 8</td>
</tr>
</tbody>
</table>

• **IF Passband**: Select the required IF passband width in which you wish to place the carriers. The range of valid values for this option is: 5, 10, 15, 20, 30, 40, 50, 60, 80 and 100 MHz; however, some limits are applied to limit this range further. The minimum value is the smallest option in which all carriers fit, while the largest option is no more than twice than the minimum, or half of the RF Sample Rate value, whichever is smaller.
Chapter 4: Design Flow Steps

- **Number of Antennas**: Select the number of antennas to be implemented, up to 8 antennas in total. A separate datapath and rate conversion filter cascade are implemented for each antenna, and appropriate I/O pins appear on the IP symbol and in the output netlist for these datapaths.

**Implementation Tab**

- **Clock Frequency**: The clock frequency is selected using a list box which is limited to integer multiples of the RF Sample Rate, within the limits of the selected FPGA family and the selected speed grade.

- **Data Precision – Input Data Width**: Select the required input data width, from 11 to 17 bits.

- **Data Precision – Output Data Width**: Select the required output data width, from 11 to 18 bits.

- **Data Interface Format**: Select the interface format used for both input and output data ports, either separate I & Q data ports or a single complex data port with I & Q supplied in a TDM format (In-Phase first.) This option is only available if the clock frequency is at least twice the RF Sample Rate value; if it is less than twice the RF Sample Rate, it is set to separate I & Q data ports. The option is also disabled when using a DDC with the digital IF set as $F_s/4$, in which case only a real valued data signal is provided as input, and the output is separate I & Q data ports.

- **Optional Pins – Reset**: Select whether or not to add reset capability to the core. This option adds two reset pins to the core, one for the main datapath logic ($ARESETN$, active-Low) and another for the programming port ($SREG_PRESETn$, active-Low.) $ARESETN$ does not reset filter sample history.

- **Optimize Options – Implementation Goal**: Select either Minimum Area or Maximum Speed as an Implementation Goal. The recommendation for this setting is to use Minimum Area first, as generally this setting also achieves maximum performance and does not require any additional resources; however, if timing goals are not being achieved, the architectural changes enabled by the Maximum Speed setting can improve results, at the expense of an increase in core resource usage.

- **BRAM Usage**: Select whether to use more block RAMs or leave the decision to the filter sub-core functions to select storage type appropriately. This optimization directive is a goal and does not remove all block RAM usage from the core.

**Carrier Specification Tab(s)**

When multiple carriers are used, the Carrier Specification tab allows the specification of carrier frequencies and phase offsets for each carrier. Where more than 18 carriers are used (TD-SCDMA only), the carrier specification section runs onto a second tab.

*Note*: These tabs are not present when a single carrier configuration is used.
Chapter 4: Design Flow Steps

- **Carrier Specification Table**: The Carrier Specification Table presents the user with a set of text entry cells in which to specify ideal carrier frequencies and phase offsets, and a matching set of report text cells that report the closest achievable quantized equivalent of these ideal values.

- **Quantization**: This text box provides the user with information on the quantization units that can be expected when checking the quantized cells in the Carrier Specification Table. The quantized values for frequency and phase are multiples of the specified quanta.

- **Programmable Carrier Frequencies**: Select the capability to re-program carrier frequencies using the programming port.

- **Programmable Carrier Gain Control**: Select the capability to scale carrier amplitudes prior to multi-carrier mixing by a programmable gain factor. This feature is only available for a DUC, and is limited to integer powers of 2 only, with a nominal maximum gain of 1.0 ($2^0$) and a minimum positive gain of $2^{-8}$, with zero gain also available.

### Using the DUC/DDC Compiler IP Core

The GUI performs error-checking on all input parameter sets, applying range or value limitations as appropriate and providing feedback to the user to guide configuration. Resource estimation and approximate latency information are also available.

### User Parameters

Table 4-2 shows the relationship between the GUI fields in the Vivado IDE and the User parameters (which can be viewed in the Tcl console).

#### Table 4-2: GUI Parameter to User Parameter Relationship

<table>
<thead>
<tr>
<th>GUI Parameter/Value(1)</th>
<th>User Parameter/Value(1)</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Type</td>
<td>Core_Type</td>
<td>DUC</td>
</tr>
<tr>
<td>Wireless Standard</td>
<td>Wireless_Standard</td>
<td>LTE</td>
</tr>
<tr>
<td>Channel Bandwidth</td>
<td>Channel_Bandwidth</td>
<td>5</td>
</tr>
<tr>
<td>Baseband Sample Rate</td>
<td>Baseband_Sample_Rate</td>
<td>7.68</td>
</tr>
<tr>
<td>RF Sample Rate</td>
<td>RF_Sample_Rate</td>
<td>122.88</td>
</tr>
<tr>
<td>Digital IF</td>
<td>Digital_IF</td>
<td>0 Hz</td>
</tr>
<tr>
<td>Number of Carriers</td>
<td>Number_of_Carriers</td>
<td>1</td>
</tr>
<tr>
<td>IF Passband</td>
<td>IF_Passband</td>
<td>5</td>
</tr>
<tr>
<td>Number of Antennas</td>
<td>Number_of_Antennas</td>
<td>1</td>
</tr>
<tr>
<td>Clock Frequency</td>
<td>Clock_Frequency</td>
<td>368.64</td>
</tr>
<tr>
<td>Input Data Width</td>
<td>Input_Data_Width</td>
<td>16</td>
</tr>
<tr>
<td>Output Data Width</td>
<td>Output_Data_Width</td>
<td>16</td>
</tr>
</tbody>
</table>
Chapter 4: Design Flow Steps

Table 4-2: GUI Parameter to User Parameter Relationship (Cont’d)

<table>
<thead>
<tr>
<th>GUI Parameter/Value(1)</th>
<th>User Parameter/Value(1)</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Interface Format</td>
<td>Data_Interface_Format</td>
<td>Separate_I_and_Q_signals</td>
</tr>
<tr>
<td>Separate I and Q signals</td>
<td>Separate_I_and_Q_signals</td>
<td></td>
</tr>
<tr>
<td>Combined complex signal with I and Q TDM</td>
<td>Combined_complex_signal_with_I_and_Q_TDM</td>
<td></td>
</tr>
<tr>
<td>RESET</td>
<td>RESET</td>
<td>False</td>
</tr>
<tr>
<td>Implementation Goal</td>
<td>Implementation_Goal</td>
<td>Optimize_For_Minimum_Area</td>
</tr>
<tr>
<td>Optimize For Minimum Area</td>
<td>Optimize_For_Minimum_Area</td>
<td></td>
</tr>
<tr>
<td>Optimize for Maximum Speed</td>
<td>Optimize_For_Maximum_Speed</td>
<td></td>
</tr>
<tr>
<td>BRAM Usage</td>
<td>BRAM_Usage</td>
<td>Auto</td>
</tr>
<tr>
<td>Programmable Carrier Frequencies</td>
<td>Programmable_Carrier_Frequencies</td>
<td>False</td>
</tr>
<tr>
<td>Programmable Carrier Gain Control</td>
<td>Programmable_Carrier_Gain_Control</td>
<td>False</td>
</tr>
<tr>
<td>Carrier 1(2) (under column &quot;Frequency (kHz)&quot;)</td>
<td>Carrier_Frequency_1</td>
<td>0</td>
</tr>
<tr>
<td>Carrier 1(2) (under column “Phase Offset (radians)”)</td>
<td>Carrier_Phase_Offset_1</td>
<td>0</td>
</tr>
</tbody>
</table>

1. Parameter values are listed in the table where the GUI parameter value differs from the user parameter value. Such values are shown in this table as indented below the associated parameter.
2. Carriers 2 thru 30 (when and whichever available) follow the same pattern of GUI field to user parameter as for Carrier 1.

Output Generation

For details, see the Vivado Design Suite User Guide: Designing with IP (UG896) [Ref 7].

Constraining the Core

This section contains information about constraining the core in the Vivado Design Suite.

Required Constraints

This section is not applicable for this IP core.

Device, Package, and Speed Grade Selections

This section is not applicable for this IP core.
Clock Frequencies

The core operates with a single synchronous clock. This is specified in the implementation tool constraints.

Clock Management

This section is not applicable for this IP core.

Clock Placement

This section is not applicable for this IP core.

Banking

This section is not applicable for this IP core.

Transceiver Placement

This section is not applicable for this IP core.

I/O Standard and Placement

This section is not applicable for this IP core.

Simulation

For comprehensive information about Vivado simulation components, as well as information about using supported third-party tools, see the Vivado Design Suite User Guide: Logic Simulation (UG900) [Ref 9].

Synthesis and Implementation

For details about synthesis and implementation, see the Vivado Design Suite User Guide: Designing with IP (UG896) [Ref 7].
Chapter 5

C Model Reference

This chapter provides information about the Xilinx LogiCORE IP DUC/DDC Compiler v3.0 bit accurate C model for 32-bit and 64-bit Linux and 32-bit and 64-bit Windows platforms.

The model consists of a set of C functions that reside in a shared library. Example C code is provided to demonstrate how these functions form the interface to the C model. Full details of this interface are given in DUC/DDC C Model Interface.

The model is bit accurate but not cycle-accurate. It produces exactly the same output data as the core on a sample-by-sample basis. However, it does not model the core clock cycle latency, its interface signals or its register map.

The DUC/DDC core contains a number of internal FIR filter stages. Each filter stage has a sample history pipeline that contributes to the overall sample latency of the core. The C model does not attempt to model the contents of these pipelines at initialization and after reset events. Bit accuracy is only guaranteed once these pipelines have flushed, which takes a number of samples equal to the total sample latency.

Unpacking and Model Contents

Unzipping the DUC/DDC C model ZIP file produces the directory structure and files shown in Table 5-1.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>duc_ddc_compiler_v3_0_bitacc_cmodel.h</td>
<td>Model header file</td>
</tr>
<tr>
<td>libIp_duc_ddc_compiler_v3_0_bitacc_cmodel.so</td>
<td>Model shared object library (Linux platforms only)</td>
</tr>
<tr>
<td>libIp_duc_ddc_compiler_v3_0_bitacc_cmodel.dll</td>
<td>Model dynamically linked library (Windows platforms only)</td>
</tr>
<tr>
<td>libIp_duc_ddc_compiler_v3_0_bitacc_cmodel.lib</td>
<td>Model library file for static linking (Windows platforms only)</td>
</tr>
<tr>
<td>duc_ddc_compiler_v3_0_bitacc_mex.cc</td>
<td>MATLAB® MEX function source</td>
</tr>
<tr>
<td>@duc_ddc_compiler_v3_0_bitacc</td>
<td>Matlab class directory</td>
</tr>
<tr>
<td>make_duc_ddc_compiler_v3_0_mex.m</td>
<td>MEX function compilation script</td>
</tr>
<tr>
<td>run_duc_ddc_compiler_v3_0_mex.m</td>
<td>MATLAB MEX function example script</td>
</tr>
<tr>
<td>run_bitacc_cmodel.c</td>
<td>Example source file</td>
</tr>
<tr>
<td>ducddc_testcases.dat</td>
<td>Test vectors for example source file</td>
</tr>
</tbody>
</table>
Chapter 5: C Model Reference

Installation

• On Linux platforms, ensure that the directory in which the file
  libIp_duc_ddc_compiler_v3_0_bitacc_cmodel.so is located appears as an
  entry in your LD_LIBRARY_PATH environment variable.

• On Windows platforms, ensure that the directory, including
  libIp_duc_ddc_compiler_v3_0_bitacc_cmodel.dll either:
    • Appears as an entry in your PATH environment variable, or
    • Is the directory in which you will run your executable that calls the DUC/DDC C
      model.

DUC/DDC C Model Interface

The Application Programming Interface (API) of the C model is defined in the header file
duc_ddc_compiler_v3_0_bitacc_cmodel.h. The interface consists of three
user-visible structures, ten functions to manipulate the model and perform simulation, and
four functions to assist in allocating input and output data structures. The header file also
contains some useful constants and type definitions.

Type Definitions

The interface contains the type definitions shown in Table 5-2.

Table 5-2: Type Definitions

<table>
<thead>
<tr>
<th>Type Definition</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef double xip_ducddc_data;</td>
<td>Defines the data type used to represent input and output samples</td>
</tr>
<tr>
<td>typedef int xip_ducddc_status;</td>
<td>Defines the data type used to represent status codes</td>
</tr>
<tr>
<td>typedef void (<em>msg_handler)(void</em> handle, int error, const char* msg);</td>
<td>Defines the interface to a message handler function</td>
</tr>
<tr>
<td>typedef struct _xip_ducddc_v3_0 xip_ducddc_v3_0;</td>
<td>Declares a handle type to refer to instances of the C model object</td>
</tr>
</tbody>
</table>
Chapter 5: C Model Reference

Structures

The interface contains the following structures.

Configuration

The `xip_ducddc_v3_0_config` structure specifies the configuration that should apply to the modelled core, as shown in Table 5-3.

**Note:** See Chapter 4, Customizing and Generating the Core for a full description of the parameters.

### Table 5-3: Configuration Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>char*</td>
<td>Instance name (arbitrary)</td>
</tr>
<tr>
<td>core_type</td>
<td>int</td>
<td>0 = DUC, 1 = DDC</td>
</tr>
<tr>
<td>ch_bandwidth</td>
<td>int</td>
<td>Channel bandwidth (1)</td>
</tr>
<tr>
<td>if_passband</td>
<td>int</td>
<td>IF passband (in MHz) (1)</td>
</tr>
<tr>
<td>digital_if</td>
<td>int</td>
<td>0 = Digital IF at 0Hz, 1 = Digital IF at FS/4</td>
</tr>
<tr>
<td>rf_rate</td>
<td>int</td>
<td>RF sample rate (in Hz) (1)</td>
</tr>
<tr>
<td>clock_rate</td>
<td>int</td>
<td>Clock speed (in Hz) (1)(2)</td>
</tr>
<tr>
<td>n_carriers</td>
<td>int</td>
<td>Number of carriers (1 - 30) (1)</td>
</tr>
<tr>
<td>n_antennas</td>
<td>int</td>
<td>Number of antennas (1, 2, 4 or 8)</td>
</tr>
<tr>
<td>din_width</td>
<td>int</td>
<td>Bit width (precision) of input data (11 - 17)</td>
</tr>
<tr>
<td>dout_width</td>
<td>int</td>
<td>Bit width (precision) of output data (11 - 18)</td>
</tr>
<tr>
<td>rounding_mode</td>
<td>int</td>
<td>0 = Round ties up (POSIX/MATLAB®), 1 = Round ties to even (IEEE) - as per core</td>
</tr>
</tbody>
</table>

1. Valid values for these fields are dependent on other parameters, as detailed in Chapter 4.
2. While the DUC/DDC C model is not cycle accurate, the implementation clock frequency must be known to correctly determine how the internal filter structure will be decomposed (which affects both filter coefficient values and latency).

Request

The `xip_ducddc_v3_0_data_req` structure is used to specify input complex sample data for the C model, as shown in Table 5-4.

### Table 5-4: Request Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>din_size</td>
<td>size_t</td>
<td>Number of (I,Q) sample pairs (per carrier, per antenna)</td>
</tr>
<tr>
<td>din_dim0</td>
<td>size_t</td>
<td>Number of antennas</td>
</tr>
<tr>
<td>din_dim1</td>
<td>size_t</td>
<td>Number of carriers</td>
</tr>
<tr>
<td>din_i</td>
<td>xip_ducddc_data*[][]</td>
<td>Real part of sample data</td>
</tr>
<tr>
<td>din_q</td>
<td>xip_ducddc_data*[][]</td>
<td>Imaginary part of sample data</td>
</tr>
</tbody>
</table>
The \texttt{din_i} and \texttt{din_q} arrays are two-dimensional arrays in which each element is a pointer to a sample buffer. The first array dimension is the number of antennas, and the second dimension is the number of carriers. The arrays are statically dimensioned to hold data for the maximum numbers of antennas and carriers (8 and 30 respectively). The number of valid entries in each dimension should be indicated in the \texttt{din_dim0} and \texttt{din_dim1} fields. Unused pointer entries in each dimension are ignored and should be filled with null pointers. This mechanism is illustrated in Figure 5-1. This example shows a configuration with six carriers and four antennas.

![Figure 5-1: Request Structure Example](image)

Allocation of the arrays is your responsibility. They may be allocated statically or dynamically. If they are allocated dynamically, then you are responsible for their deallocation. The buffers for every carrier and antenna should be at least as large as \texttt{din_size} samples. If they are larger, excess samples will be ignored.

The most efficient way to perform dynamic allocation of such a multi-dimensional buffer structure is to allocate one large buffer with enough sample storage for all carriers and antennas in use, then logically divide the allocated region into smaller blocks and compute pointers to the start of these blocks. The function \texttt{xip_ducddc_v3_0_alloc_data_req} performs this operation for data request structures. Its use is recommended, but not mandatory.

Input sample data is in double-precision format, and should be scaled to lie within the range -1.0 to (just less than) +1.0. The input data is expected to be quantized according to the \texttt{din_width} that was specified in the model configuration. If the model detects an
input value with excessive range or precision, it will saturate and/or round to the nearest valid value and issue a warning message.

For a DDC core, the input is a single sample stream per antenna, even if there is more than one carrier. For a DDC, only the first entry of the carrier dimension is used (that is, din_dim1 must be equal to 1).

For a DDC configured with a digital IF of FS/4, the input data is real-valued. In these cases, the model ignores the contents of the dout_q array.

**Response**

The xip_ducddc_v3_0_data_resp structure is used to specify output sample data from the C model, as shown in Table 5-5.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dout_size</td>
<td>size_t</td>
<td>Number of (I,Q) sample pairs (per carrier, per antenna) output</td>
</tr>
<tr>
<td>dout_max_size</td>
<td>size_t</td>
<td>Number of samples for which space is allocated</td>
</tr>
<tr>
<td>dout_clean</td>
<td>size_t</td>
<td>Number of samples which are clean (bit accurate)</td>
</tr>
<tr>
<td>dout_dim0</td>
<td>size_t</td>
<td>Number of antennas</td>
</tr>
<tr>
<td>dout_dim1</td>
<td>size_t</td>
<td>Number of carriers</td>
</tr>
<tr>
<td>dout_i</td>
<td>xip_ducddc_data*[]</td>
<td>Real part of sample data</td>
</tr>
<tr>
<td>dout_q</td>
<td>xip_ducddc_data*[]</td>
<td>Imaginary part of sample data</td>
</tr>
</tbody>
</table>

The dout_i and dout_q arrays are two-dimensional arrays in which each element is a pointer to a sample buffer, as previously described in detail for the Request structure. All comments regarding the dimensioning and allocation of arrays apply equally and equivalently to the response structure.

The dout_max_size field should be used by the caller to specify the amount of space that has been allocated in each output sample buffer. When processing a transaction, the C model will check that the output arrays have sufficient space for the resulting data before updating the dout_size field with the actual number of output samples that were written.

The dout_clean field is updated by the C model when a transaction is performed. It indicates how many samples in each output sample buffer are bit accurate. At initialization and after a reset, the simulated state of the sample histories of the DUC/DDC constituent FIR filters may differ from those that would be observed in the hardware. Until new input data has propagated through these sample histories, the outputs of the model and the hardware may not match.

When the value of dout_clean is the same as the value of dout_size, all output samples are bit accurate. If dout_clean is less than dout_size, only the last dout_clean output samples in the output buffer are bit accurate; earlier samples may not be bit accurate. If dout_clean is zero, none of the output samples are guaranteed to be bit accurate.
Output sample data is in double-precision format. It will be scaled to lie within the range -1.0 to (just less than) +1.0, and will be quantized according to the `dout_width` that was specified in the model configuration.

For a DUC core, the output is a single sample stream per antenna, even if there is more than one carrier. For a DUC, only the first entry of the carrier dimension is used (that is, `dout_dim1` will always equal 1).

**Functions**

The application programming interface contains the following functions. Most functions return a value of type `xip_ducddc_status` to indicate success (0, `XIP_DUCDDC_STATUS_OK`) or failure (1, `XIP_DUCDDC_STATUS_ERROR`).

**Get Version**

```c
const char*
xip_ducddc_v3_0_get_version(void);
```

This function returns a string describing the version of the DUC/DDC Compiler core modeled by the C library (for example “1.0beta,” “1.0,” “1.0patch1”).

**Get Default Configuration**

```c
xip_ducddc_status
xip_ducddc_v3_0_default_config(xip_ducddc_v3_0_config *config);
```

This function populates the `xip_ducddc_v3_0_config` configuration structure pointed to by `config` with the default configuration data for the DUC/DDC core. The data in this structure can be further customized to specify user parameters.

**Create Model Object**

```c
xip_ducddc_v3_0 *
xip_ducddc_v3_0_create(
    const xip_ducddc_v3_0_config *config,
    msg_handler handler,
    void *handle
);
```

This function creates a new instance of the model object based on the configuration data pointed to by `config`. 


The `handler` argument is a pointer to a function taking three arguments as previously defined in `Type Definitions`. This function pointer is retained by the model object and is called whenever the model wishes to issue a note, warning or error message. Its arguments are:

1. A generic pointer (void*). This will always be the value that was passed in as the `handle` argument to the create function.
2. An integer (int) indicating whether the message is an error (1) or a note or warning (0).
3. The message string itself.

If the `handler` argument is a null pointer, then the C model outputs no messages at all. Using this mechanism, you may choose whether to output messages to the console, log them to a file, or ignore them completely.

The create function returns a pointer to the newly created object. If the object cannot be created, then a diagnostic error message is emitted using the supplied handler function (if any) and a null pointer is returned.

**Reset Model Object**

```c
xip_ducddc_status
xip_ducddc_v3_0_reset(xip_ducddc_v3_0 *s);
```

This function resets the internal state of the DUC/DDC C model object pointed to by `s`. (This is not equivalent to a hardware reset.)

A reset causes the phases of any DDS and/or IF mixer components to return to their default starting states. It clears the sample histories of all constituent filters to zero, and causes any partially-accumulated sample data to be discarded. However, a reset does not alter any of the carrier frequencies, phase offsets or gain settings that may have been applied.

**Simulate**

```c
xip_ducddc_status
xip_ducddc_v3_0_data_do(
    xip_ducddc_v3_0 *s,
    xip_ducddc_v3_0_data_req *req,
    xip_ducddc_v3_0_data_resp *resp
);
```

This function applies the data specified in the request structure pointed to by `req` to the model object pointed to by `s`. Any output data resulting from the processing of this transaction is written into the response structure pointed to by `resp`.

The dimensions of the request and response buffer arrays must match the number of antennas and carriers as configured when the model object was created, as previously described in `Structures`.
The `dout_i` and `dout_q` buffers and the corresponding `dout_max_size` field of the response structure must be sufficiently large to accommodate the output samples that will be produced. If `dout_max_size` indicates that the buffers are too small, the transaction is ignored and an error message is issued; `dout_size` will be updated with the number of samples that would have been produced.

**Calculate Output Sizes**

```c
xip_ducddc_status
xip_ducddc_v3_0_data_calc_size(
    xip_ducddc_v3_0 *s,
    xip_ducddc_v3_0_data_req *req,
    xip_ducddc_v3_0_data_resp *resp
);
```

This function determines the output buffer sizes required for a call to the simulation function.

The array dimensions of the request and response structures pointed to by `req` and `resp` should be appropriately initialized as if for a call to `xip_ducddc_v3_0_data_do`, as should the `din_size` field of the request structure. The C model will update the `dout_size` field with the number of samples that would be returned by a simulation operation. You can then use this information to ensure that the output buffers are sufficiently large to accept this data.

If the structures supplied are invalid, the function issues an error message and returns a non-zero error code.

**Get Raster Parameters**

```c
xip_ducddc_status
xip_ducddc_v3_0_ctrl_get_raster(
    xip_ducddc_v3_0 *s,
    double *freq_raster,
    double *phase_raster,
    double *gain_step
);
```

This function retrieves information about the quantization of frequency, phase and gain information within the DUC/DDC core. The double-precision values pointed to by `freq_raster`, `phase_raster` and `gain_step` are overwritten with the corresponding information retrieved from the model object pointed to by `s`. If any of the pointers is null, no value is returned for that parameter.

- The frequency raster gives the resolution to which the carrier frequencies can be set, in Hz.
- The phase raster gives the resolution to which the carrier phases can be set, in radians.
- The gain step gives the minimum non-zero gain that can be programmed for a carrier. Valid gain values are all power-of-two multiples of this value, up to and including 1.0. This quantity is meaningful only for DUC configurations.
If the model object has only a single carrier, then no mixer will be present in the core and, consequently, raster information will not be available. Calling this function on such an object will result in an error.

**Set Carrier Parameters**

```c
xip_ducddc_status
xip_ducddc_v3_0_ctrl_set_carrier(  
    xip_ducddc_v3_0 *s,
    int index,
    double f,
    double phi,
    double beta
);
```

This function configures the carrier addressed by index within the model object pointed to by `s` with the frequency, phase and gain settings supplied in `f`, `phi` and `beta`.

The internal quantization of the floating-point values supplied was previously described in *Get Raster Parameters*. If the values provided are not already quantized, they will be adjusted by the C model. The actual values that were ultimately set can be retrieved by the *Get Carrier Parameters* function described in the following section.

If the model object has only a single carrier, then no mixer will be present in the core and, consequently, carrier parameters cannot be set. Calling this function on such an object will result in an error unless the values specified are 0.0 for the frequency and phase rasters and 1.0 for the gain step.

The default carrier parameters are a phase offset of 0.0, a gain of 1.0, and a set of frequencies symmetrically distributed around the 0Hz center point, spaced out according to the specified channel bandwidth.

**Get Carrier Parameters**

```c
xip_ducddc_status
xip_ducddc_v3_0_ctrl_get_carrier(  
    xip_ducddc_v3_0 *s,
    int index,
    double *f,
    double *phi,
    double *beta
);
```

This function retrieves the configuration of the carrier addressed by index within the model object pointed to by `s`, placing the frequency, phase and gain settings into the double-precision values pointed to by `f`, `phi` and `beta`.

If the model object has only a single carrier, then no mixer will be present in the core and, consequently, carrier parameters cannot be set. Calling this function on such an object will result in values of 0.0 for the frequency and phase rasters and a value of 1.0 for the gain step.
**Destroy Model Object**

```c
xip_ducddc_status
xip_ducddc_v3_0_destroy(xip_ducddc_v3_0 *s);
```

This function deallocates the model object pointed to by `s`.

Any system resources or memory belonging to the model object are released on return from this function. The model object becomes undefined, and any further attempt to use it is an error.

**Utility Functions**

**Allocate Data Request**

```c
xip_ducddc_status
xip_ducddc_v3_0_alloc_data_req(
    xip_ducddc_v3_0 *s,
    xip_ducddc_v3_0_data_req *r,
    unsigned n_samples
);
```

This function allocates buffers with space for `n_samples` input samples in a request structure pointed to by `r` suitable for use with the model object pointed to by `s`.

Allocation is performed as previously described in Structures. The `din_dim0`, `din_dim1` and `din_size` fields are filled in appropriately. Any pointers already present in the `din_i` and `din_q` arrays are overwritten. This function should be called only on an uninitialized request structure.

**Allocate Data Response**

```c
xip_ducddc_status
xip_ducddc_v3_0_alloc_data_resp(
    xip_ducddc_v3_0 *s,
    xip_ducddc_v3_0_data_resp *r,
    unsigned n_samples
);
```

This function allocates buffers with space for `n_samples` output samples in a response structure pointed to by `r` suitable for use with the model object pointed to by `s`.

Allocation is performed as previously described in Structures. The `dout_dim0`, `dout_dim1` and `dout_max_size` fields are filled in appropriately. Any pointers already present in the `dout_i` and `dout_q` arrays are overwritten. This function should be called only on an uninitialized response structure.
**Deallocate Data Request**

```c
xip_ducddc_status
xip_ducddc_v3_0_free_data_req(
    xip_ducddc_v3_0 *s,
    xip_ducddc_v3_0_data_req *r
);
```

This function deallocates a request structure allocated by the `alloc_data_req()` function described previously, and clears its constituent fields to an empty state. It should not be used to deallocate a request structure that was allocated in any other way.

**Deallocate Data Response**

```c
xip_ducddc_status
xip_ducddc_v3_0_free_data_resp(
    xip_ducddc_v3_0 *s,
    xip_ducddc_v3_0_data_resp *r
);
```

This function deallocates a response structure allocated by the `alloc_dataResp()` function described previously, and clears its constituent fields to an empty state. It should not be used to deallocate a response structure that was allocated in any other way.

**Compiling**

To compile code using the C model, the compiler requires access to the `duc_ddc_compiler_v3_0_bitacc_cmodel.h` header file.

This can be achieved in either of two ways:

- Add the directory containing the header file to the include search path of the compiler;
- Copy the header file to a location already on the include search path of the compiler.

**Linking**

To use the C model, the user executable must be linked against the correct libraries for the target platform.

**Linux**

The executable must be linked against the `libIp_duc_ddc_compiler_v3_0_bitacc_cmodel.so` shared object library.

Using gcc, linking is typically achieved by adding the following command line options:

```
-L. -lIp_duc_ddc_compiler_v3_0_bitacc_cmodel
```
This assumes that the shared object library is in the current directory. If this is not the case, the `-L` option should be changed to specify the library search path to use.

Using GCC, the provided example program `run_bitacc_cmodel.c` can be compiled and linked using the following command:

```
gcc -x c++ -I. -L. -lIp_duc_ddc_compiler_v3_0_bitacc_cmodel -o run_bitacc_cmodel run_bitacc_cmodel.c
```

**Windows**

The executable must be linked against the `libIp_duc_ddc_compiler_v3_0_bitacc_cmodel.dll` dynamic link library.

Depending on the compiler, the import library `libIp_duc_ddc_compiler_v3_0_bitacc_cmodel.lib` may be required.

Using Microsoft Visual Studio.NET, linking is typically achieved by adding the import library to the **Additional Dependencies** edit box under the **Linker** tab of **Project Properties**.

**Example**

The `run_bitacc_cmodel.c` file contains some example code to illustrate the basic operation of the C model.

**MATLAB**

A MEX function and MATLAB® software class are provided to simplify the integration with MATLAB. The MEX function provides a low-level wrapper around the underlying C model, while the class file provides a convenient interface to the MEX function.

**MEX Function**

Compiled MEX functions are provided for MATLAB R2009b for each platform. They can be found in the C model matlab directory and have MATLAB MEX platform extensions (for example, mexw32, mex64, mexglx and mexa64).

Source code for the MEX function is also provided so that it can be compiled for other MATLAB software releases if necessary. To compile the MEX function within MATLAB, change to the C model matlab directory and run the `make_mex.m` script.
MATLAB Class

The at theduc_ddc_compiler_v3_0_bitacc class handles the create/destroy semantics on the C model. An instance of the class can exist in two states – empty and created. An empty instance has not yet been attached to an instance of the C model and so cannot be used to simulate. A created instance has been attached to a C model instance and can simulate.

The class provides the following methods:

**Constructor**

\[
\text{[model]} = \text{duc_ddc_compiler_v3_0_bitacc,} \\
\text{[model]} = \text{duc_ddc_compiler_v3_0_bitacc(config),} \\
\text{[model]} = \text{duc_ddc_compiler_v3_0_bitacc(field, value [, field, value]*)}
\]

The first version of the class constructor method constructs an empty model object that can be created later using the create method. The second version constructs a fully-created model object from a structure that specifies the configuration parameter values to use. The third version is the same as the second, but allows the configuration to be specified as a series of (parameter name, value) pairs rather than a single structure.

The names and valid values of configuration parameters are identical to those previously described in Configuration.

**Get Version**

\[
\text{[version]} = \text{get_version(model)}
\]

This method returns the version string of the C model library used.

**Get Configuration**

\[
\text{[config]} = \text{get_configuration(model)}
\]

This method returns the current parameters structure of a model object.

If the model object is empty, the method returns the default configuration. If the model object has been created, the method returns the configuration parameters that were used to create it.

**Create**

\[
\text{[model]} = \text{create(model, config),} \\
\text{[model]} = \text{create(model, field, value [, field, value]*)}
\]

This method creates an instance of the C model using the supplied configuration parameters and attaches it to the model object. If the object has already been created, the existing model is automatically destroyed and a new one created and attached in its place.
Chapter 5: C Model Reference

The first version creates a model object from a structure that specifies the configuration parameter values to use. The second version is the same as the first, but allows the configuration to be specified as a series of (field, value) pairs rather than a single structure.

The names and valid values of configuration parameters are identical to those previously described in the Configuration subsection of Structures.

**Destroy**

```c
[model]=destroy(model)
```

This method destroys any C model instance attached to the model object.

If the model object is attached to a C model instance, the instance is destroyed and the model object state updated to empty. It is safe to call the destroy method on an empty model object. After a call to the destroy method, the model object will always be empty.

**Created**

```c
[created]=is_created(model)
```

This method returns true if the model object is in the created state, and false if it is in the empty state.

**Simulate**

```c
[model, resp]=simulate(model, req)
[model, resp]=simulate(model, field, value [, field, value]*)
```

This method processes a data transaction on the model object.

The first version uses a structure to pass the input data into the model. The second version allows the data to be supplied using (field, value) pairs.

The request is a structure with the following members:

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| din    | 3-dimensional complex scalar array | Input sample data (combined I and Q)  
• First dimension is sample index  
• Second dimension is antenna index  
• Third dimension is carrier index |
The response is a structure with the following members:

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| dout   | 3-dimensional complex scalar array | Output sample data (combined I and Q)  
• First dimension is sample index  
• Second dimension is antenna index  
• Third dimension is carrier index |

**Get Raster**

\[
[raster] = \text{get}_\text{raster}(\text{model})
\]

This method returns the raster parameters of the model object. The output is a structure with the following members:

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>freq_raster</td>
<td>Real scalar</td>
<td>Frequency raster, in Hz</td>
</tr>
<tr>
<td>phase_raster</td>
<td>Real scalar</td>
<td>Phase raster, in radians</td>
</tr>
<tr>
<td>gain_step</td>
<td>Real scalar</td>
<td>Minimum non-zero carrier gain (dimensionless)</td>
</tr>
</tbody>
</table>

**Get Carrier**

\[
[\text{carrier}] = \text{get}_\text{carrier}(\text{model, index})
\]

This method returns the parameters for the carrier with the specified index within the model object. The output is a structure with the following members:

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>Real scalar</td>
<td>Carrier frequency, in Hz</td>
</tr>
<tr>
<td>phi</td>
<td>Real scalar</td>
<td>Carrier phase offset, in radians</td>
</tr>
<tr>
<td>beta</td>
<td>Real scalar</td>
<td>Carrier gain (dimensionless)</td>
</tr>
</tbody>
</table>

**Set Carrier**

\[
[\text{model}] = \text{set}_\text{carrier}(\text{model, index, carrier_info})
\]

\[
[\text{model}] = \text{set}_\text{carrier}(\text{model, index, field, value [,field, value]*})
\]

This method configures the parameters for the carrier with the specified index within the model object.

The first version takes a structure of the form returned by the Get Carrier. The second version allows the carrier settings to be specified individually as (field, value) pairs.
Chapter 5: C Model Reference

Installation

To use the MEX function and class, the files in the C model matlab class directory must be present on the MATLAB search path. This can be achieved in either of two ways:

- Add the C model matlab directory to the MATLAB search path (see the MATLAB addpath function), or
- Copy the files to a location already on the MATLAB search path.

As with all uses of the C model, the correct C model libraries also need to be present on the platform library search path (that is, PATH or LD_LIBRARY_PATH).

Example

The run_duc_ddc_compiler_v3_0_mex.m script is provided as an example of how to use the MATLAB class. It passes some test data samples through the model and verifies that the results are as expected.
Appendix A

Migrating and Upgrading

This appendix contains information about migrating a design from ISE® to the Vivado® Design Suite, and for upgrading to a more recent version of the IP core. For customers upgrading in the Vivado Design Suite, important details (where applicable) about any port changes and other impact to user logic are included.

Migrating to the Vivado Design Suite

For information about migrating to the Vivado Design Suite, see the ISE to Vivado Design Suite Migration Guide (UG911) [Ref 10].

Upgrading in the Vivado Design Suite

This section provides information about any changes to the user logic or port designations that take place when you upgrade to a more current version of this IP core in the Vivado Design Suite.

Parameter Changes

No changes.

Port Changes

No changes.

Other Changes

No 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 DUC/DDC Compiler, 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 DUC/DDC Compiler. 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 proper 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 DUC/DDC Compiler

AR: 54476

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 DUC/DDC Compiler core 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 lab tools 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) [Ref 11].
Appendix C

Additional Resources and Legal Notices

Xilinx Resources

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

References

These documents provide supplemental material useful with this product guide:

1. AMBA 3 APB Protocol, v1.0 (ARM IHI 0024B)
2. 3GPP TS 36.104, 3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access (E-UTRA); Base Station (BS) radio transmission and reception; (Release 9)
3. 3GPP TR 25.105, 3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Base Station (BS) radio transmission and reception (TDD) (Release 9)
4. 3GPP TR 25.104, 3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Base Station (BS) radio transmission and reception (FDD) (Release 9)
5. AMBA® AXI4-Stream Protocol Specification (ARM IHI 0051A)
10. ISE to Vivado Design Suite Migration Guide (UG911)
12. FIR Compiler Product Guide (PG149)
Appendix C: Additional Resources and Legal Notices

13. Designing Efficient Wireless Digital Up and Down Converters Leveraging CORE Generator and System Generator (XAPP1018)


15. 3GPP LTE Digital Front End Reference Design (XAPP1123)

Revision History

The following table shows the revision history for this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>11/18/2015</td>
<td>3.0</td>
<td>Added support for UltraScale+ families.</td>
</tr>
<tr>
<td>11/19/2014</td>
<td>3.0</td>
<td>• Added link to resource utilization figures.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added User Parameter table.</td>
</tr>
<tr>
<td>12/18/2013</td>
<td>3.0</td>
<td>• Added support for UltraScale™ architecture.</td>
</tr>
<tr>
<td>10/02/2013</td>
<td>3.0</td>
<td>• Revision number advanced to 3.0 to align with core version number 3.0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• C Model updates.</td>
</tr>
<tr>
<td>03/20/2013</td>
<td>1.0</td>
<td>Initial release as a product guide. This document derived from DS766,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LogiCORE IP DUC/DDC Compiler Data Sheet and UG724, LogiCORE IP DUC/DDC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Compiler C Model User Guide.</td>
</tr>
</tbody>
</table>

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 http://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 http://www.xilinx.com/legal.htm#tos.

© Copyright 2013–2015 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. All other trademarks are the property of their respective owners.