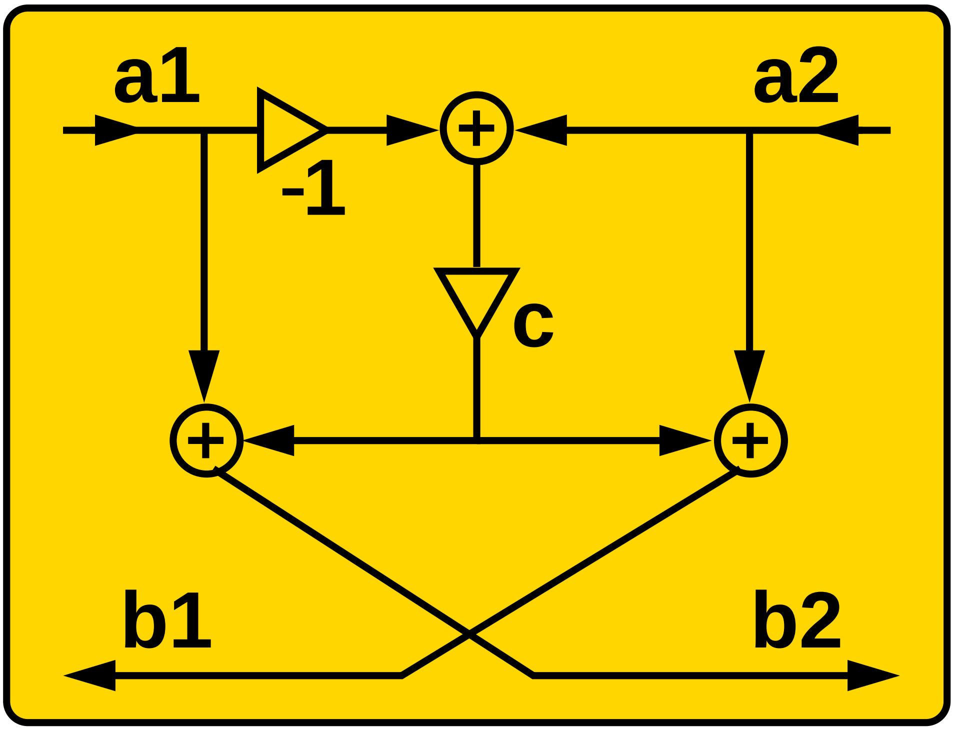

Virtual RF IP

Ingenieurbüro BAY9
Nordstr. 40
01099 Dresden / Germany
+49 351 7924700
info@bay9.de
http://www.bay9.de

Contents

1 Introduction

1.1 Scope

This document describes Ingenieurbüro BAY9's Virtual RF IP core.

1.2 Delivery file structure

The delivery contains 5 subdirectories:

- dat: Boot data file and memory contents / start command files
- doc: This documentation
- msg: Message ID definitions and example functions of control message handlers in Matlab/Octave.
- src: Verilog sources
  - src/common: Miscellaneous general purpose modules
  - src/modules: Signal processing and control modules
  - src/top: The top module *vrf.v* and a corresponding instance definition *vrf_0.v* that can be included into the Verilog file containing the VRF IP core
- tst: Test build examples
  - tst/altera: Build script and configuration files for Altera-Quartus
  - tst/xilinx: Build script and configuration files for Xilinx-Vivado
  - tst/xilinxIse: Build script and configuration files for legacy Xilinx-Ise
  - tst/m: Matlab/Octave example setup test scripts
  - tst/wlan: Verilog source files, boot data, and Matlab/Octave scripts needed to connect two 802.11a/p IP cores to the VRF IP core.

1.3 Features

The BAY9 Virtual RF (VRF) is an IP core written in Verilog, that allows to emulate most system aspects of a typical RF transmission. The emulation covers TX and RX impairments of typical real world RF circuits (noise, IQ imbalance, PA non-linearities, clock/frequency offset, etc...), RF features such as gain control, and channel impairments such as multipath fading.

When connected to a physical layer (PHY) core, the VRF IP core replaces a real RF device with respect to most aspects between TX-DAC output and RX-ADC input, thereby allowing verification and test of the PHY implementation in (almost) real time. Consequently, the IP core can be used to speed up creation of bit / frame error rate curves and for parameter optimization.

The core does not use any vendor specific HW blocks and runs on Altera and Xilinx FPGAs.

Main features summary:

- Control via message API
- Internal data generation and checking
- Control message forwarding from/to PHY core(s)
- Covers the following aspects of typical RF chains
  - DAC simulation, i.e., requesting p samples during q clock cycles
  - TX-IQ imbalance
  - TX-DC offset
  - PA amplitude compression and phase shift
  - FIR type fading channel
  - Frequency offset
  - Clock offset

- Additive White Gaussian Noise
- RX gain control with arbitrary definition of
    - Attenuation
    - Pin settings
    - Delay
- RX-IQ imbalance
- RX-DC offset
- RX-ADC input saturation

1.4 Overview

1.4.1 System

The IP core overview is shown in figure 1. The complete system can be controlled by messages sent via UART, cf. section 3.3.3, or via the generic parallel interface called *ctrl*, cf. section 3.3.2.

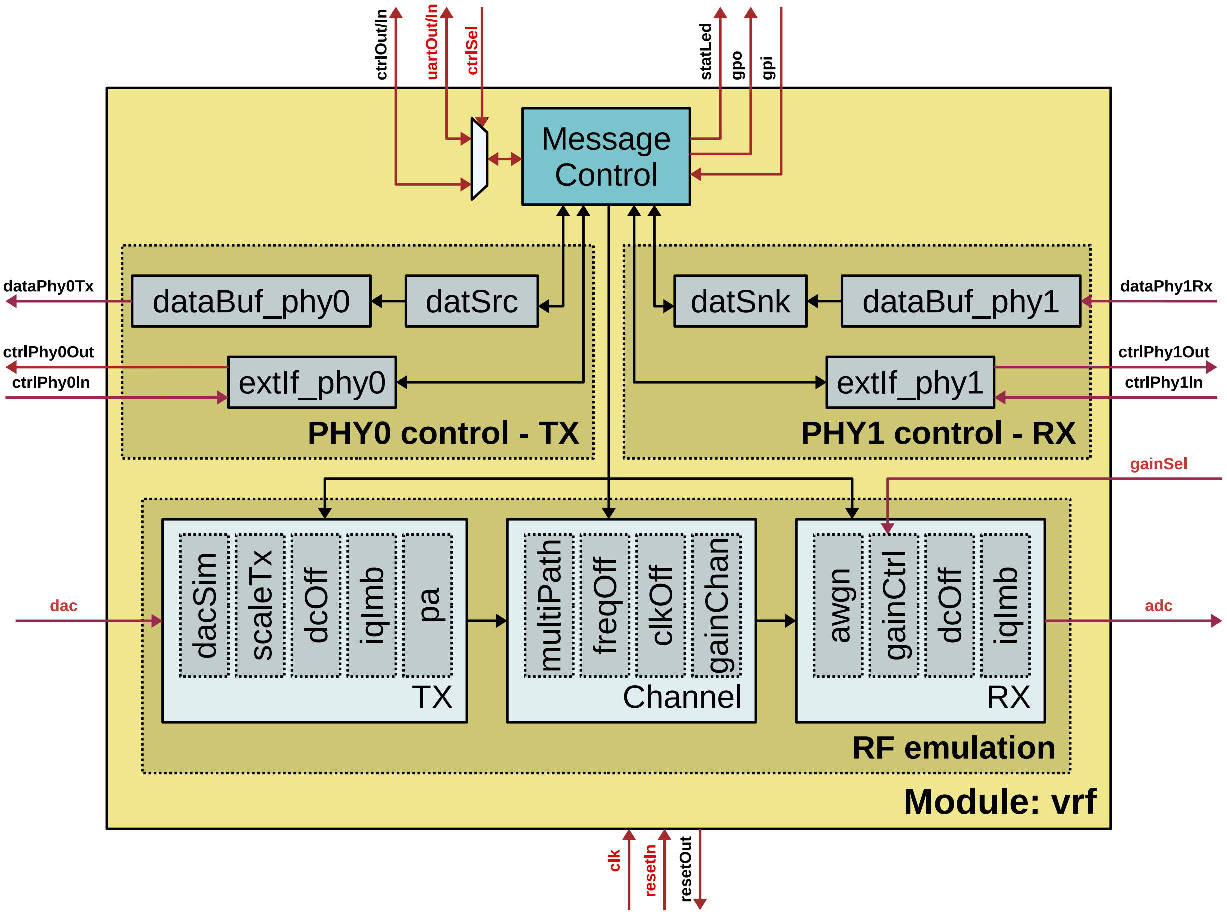

Fig. 1: VRF system overview

The IP core is split into 2 main parts

- RF + channel emulation
- Physical layer control

Basic functionality can be achieved by using only the signals marked in red in figure 1. Status LEDs provide feedback about the boot process and current state of the IP core.

1.4.2 RF / channel emulation

For RF / channel emulation, it is necessary to connect the DAC output of a physical layer core providing a TX signal (PHY0-TX-DAC output) to the VRF *dac* input, and the VRF *adc* output to the ADC input of a physical layer core receiving/decoding the RX signal (PHY1-RX-ADC input), while simultaneously providing a proper *gainSel* signal. RF/channel emulation is explained in more detail in section 4.

1.4.3 PHY layer control

Physical layer control provides basic functionality for sending data/messages to or receiving data/messages from TX + RX physical layer cores. It also includes data buffers and comparison routines. These modules may or may not be used depending on customer requirements. PHY control is explained in detail in section 5. If not needed, the buffer memory sizes should be set as small as possible to save FPGA resources, cf. section 5.3.1.

Additionally, general purpose input/output signals *gpi/gpo* can be used to control the PHY cores, e.g. for reset.

1.5 Evaluation vs. full featured version

In the evaluation version, only the following features are enabled:

- TX scaling
- Channel gain
- AWGN
- Gain control
- PHY control

All other features, DAC sampling simulation, DC offset, IQ imbalance, power amplifier, multipath channel, and clock/frequency offset are disabled. The corresponding source code is replaced by empty placeholders.

2 Getting started

2.1 Overview

This section provides an example how to:

- Synthesize the VRF IP core together with other physical layer TX and RX cores
- Boot and configure the VRF core (and possibly the PHY cores)
- Transmit data packages via the core and evaluate reception results

In order to ease these initial steps, this package additionally provides:

- An evaluation version of an 802.11a/p physical layer IP core
- An example Verilog module *vrfX* where the VRF core is connected to a physical layer TX and RX
- Bash scripts to synthesize module *vrfX* using Altera Quartus, Xilinx Vivado, and Xilinx ISE
- Example control message handlers in Octave/Matlab to access the VRF and the physical layer cores
- An example Octave/Matlab backend function to send messages to the FPGA via UART

None of the above is needed to use the VRF IP core. Rather, the core will typically be connected a customer specific physical layer core, maybe use control messages implemented in C, Perl, etc instead of Octave/Matlab, and possibly be accessed via the generic parallel interface instead of the UART.

Throughout this section however, extensive use will be made of the supplementary functions to get the *vrfX* example module running. Module *vrfX* is shown in figure 2 and explained in more detail in section 2.3.2 below.

2.2 Prerequisites

Build scripts and message control functions are provided for Bash and GNU/Octave, respectively. They will run equally on Linux or on Microsoft Windows with Cygwin installed.

Assuming the HW setup given in figure 3, a USB-to-UART converter should be attached to the PC. Default UART speed settings are up to 2 Mbaud, which is typically provided by these converters.

Currently tested software versions are

- Octave 4.0.2
- Altera Quartus 16.0
- Xilinx Vivado 2015.04
- Xilinx ISE 14.7 (legacy)
- Cygwin 1.7.2 on Windows 7

All examples below run with the free/web edition versions of Altera Quartus and Xilinx Vivado/ISE.

2.3 Example module *vrfX*

2.3.1 Description

For basic testing it is necessary to connect the VRF core to a physical layer transmitter and receiver. An example setup is displayed in figure 2. The VRF core and 2 WLAN 802.11a/p PHY cores (eval version only) are embedded into the overall test example module *vrfX*. Modules instances *wlan_0* and *wlan_1* serve as TX and RX, respectively. Submodule *pllX* must be adapted/replaced by a PLL fitting the external clock and the actual FPGA.

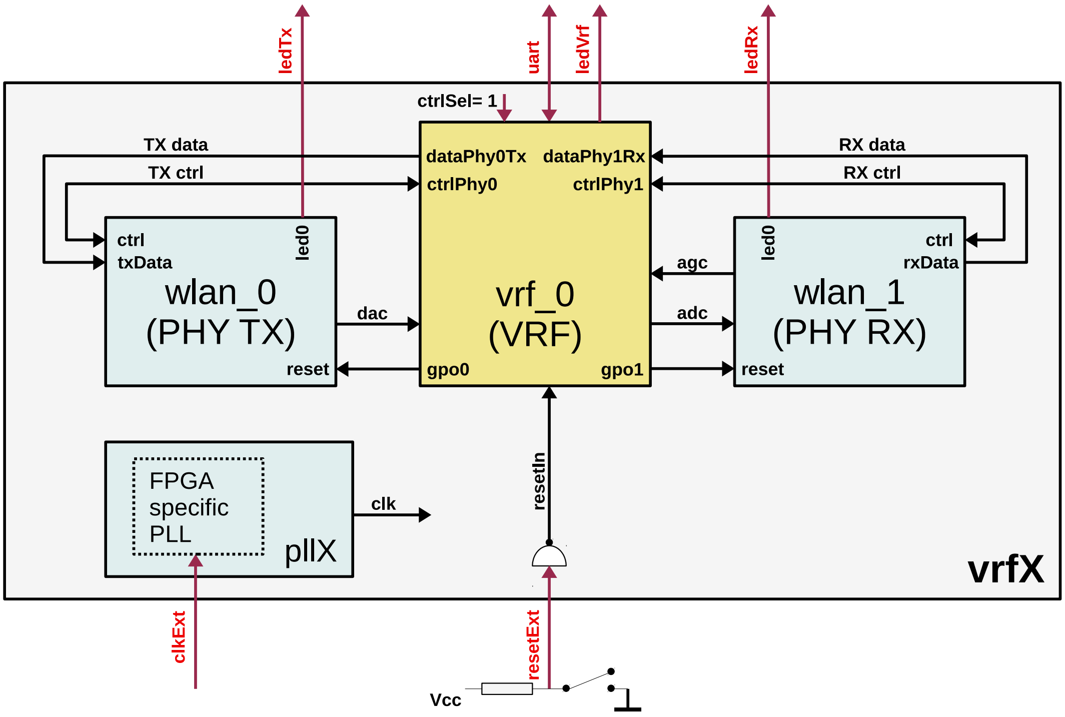

Fig. 2: Example test setup

2.3.2 Minimum setup

All components of *vrfX* are controlled by a single UART, see section 3.3.3. The VRF core is controlled directly, while the 2 WLAN PHY cores are controlled via message forwarding, cf. section 3.3.4 and figure 1. For data packet transmission, reception, and comparison, the VRF PHY control is used, cf. section 5 for details. GPO signals 0 and 1 reset the WLAN PHY cores. Status LEDs indicate booting of the VRF and the WLAN PHY cores.

2.3.3 Build

Altera-Quartus / Xilinx-Vivado

In order to build with Altera-Quartus or Xilinx-Vivado, go to directory *./tst/(altera|xilinx)* and run script *./(altera|xilinx)Compile*. Synthesis results are in the corresponding directory *./tst/(altera|xilinx)/out*. Build examples

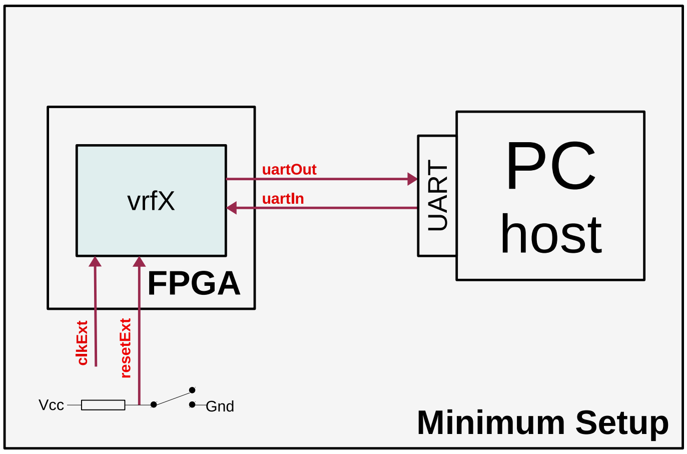

Fig. 3: Minimum hardware setup

are provided for

- Artix 7 XC7A200TFBG484-2, 80 MHz clock, UART speed 2 MHz
- Cyclone IV EP4CE115F29, 40 MHz clock, UART speed 1 MHz

Configuration files can be found in *./tst/(altera|xilinx)/cfg*. Before building for your own setup, you might want to change:

- The FPGA device to build for
- Physical PIN connections
- Module *./tst/(altera|xilinx)/src/pllX.v* for clock generation
- The UART divider depending on baud rate and clock speed, see section 3.3.3

Xilinx-ISE

There is also an example script for legacy Xilinx-ISE. Because ISE has problems to synthesize for 80 MHz on Artix 7, the timing constraints are relaxed here. Furthermore, synthesis results are not tested anymore. Please use ISE only in case you need to synthesize for an older device not supported by Vivado.

Board setup

The Xilinx example runs on a ZTEX-2.16 board (+ debug extension) using

- uart(Gnd|In|Out): Pins D30/29/28
- clkExt: 48 MHz on board FX clock
- resetExt: Switch S1-10
- ledVrf/Tx/Rx: LED1-1/9/10

The Altera example runs on a Terasic DE2 115 board using

- uart(Gnd|In|Out): Pins EX_IO[0/1/2]
- clkExt: 50 MHz on board crystal
- resetExt: Push button KEY0
- ledVrf/Tx/Rx: Green LEDG[0/6/7]

2.3.4 UART access configuration

Function *./tst/m/uartInit.m* is used for configuration of UART access from the PC host. Access is provided via the Linux device file interface (*/dev/ttyXYZ*), also available under Windows/Cygwin. The function must be edited and adapted to the actual device file name, typically */dev/ttyUSBx* under Linux and */dev/ttySx* under Windows/Cygwin, and the UART speed. Under Linux, make sure you have read/write access to the */dev/ttyUSBx* device file.

2.3.5 Reset

Signal *resetExt* of test module *vrfX* signal is active low. Note that this is different from the internal VRF core signal *resetIn* which is active high, cf. figure 2. A short pulse *resetExt* = 0 resets the system. See also section 3.1 for details.

2.4 Booting and basic messages

2.4.1 Booting and version

After reset, the VRF core is in boot mode. The contents of the boot data file *./dat/boot.bin* needs to be transferred via the UART control interface. This can be accomplished in Octave by calling the special control file *./msg/msgVrfBootReq.m*:

```
$ cd ./tst/m
$ octave --traditional --quiet
>> def_MsgId_vrf;
>> msgVrfBootReq;
Booting OK;
>> msgVrfVersionReq;
Version A.B.C
>>
```

Successful boot is indicated by

- switching on *ledVrf*
- a *BootCfm* message sent from the IP core via the (UART) control interface, cf. section 6.3.29

The *BootCfm* must be read by the host, which is done in *msgVrfBootReq.m*. In the example above, an additional *VersionReq* message (see section 6.3.27 and section 6.3.54) is sent.

After booting, the VRF core is in operation mode. All control messages can be used. Rebooting is only possible after another reset. Details about messages are described in section 6. Matlab/Octave messages example message handlers are located in directory *./msg*.

See also section 3.2 for details and other possibilities to initialize memories and start the system.

2.4.2 Configuration

Once the connection the VRF core is successfully established, the PHY control part of module *vrfX* can be used to create TX frames at *wlan_0*, pass the samples through the VRF core *vrf_0*, and receive the frames at *wlan_1*.

Script *setupVrf.m* first boots and configures the VRF core using the following messages:

msgVrfResetReq:	Reset VRF core (SW reset only)
msgVrfBootReq:	Boot VRF core (and read version number)
msgVrfOnOffReq:	Switch VRF core to off state (paranoia setting)
msgVrfCfgTxInpScReq:	Set input scaling according to the TX WLAN core DAC output
msgVrfCfgChGainReq:	Set the channel gain to -100 dB initially
msgVrfCfgRxGainTblReq:	Set an arbitrary gain table to emulate the RF gain control feature
msgVrfOnOffReq:	Switch VRF core to on state

The VRF core now behaves like a real RF with no input at the antenna (due to the -100 dB channel gain). It produces white gaussian noise at its output *adc* depending on the *gainSel* input signal. This signal selects a virtual "RF gain" from the internal table (defined earlier by *msgVrfCfgRxGainTblReq*).

Finally, the 802.11a WLAN cores are reset and configured. Script *setupVrf.m* uses *msgVrfCfgGpoReq.m* to reset the cores via the general purpose outputs (GPOs). After reset the cores are booted and configured via subfunctions *setupWlan.m* and *setupWlanAgc.m* (not discussed in detail here).

For each core, message *LedBlinkReq* is called directly after booting to indicate the activity.

```
>> setupVrf
-------------------------------------
Reset + boot VRF system ...
Booting VRF OK
Version A.B.C
Set RF off
Set input scaling
Set channel gain to -100 dB initially
Set RF gain delay and RF gain step table
Set RF on
```

```
-------------------------------------
Boot + setup WLAN-TX
-------------------------------------
Boot WLAN
Booting WLAN OK
Version X.Y.Z
Set band selection
Set TX timing
Set TX backoff
Set RX time tracking parameters
Set normal ACQ threshold + normal (automatic) AGC mode
Set CCA to 5500us (max length), 0 dB offset
-------------------------------------
Boot + setup WLAN-RX
-------------------------------------
Boot WLAN
Booting WLAN OK
Version X.Y.Z
Set band selection
Set TX timing
Set TX backoff
Set RX time tracking parameters
Set normal ACQ threshold + normal (automatic) AGC mode
Set CCA to 5500us (max length), 0 dB offset
-------------------------------------
Calibrate WLAN-RX AGC table
-------------------------------------
Set RX AGC gain table for calibration
Set low ACQ threshold
Send RX request messages 10x + evaluate input level
-------------------
Set attnIdx -> 0
Input amplitude (mean) = 25.7
-------------------
Set attnIdx -> 1
Input amplitude (mean) = 18.2
-------------------
Set attnIdx -> 2
Input amplitude (mean) = 12.9
-------------------------------------
Attenuation used: Index = 1, MeanAmp = 18.2
Set RX AGC gain table again after calibration
-------------------------------------
Set normal ACQ threshold + normal (automatic) AGC mode
-------------------------------------
```

2.4.3 Data transmission

Function *testTxRx11g.m* initiates transmission of 802.11a OFDM frames and checks correct reception via the VRF core's PHY control features. For each transmitted frame it uses

msgVrfCfgChGainReq:	Set the VRF gain to yield the desired carrier-to-noise ratio
msgVrfDatBufRxStartReq:	Prepare VRF RX buffer to receive data from RX PHY core
msgVrfDatBufTxFillReq:	Fill VRF TX buffer with data to send to TX PHY core
msgVrfDatBufTxStartReq:	Enable VRF TX buffer data transmission
msgWlanRxRun1:	Start WLAN RX PHY core acquisition

msgWlanTxRun:	Start WLAN TX PHY core OFDM frame transmission
msgWlanRxRun2:	Read reception results of the OFDM frame from the RX PHY core
msgVrfDatBufRxCheckReq:	Check VRF RX buffer data

Additionally, function *testTxRx11g.m* prints an evaluation line for each frame transmitted and finally statistics about the reception results. In the example below 10 frames of mode 7 (64-QAM, R=3/4), length 1000 bytes, C/N = 19 dB are transmitted:

```
>> testTxRx11g(10, 7, 1000, 19, 1);

+--------+------+---+------+-----+-----+---------+---+
|   No   |  C2N | M |  Len | Amp | AGC |  fOffHz | C |
+--------+------+---+------+-----+-----+---------+---+
|      0 | 19.0 | 7 | 1000 | 147 |   0 |    -305 | 1 |
|      1 | 19.0 | 7 | 1000 | 104 |   3 |    -153 | 1 |
|      2 | 19.0 | 7 | 1000 | 101 |   3 |   +2060 | 1 |
|      3 | 19.0 | 7 | 1000 | 103 |   3 |    +229 | 0 |
|      4 | 19.0 | 7 | 1000 | 102 |   3 |   -1144 | 0 |
|      5 | 19.0 | 7 | 1000 | 105 |   3 |    -229 | 1 |
|      6 | 19.0 | 7 | 1000 | 148 |   0 |     +76 | 0 |
|      7 | 19.0 | 7 | 1000 | 102 |   3 |   +1144 | 0 |
|      8 | 19.0 | 7 | 1000 | 101 |   3 |    -610 | 0 |
|      9 | 19.0 | 7 | 1000 | 103 |   3 |    +153 | 0 |
------------------------------------------------------

  CRC OK:   60.0%
  CRC Err:  40.0%
  Hdr Err:   0.0%
  No  Acq:   0.0%
```

3 Description – General aspects

3.1 Reset

3.1.1 Overview

The VRF core can be reset by a HW signal or a SW message. In addition, an internal self reset feature is provided that normally puts the core into reset state directly after loading the FPGA.

3.1.2 Hardware reset

HW reset is provided via signal *resetIn* in figure 1 and is active high. Note that in example setup *vrfX* in figure 2 the signal *resetExt* is inverted (thus active low) to be compliant with typical board setups.

It is necessary to wait 128 cycles after HW reset before booting the core, because the internal reset signal is delayed and kept active during that time. Alternatively, signal *resetOut* (cf. figure 1) can be monitored, *resetOut* = 1 indicates that the internal reset is still active.

3.1.3 Software reset

SW reset is triggered by a *ResetReq* message sent via the control interface, cf. section 3.3.2 and section 6.3.2. SW reset normally only works when the core is ready to receive and evaluate messages, i.e., in normal operation mode after booting. If the systems hangs for some reason, e.g. misconfiguration or similar, then the SW reset might not work.

If an additional *ResetReq* is sent while the core is still in boot mode, i.e., directly after a previous SW or HW reset or loading of the FPGA, then it is captured by the IP core control interface. Because the core cannot handle messages before being booted/started, detection of the reset is timer based in this case.

In boot mode, the core normally expects packets of 2 data words via the control interface for booting and configuration. A *ResetReq* is only a single data word. Therefore, if the second data word is missing, a watchdog triggers reset internally about 2^{22} clock cycles after the *ResetReq* (or any single word) has been received. One needs to wait at least 2^{22} cycles or 0.05 ms @ 80MHz after using a SW reset, or monitor signal *resetOut* for activity.

3.1.4 Internal self reset

Directly after loading the FPGA, the VRF core tries to reset itself. This feature is based on an internal 8-bit counter whose initial state is assumed to be random or all 0s or all 1s directly after loading. This feature might be unreliable on some FPGAs.

3.1.5 Boot confirm message

The SW reset request has no corresponding confirm. Instead, a *BootCfm* message (section 6.3.29) is sent from the VRF core to the host independent of the type of reset. This confirm is sent only after booting and starting the core, not directly after reset.

3.2 Booting

3.2.1 Using the boot file

After reset, the system is in boot mode. The contents of the boot data file *./dat/boot.bin* needs to be transferred via the control interface (see section 3.3 below). Transferring this file initializes the internal controller memories and starts the controller. File *boot.bin* is also provided as *boot.hex*. Successful boot is indicated by

- a 01010101 pattern at the LED outputs
- a *BootCfm* message sent from the IP core to the host via the control interface, cf. section 6.3.29

The *BootCfm* message must be read by the host. After booting, the system is in operation mode. Rebooting is only possible after another reset.

Section 2.4.1 provides an example how to boot via UART using Octave/Matlab function *msgVrfBootReq.m*. Despite its name, this function is not really a request message. Instead, it transfers the boot data to the IP core and reads the *BootCfm*.

Another option is to transfer *boot.bin* via the command line, e.g., *cat boot.bin > /dev/ttyUSB0*. See also section 3.3.3 for proper UART init in the latter case.

3.2.2 Using the memory init files

Alternatively to section 3.2.1, the memory contents files *X16_NNN_(P|D)ram.hex* can be used if the controller memory is replaced with preinitialized FPGA block memory. In this case, only the 2 words given in file *X16StartCmd.hex* need to be transferred via the control interface to start the IP core operation.

3.3 Control interface

3.3.1 Interface selection

Control interface selection is made via the *ctrlSel* input. Setting *ctrlSel* = 0 selects the *ctrl* lines, while *ctrlSel* = 1 selects the UART. Both interfaces are functionally equivalent. All control data comes in words of 16 bits.

3.3.2 Ctrl

The *ctrl* interface is a generic interface with 16 parallel data lines each for input and output (*ctrlIn*, *ctrlOut*), and the corresponding handshake signals described in section 3.5. It is much faster than the UART and can either be used directly, or adapted to other interface types like SPI etc.

3.3.3 UART

The *uartIn* and *uartOut* signals are used to receive and transmit UART data in 8N1 mode (8 bit, no parity, 1 stop bit). There is no handshaking. Internal processing of the core is sufficiently fast to receive data from *uartIn*, the *uartOut* line must be handled accordingly by the host. In order to handle the 16 bit control data, the UART uses little-endian format (low byte first).

The default UART speed is 2 Mbaud for 80 MHz clock frequency. Configuration of parameter *VrfUartDivider_C* in file *./src/common/instances/def_Const_vrf.v* allows to change the speed. It must be set to

$$VrfUartDivider\_C = \mathrm{round}(f_{clk}/f_{baud}) - 1 \tag{1}$$

Function *./tst/m/uartInit.m* contains the default settings for the UART device file and the UART speed. It also contains the proper settings to use the UART via the Linux or Windows/Cygwin device file interface (/dev/ttyS0, /dev/ttyUSB0 or similar). Please edit this function to adjust the UART device file and speed.

3.3.4 Message handling + forwarding

Messages from the UART or *ctrl* interface can be forwarded to connected physical layer IP cores via inputs / outputs *ctrlPhy(0|1)(In|Out)*, cf. figure 1. The exact mechanism of message forwarding is described in section 6.1 and section 6.2.

3.4 Control messages and configuration

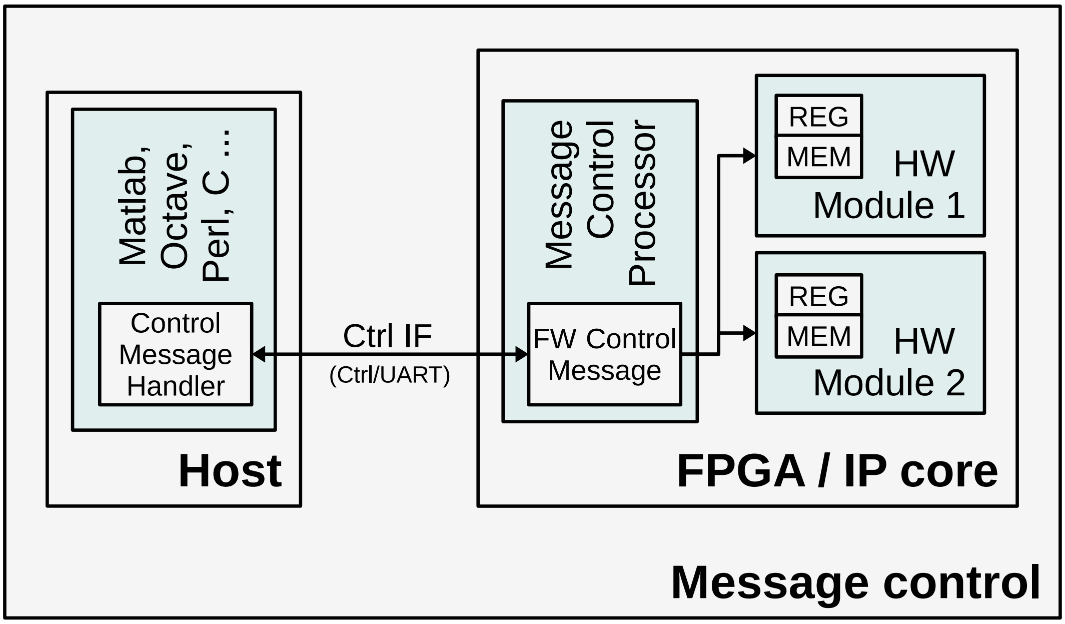

Fig. 4: Configuration via control messages

Setup of IP core features involve 3 stages, the control message handlers, the firmware control messages, and the actual signal processing module(s) implemented in Verilog, also called HW module(s) throughout this document, see also figure 4. Details on messages can be found in section 6.

- **Verilog modules (IP core hardware modules)**:
  One or several Verilog modules implement the actual signal processing operation on the FPGA. Such a module could be an FIR filter, or a CORDIC. HW modules are configured via registers. The configuration of these HW modules is not done by the user directly. Instead, control messages provide an abstract way to access the functionality.
- **Control messages (IP core firmware functions)**:
  The firmware based control messages are the interface the VRF IP core provides for host access. A control message configures HW module registers and possibly memory. Typically, HW module registers are set directly from the message parameters with little or no calculation being carried out on the internal message controller. Each control message request has a corresponding confirm. Control messages are listed in section 6.3.
- **Control message handlers (Octave/Matlab or other host software functions)**:
  On the host side, Octave/Matlab example implementations of message handler functions are used to send data to / read data from the VRF IP core. The functions either pass parameters directly, or perform additional calculation using more abstract inputs. E.g., a clock offset might be passed in ppm to the message handler, the message handler then calculates the necessary fixed point parameters needed to configure the HW. The messages control handler functions distributed with the IP core do not need to be used. The user is free to replace or extend them according to his requirements. Control message handlers are found in directory *./msg*.

Combined functions

In addition, some setup example functions are provided that combine several control message handler calls. E.g., to emulate coupled baseband and RF clock, *setupVrfClkFreqOff.m* combines setting of frequency and clock offset. Similarly, function *setupVrfPa.m* uses a predefined example curve to calculate the LUT for power amplifier emulation based on a few simple parameters. These functions can be found in directory *./tst/m/test*.

3.5 Interfaces and handshake signals

3.5.1 Overview

The IP core has several data and control interfaces such as *ctrlIn/Out*, *dacRe/Im*, *adcRe/Im*, etc... An interface *xyz* typically comes with corresponding *xyz_ir* (input ready) and *xyz_or* (output ready) signals. Data transfer occurs at the positive clock edge following the handshake signals, cf. figure 5.

3.5.2 Inputs

For inputs (*xyzIn*), the input ready signal *xyzIn_ir* is an output indicating to the connecting module that the input *xyzIn* is ready to accept data. The output ready signal *xyzIn_or* is an input controlled by the connecting module to indicate that data is available.

Data is transferred if *xyzIn_ir*=1 and *xyzIn_or*=1 simultaneously. If signal *xyzIn_or* is not available, the connecting module must supply data each time *xyzIn_ir*=1. If signal *xyzIn_ir* is not available, the input will accept data at any time with data transfer occuring if *xyzIn_or*=1 is set by the connecting module.

3.5.3 Outputs

For outputs (*xyzOut*), the output ready signal *xyzOut_or* is an output indicating to the connecting module that the output *xyzOut* has data available. The input ready signal *xyzOut_ir* is an input controlled by the connecting module to indicate that data can be accepted.

Data is transferred if *xyzOut_ir*=1 and *xyzOut_or*=1 simultaneously. If signal *xyzOut_ir* is not available, the connecting module must accept data each time *xyzOut_or*=1. If signal *xyzOut_or* is not available, the output will provide data at any time with data transfer occuring if *xyzOut_ir*=1 is set by the connecting module.

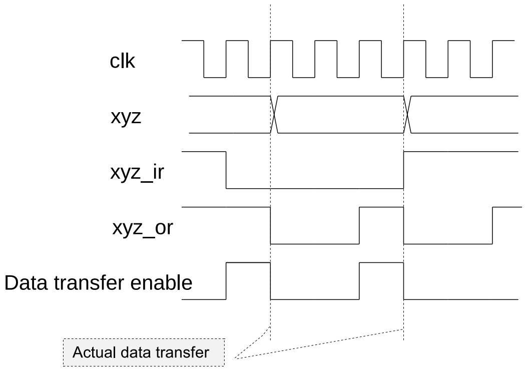

Fig. 5: Data transfer

3.5.4 Summary

In order to connect an IP core input (*xyzIn*) with a corresponding output (*xyzOut*) of a connecting external module assumed to have the same type of interface (or v.v.), connection must be made as

- assign *xyzIn* = *xyzOut*
- assign *xyzIn_or* = *xyzOut_or*
- assign *xyzOut_ir* = *xyzIn_ir*

Data transfer signaling is depicted in figure 5. Full handshaking does not always make sense, e.g. it is provided for *ctrlIn/Out*, but in some other cases, one of the signals is omitted.

3.5.5 Partial use of handshaking lines

Full handshaking does not always make sense. It is provided only for *ctrlIn/Out*. In other cases, one of the signals is omitted. E.g., for the DAC input, there is only handshake signal *dac_ir* available. Simulating a real DAC at a certain clock rate, it cannot wait for the connecting TX-PHY core, but needs data at certain clock cycles. Similarly, the ADC output provides data at some clock cycles like a real ADC and cannot wait for the RX-PHY core to accept them. Therefore, *adc_or* is the only handshake signal there.

4 RF emulation

4.1 Overview

4.1.1 Summary

RF emulation is depicted in the lower part of figure 1, details are shown in figure 6, figure 7, and figure 8. The emulation consists of 3 major components, RF-TX, RF channel, and RF-RX, which are explained in more detail below.

4.1.2 TX

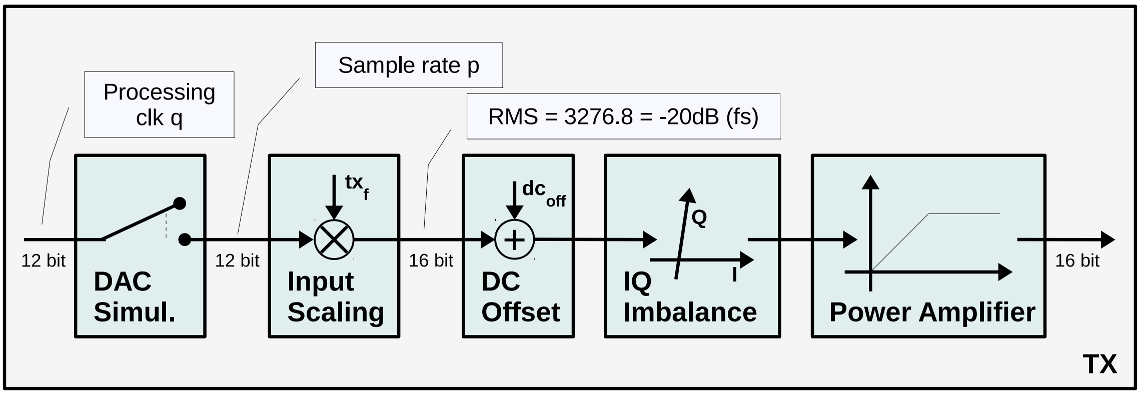

Fig. 6: TX overview

An overview over TX is shown in figure 6. The DAC input accepts 12 bit complex valued data. In order to simulate DAC rates that are lower than the processing clock frequency, DAC simulation requests data only during p out of q clock cycles.

Input scaling adapts data to the default internal level of -20 dB full scale with respect to 16 bit internal processing. DC offset and IQ imbalance can be added subsequently.

Finally, a power amplifier with arbitrary definition of the compression curve depending on the input power completes the TX model.

4.1.3 Channel

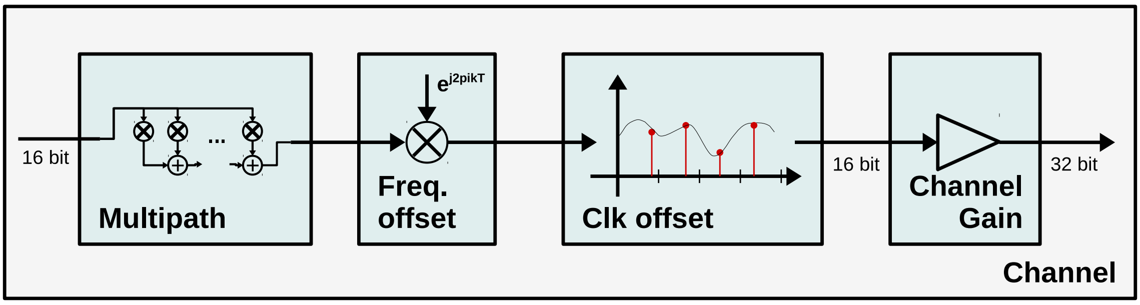

Fig. 7: Channel overview

Channel modeling is depicted in figure 7. Multipath fading is accomplished by an FIR filter with up to 10 coefficients. The coefficients can be defined arbitrarily with respect to amplitude and time delay.

Frequency offset is added by rotation of the signal in the time domain, and clock offset can be set in the range $[-1000 \dots +1000]$ ppm using filtering on a subsampling basis.

Finally, the channel gain multiplies the signal by an arbitrary factor while at the same time increasing the bit width from 16 to 32 bit, thereby modeling the wide dynamic range of the signal on the air.

4.1.4 RX

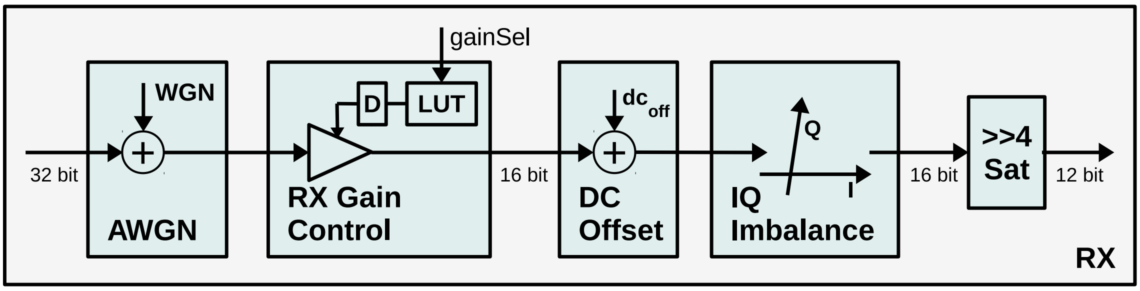

Fig. 8: RX overview

RX modeling is shown in figure 8. First, white gaussian noise (WGN) with constant signal power is added to the RX input. Since this input has 32 bit, SNR modeling is possible over a wide range by selecting a proper factor for the channel gain.

RX gain control features a 7 bit *gainSel* signal that must be driven by the users RX AGC. Signal *gainSel* serves as index to a LUT, which contains gain settings that can be defined arbitrarily by the user. Typically, one will select gain settings similar to the ones of the real RF that is connected later. Gain settings can be delayed by a fixed number of samples to model possible delays in the real RF-RX path.

DC offset and IQ imbalance modeling is provided the same way as for TX. Finally, the signal is shifted right $(>> 4)$ and saturated to 12 bit before being passed to the ADC output of the VRF core.

4.2 VRF on/off

The complete VRF chain can be enabled / disabled with message *OnOffReq*. Whenever the VRF configuration is changed, the VRF should be switched off before, and switched on after the config changes have been made. See section 6.3.3 for details.

4.3 TX - DAC

4.3.1 Functional description

The VRF IP core can process 1 sample in each clock cycle. If the processing clock frequency (parameter q) is higher than the baseband PHY sample rate (=ADC/DAC rate, parameter p) of the real system, DAC simulation provides the means to emulate a DAC running at sample rate p, i.e. requesting data from the TX PHY only in p out of q clock cycles.

Assume an LTE system with the PHY signal processing operating at 100 MHz, while the BB sampling rate is only 30.72 MS/s. A real DAC connected to the TX part of the system would request $p = 30.72 \cdot 10^6$ samples each $q = 100 \cdot 10^6$ clock cycles.

Different from all other modules, TX DAC emulation has *no signal processing capabilities*. Instead, it only slows down input sample reading and subsequent processing. Thereby, it allows indirect validation of the TX-PHY core implementation in terms of providing data at the rate requested by the DAC. Likewise, the same applies on the RX side for data supplied by the ADC.

4.3.2 HW operation

Both parameters p and q are 32-bit, hence $p,q=[1..2^{32}-1]$, with $p \leq q$. For the example above, $p=30720000$ and $q=100000000$ could be set, or equally $p=192$, $q=625$. Signal *adc_ir* (ADC input ready) is "1" whenever data is requested from the TX PHY during processing. The request pattern is as regular as possible, i.e., because $100000000/30720000=3.2552$, *adc_ir* = "1" will occur every 3rd or 4th clock cycle, with a repeating pattern each 625 clock cycles.

4.3.3 FW control message

Message *CfgDacRateReq* takes the 4 16-bit parameters $pLo16$, $pHi16$, $qLo16$, and $qHi16$ and writes them to the corresponding configuration registers. See section 6.3.15 for details on the message.

4.3.4 Host control message handler

Function *msgVrfCfgDacRateReq.m* uses 32-bit values p and q as inputs, splits them into upper and lower 16 bit parts, and calls message *CfgDacRateReq*.

4.4 TX - Input scaling

4.4.1 Functional description

Most of the VRF signal processing operates at 16 bit internally. For correct operation, the signal level must be at exactly -20 dB with respect to the full scale internal representation. i.e.,

$$\sqrt{\mathrm{E}(|Y|^2)}=2^{15}\cdot 10^{-20/20}=3276.8 \tag{2}$$

where samples y are the 16 bit complex valued outputs after TX input data scaling. This must hold independent of the VRF input (= TX PHY output) signal level. Therefore, the input must be scaled such that the required VRF internal signal level is achieved.

4.4.2 HW operation

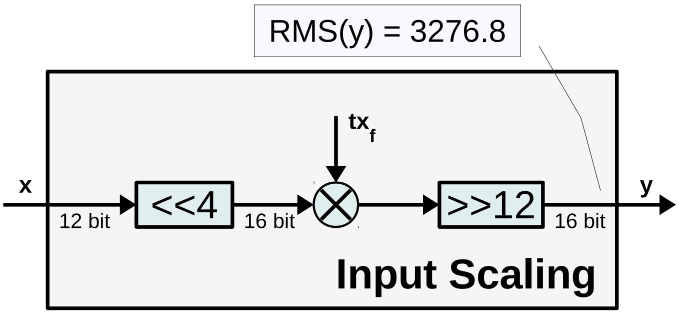

Fig. 9: TX input scaling

Given the 12 bit complex valued input samples x of the VRF, the output samples y are calculated as

$$y=x\cdot 2^4\cdot tx_f/2^{12}=x\cdot tx_f/2^8\ . \tag{3}$$

The factor 2^4 results from the MSB aligned mapping of 12 bit signal x to 16 bit, which are then internally multiplied by tx_f and finally shifted right $>>12$, see figure 9.

Factor tx_f can assume values from $[1..32767]$. It is calculated as

$$tx_f = \frac{3276.8 \cdot 2^8}{\sqrt{\mathrm{E}(|X|^2)}} \tag{4}$$

or equally

$$tx_f = 2^{12} \cdot 10^{(IBO-20)/20} \tag{5}$$

when IBO is the VRF input backoff in dB of the 12 bit TX PHY output data x defined as

$$IBO = -20 \cdot \log_{10}(\sqrt{\mathrm{E}(|X|^2)}/2^{11})\ . \tag{6}$$

The range of tx_f is $[1..32767]$, therefore, the maximum IBO is about 38 dB.

4.4.3 FW control message

Message *CfgTxInpScReq* writes the scaling parameter tx_f to the TX scaling configuration register. See also section 6.3.19 for details on message *CfgTxInpScReq*.

4.4.4 Host control message handler

Function *msgVrfTxInpScReq.m* takes the input backoff (in dB) as parameter and calculates the scaling factor according to (5) before calling message *CfgTxInpScReq*.

4.5 TX - DC offset

4.5.1 Functional description

TX DC offset adds a constant value to the TX signal. As the RMS of the complex valued signal is 3276.8 after TX scaling (see section 4.4), the DC offset must be seen relative to this.

4.5.2 HW operation

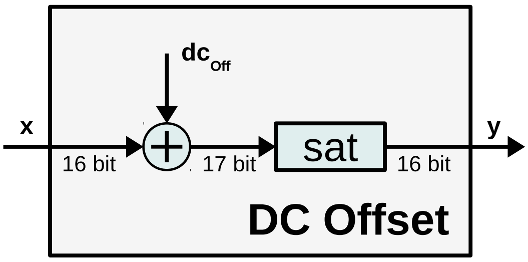

Fig. 10: TX DC offset

DC offset emulation allows arbitrary constant 16-bit signed values to be added to the signal. Final saturation is applied, see also figure 10.

4.5.3 FW control message

Message *CfgTxDcOffReq* writes the offset parameters for real and imaginary part to the VRF core. See also section 6.3.20 for details on message *CfgTxDcOffReq*.

4.5.4 Host control message handler

Function *msgVrfTxDcOffReq.m* uses integer DC offset values as parameters and transfers them as provided by calling message *CfgTxDcOffReq* for configuration.

4.6 TX - IQ imbalance

4.6.1 Functional description

TX IQ imbalance is modeled by assuming the imaginary part of the signal being distorted in terms of phase α and amplitude factor a_F, while the real part of signal remains unaffected. Additionally, normalization scaling by factor k_F shall be applied such that the power of the output signal is kept. The resulting output is

$$y = k_F \cdot (x_{Re} + \mathrm{j} \cdot a_F \mathrm{e}^{\mathrm{j}\alpha} \cdot x_{Im}) \tag{7}$$

with $k_F = \sqrt{2/(1+a_F^2)}$.

4.6.2 HW operation

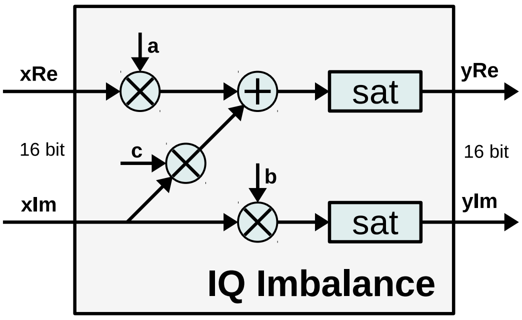

Fig. 11: TX IQ imbalance

The implementation of the TX IQ imbalance is depicted in figure 11. Factors a, b, and c can be configured via registers. Real and imaginary part of output y can be written as

$$\begin{pmatrix} y_{Re} \\ y_{Im} \end{pmatrix} = \begin{pmatrix} a & c \\ 0 & b \end{pmatrix} \begin{pmatrix} x_{Re} \\ x_{Im} \end{pmatrix} . \tag{8}$$

Comparing (7) and (8), factors a, b, and c can be calculated as

$$a = k_F \tag{9}$$

$$b = k_F \cdot a_F \cdot \cos(\alpha) \tag{10}$$

$$c = k_F \cdot a_F \cdot \sin(-\alpha) \tag{11}$$

Factors a, b, and c are internally represented by 16 bit in Q14 format, thereby providing sufficient resolution for any possible combination $a_F = [0..\infty]$ and $\alpha = [-180^\circ..180^\circ]$. Default values are $a = 1$, $b = 1$, $c = 0$ (no IQ imbalance).

4.6.3 FW control message

Message *CfgTxIqImbReq* writes parameters $a_{fxp} = a \cdot 2^{14}$, $b_{fxp} = b \cdot 2^{14}$, and $c_{fxp} = c \cdot 2^{14}$ to the configuration registers. See also section 6.3.21 for details on message *CfgTxIqImbReq*.

4.6.4 Host control message handler

Function *msgVrfTxIqImbReq.m* takes phase $\alpha = [-180..180]$ in degree and the amplitude factor $a_F = [0..\infty]$ as input, calculates a, b, and c according to (9)-(11), scales by 2^{14}, and calls message *CfgTxIqImbReq* for configuration.

4.7 TX - Power amplifier

4.7.1 Functional description

The TX power amplifier is modeled as memoryless phase and amplitude distortion depending on the magnitude of the complex input values

$$y = a(|x|)\mathrm{e}^{j\varphi(|x|)} \cdot x \tag{12}$$

Both, amplitude factor a and phase shift φ can be defined arbitrarily via lookup tables (LUTs).

4.7.2 HW operation

Power amplifier emulation involves several Cordic functions, multipliers, and LUTs followed by linear interpolation. Figure 12 gives an overview.

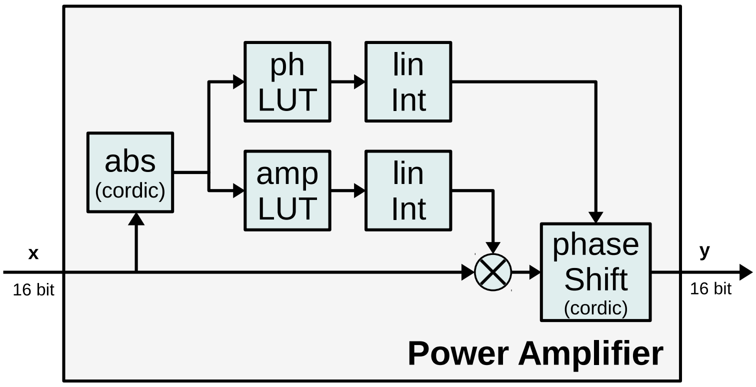

Fig. 12: Power amplifier

The magnitude of the complex input values x is calculated and fed into 2 different LUTs, one for the phase shift, one for the amplitude factor. The LUTs have 513 entries a_k for the amplitude factor and φ_k for the phase shift, with $k = 0..512$ corresponding to input magnitude values $|x| = 0, 64, 128, ..., 32768$.

Amplitude factor values are 15 bit, i.e Q15 unsigned, fixed point range $a_{fxp} = [0..32767]$ corresponding to factors $a = [0..0.99997]$. Fixed point phase shift values range from $\varphi_{fxp} = [-32768..+32767]$ corresponding to phase shifts (in degree) of $\varphi = [-180..+180]$.

The PA modeling can be bypassed completely. If not bypassed, a fixed vs. float accuracy of about 60 dB is achieved.

4.7.3 FW control message

Message *CfgTxPaLutReq* reads values a_k $(\cdot 2^{15})$ and φ_k $(\cdot 2^{16}/360)$ in an interleaved fashion for $k = 0..512$, i.e., $a_0, \varphi_0, a_1, \varphi_1, ..., a_{512}, \varphi_{512}$. The power amplifier module keeps absolute values a_k, φ_k, and differential values, $a_{k+1} - a_k$, $\varphi_{k+1} - \varphi_k$. Calculation of the differential values is done in the FW. See also section 6.3.23 for details on message *CfgTxPaLutReq*.

Message *CfgTxPaReq* allows to switch between normal PA modeling operation and bypass mode. See also section 6.3.22 for details on message *CfgTxPaReq*.

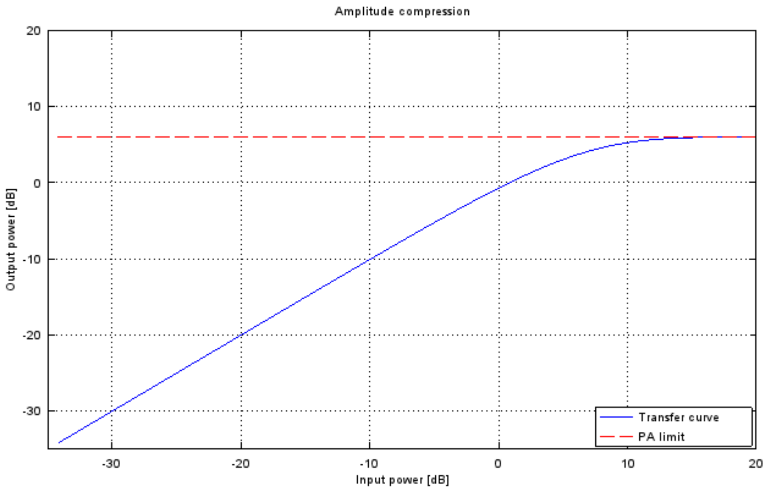

Fig. 13: PA amplitude modeling example

4.7.4 Host control message handler

Function *msgVrfTxPaLutReq.m* takes 2 vectors with 513 entries each, $a_k \in [0..0.99997]$, $\varphi_k \in [-180..180]$, scales and reorders them as described in section 4.7.3, and calls message *CfgTxPaLutReq* to configure the PA-LUTs.

Function *msgVrfTxPaReq.m* calls message *CfgTxPaReq* with the bypass parameter (0 or 1).

4.7.5 Example setup function

An example function is given in function *setupVrfPa.m*. Note that this is just an example and does not refer to any real world RF-PA model. Instead it only illustrates the setup of the PA modeling. The user is free to configure the LUTs according to his own PA model.

Amplitude compression

For amplitude compression, a tangens hyperbolicus has been selected to model the behaviour. Assuming a_M being the parameter for the maximum amplitude,

$$|y| = a_M \tanh(|x|/a_M) \tag{13}$$

defines a suitable behaviour. While $|y| = |x|$ holds for small values of $|x|$, the maximum output of $|y|$ converges to a_M for large $|x|$. E.g., if a PA backoff of 6 dB shall be modeled, $a_M = 3276.8 \cdot 10^{6/20} = 6538$ should be selected, as $\sqrt{(E(|x|^2)} = 3276.8$, cf. section 4.4. Figure 13 illustrates the related transfer function between input and output amplitude of the PA. The input amplitude of 0 dB refers to the value 3276.8.

Amplitude factors are finally calculated as

$$a_k = |y_k|/|x_k| = a_M \tanh(|x_k|/a_M)/|x_k| \tag{14}$$

with $x_k = [0, 64, 128, ..., 32704, 32768]$ for $k = 0..512$.

Because the RMS of the VRF internal signal level is 20 dB below full scale, arbitrary PA backoffs between 0..20 dB can be modeled.

Phase distortion

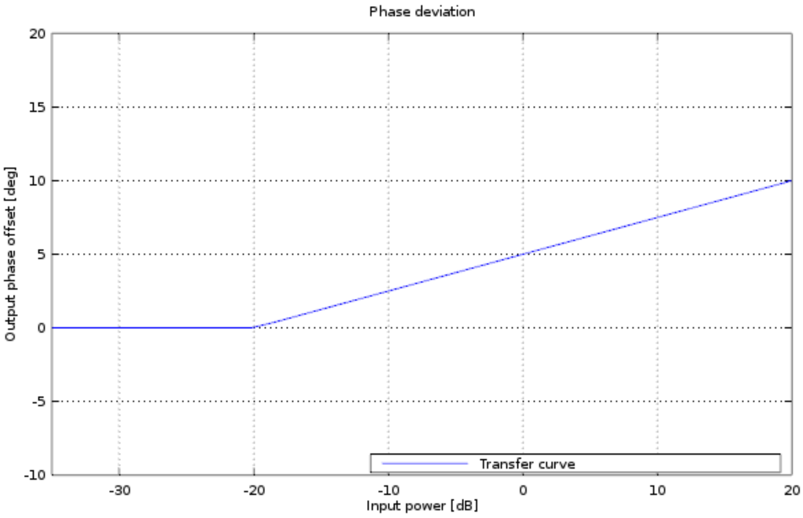

Fig. 14: PA phase distortion modeling example

Similarly, some arbitrarily defined phase shift φ is calculated in function *setupVrfPa.m*, see figure 14. The phase offset increases linearly up to a maximum value for input levels > -20dB. Depending on $x_{dB} = 20\log_{10}(|x|/3276.8)$ and a maximum phase φ_M the phase distortion can be expressed as

$$\varphi = \begin{cases} 0 & \text{for } x_{dB} < -20 \\ \varphi_M \cdot (x_{dB}+20)/40 & \text{for } x_{dB} \geq -20 \end{cases} \tag{15}$$

4.8 Channel - Multipath

4.8.1 Functional description

Multipath channel emulation uses a flexible FIR filter with up to 10 different paths. Each path can be assigned a delay between 0..29 samples. In detail, the output is calculated as

$$y_k = \sum_{l=0}^{9} c_l x_{k-d_l} \tag{16}$$

where x_{k-d_l} are the input values delayed by $d_l \in [0..29]$ samples, and c_l are the corresponding complex valued coefficients.

4.8.2 HW operation

The HW operation is shown in figure 15. Complex valued coefficients must have a magnitude $|c| < 2$. The *mux-Add* units (*ma0* .. *ma29*) include bit extension and subsequent saturation to 16 bit to prevent possible overflow. Each multiplexor can select one path $x \cdot c_l$ arbitrarily, or use 0 at its input. Coefficients are handled internally in Q13 format.

4.8.3 FW control message

Message *CfgMultiPathReq* first resets all coefficients c to 0 and also all multiplexors to the zero input. It then reads the number of coefficients $n_c = [1..10]$, and afterwards n_c triples (c_{Re}, c_{Im}, d), where c_{Re}, c_{Im} are real/imag part of the coefficients in Q13 format, and d is the corresponding delay in samples. As all complex coefficients must have $|c| < 2$, the fixed point amplitude of the corresponding Q13 values is limited to $|c| \cdot 2^{13} < 2^{14}$.

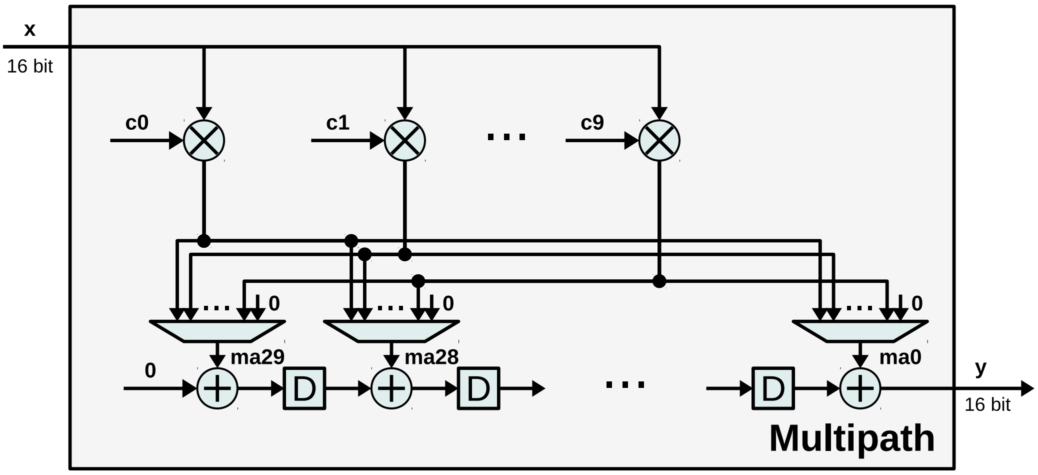

Fig. 15: Multipath operation

Coefficients are converted internally to value triples $(c_{Re},\ c_{Re}+c_{Im},\ c_{Re}-c_{Im})$ before being written to the configuration registers. Multiplexors are set according to the corresponding delay.

Note that different from most other messages, this message has a variable number of coefficients, thus a variable number of parameters. See also section 6.3.16 for details on message *CfgMultiPathReq*.

4.8.4 Host control message handler

Function *msgVrfCfgMultiPathReq.m* needs the complex valued input vector c and integer vector d as input parameters. Data is checked for integrity, $c_{fxp} = c \cdot 2^{13}$ is calculated, and the vectors are resorted into the triples. Finally, $[n_c,\ c_{0,Re,fxp},\ c_{0,Im,fxp},\ d_0,\ c_{1,Re,fxp},\ \ldots d_{n_c-1}]$ is sent to the IP core by message *CfgMultiPathReq*.

4.9 Channel - Frequency offset

4.9.1 Functional description

Functionality of frequency offset emulation can be described as

$$y_k = x_k\,\mathrm{e}^{-j2\pi k f_r} \tag{17}$$

where k is the sample index, $f_r = f_o/f_s$ is the relative frequency offset with respect to the baseband sample frequency f_s, and $f_o = f_{c(RX)} - f_{c(TX)}$ is the difference between RX and TX carrier frequency. By definition, frequency offset f_o is positive for $f_{c(RX)} > f_{c(TX)}$.

4.9.2 HW operation

The frequency offset emulation module shown in figure 16 consists of a phase accumulator with an 48 bit input for f_r and a subsequent Cordic. The range $f_r = [-0.5..+0.5]$ is mapped to $f_{r,fxp} = [-2^{47}..+2^{47}-1]$, i.e., f_r is a Q47 fixed point value. The Cordic includes scaling such that the signal amplitude is not changed.

4.9.3 FW control message

Due to the 16 bit nature of the control interface, message *CfgFreqOffReq* takes 3 16-bit inputs $f_{r,fxp} = [f_{r,fxp,HI}|f_{r,fxp,MI}|f_{r,fxp,LO}]$ and writes them to configuration registers without changes. See also section 6.3.17 for details on message *CfgFreqOffReq*.

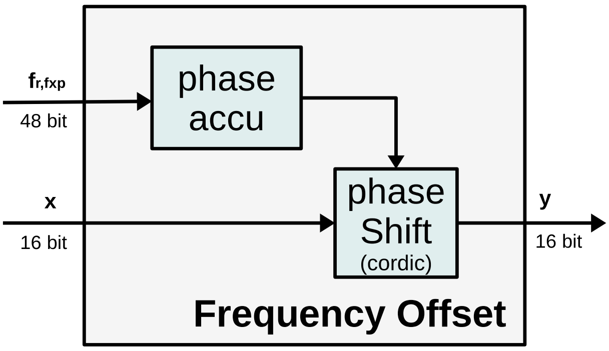

Fig. 16: Frequency offset modeling

4.9.4 Host control message handler

Function *msgVrfCfgFreqOffReg.m* needs the clock offset in PPM $(clkOffPpm)$, the carrier frequency (f_C), and the sampling frequency (f_S) as parameters to calculate the relative frequency offset

$$f_r = clkOffPpm/10^6 \cdot f_C/f_S \tag{18}$$

Fixed point conversion calculates $f_{r,fxp} = f_r \cdot 2^{48}$, selects the upper, middle, and lower 16 bit, and configures the IP core using message *CfgFreqOffReq*.

Note that the definition of the clock offset in function *msgVrfCfgFreqOffReg.m* is consistent with the clock offset definition in *msgVrfCfgClkOffReg.m* if baseband clock and carrier frequency are derived from the same clock source, cf. section 4.10.4.

4.10 Channel - Clock offset

4.10.1 Functional description

Clock offset emulation is realized by passing the signal through an FIR filter while shifting the filter coefficients in the time domain.

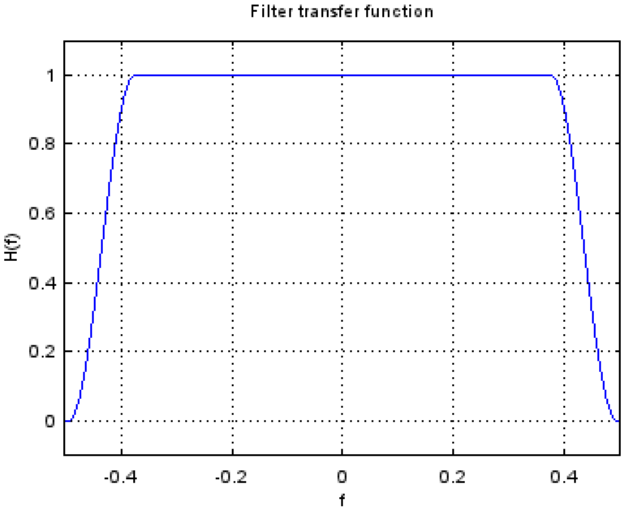

Fig. 17: Clock offset filter function

The filter transfer function is depicted in figure 17. Choosing $H(f) = 1$ for frequencies $f = [-0.375..+0.375]$ (normalized to the sample frequency), with cos-roll-off to the band edges yields a good compromise of a relatively wide passband in combination with a time domain impulse response of a limited length.

Let $h_{k,\Delta} = \hat{h}((k+\Delta)T_s)$ be the sampled version of the analog filter impulse response $\hat{h}(t)$ sampled at time $(k+\Delta)T_s$, with $\Delta = [-0.5..+0.5]$, and $y_{k,\Delta} = \hat{y}((k+\Delta)T_s)$ the corresponding sampled outputs y obtained by filtering input x by h, then the output can be described as

$$y_{k,\Delta} = \sum_l x_l \cdot h_{k-l,\Delta} \,. \tag{19}$$

Shifting the output by a fraction of a sample is achieved by selecting proper filter coefficients $h_{k,\Delta}$. Clock offset is emulated by continuously updating Δ, i.e. changing the coefficients based on a value c_o that represents the relative offset of the oscillators,

$$\Delta = -N \cdot c_o \tag{20}$$

where N is the number of samples passed since the simulation started. For practical purposes, $\Delta \in [-0.5..+0.5]$ by definition with additional sample slip occuring upon overflow, see description of the HW operation below.

The disadvantage of the limited length of the impulse response is that only about 75% of the available band width can be used when clock offset emulation is active, even when the actual offset is set to zero. Furthermore, due to limited filter length, resolution, and update rate, the output signal accuracy of the block (compared to a perfect floating point implementation) is limited to about 50 dB. This value is strongly depending on the simulated clock offset, see table 1 for details. Using a clock offset > 1000 ppm is not recommended.

Clock offset in ppm	50	100	200	500	1000
Signal accuracy in [dB]	50	49	47	42	36

Tbl. 1: Clock offset signal accuracy (approximate)

If the feature is not needed, clock offset emulation can be bypassed completely, thereby avoiding the limitation of signal accuracy and the bandwidth restriction.

4.10.2 HW operation

The HW implementation consist of 2 submodules *firToff* and *coeffToff*, see figure 18.

Submodule *firToff* represents an FIR filter with variable coefficients. In each processing step, a new set of coefficients is loaded and 32 output samples are created. Input data is normally shifted by 32 samples during that time. Depending on the input control signal, one sample can be either skipped (extra input data shift) or used a second time (skip input data shift) before the operation starts. This compensates for possible coefficient time jump of $h_{k,\Delta}$ from $\Delta = -0.5 \rightarrow +0.5$ or v.v.

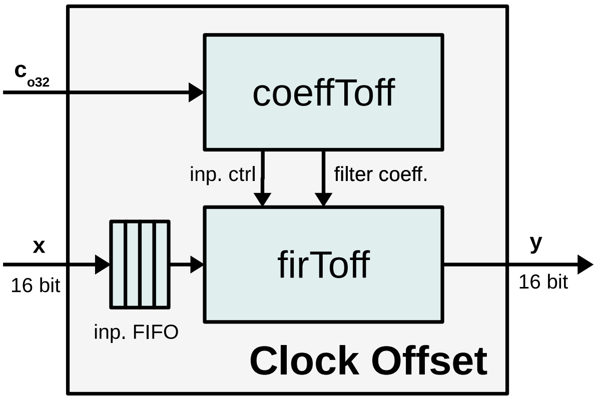

Fig. 18: Time/clock offset module

Because clock offset emulation operates in blocks of 32 samples, it needs $c_{o32} = 32 \cdot c_o$ as parameter to update the FIR filter coefficients. This update rate is assumed to be sufficient since typical clock offsets are $< 100\,\text{ppm}$. The internal control uses a 48 bit (Q47) representation for c_{o32}, i.e., $c_{o32,fxp} = 32 \cdot c_o \cdot 2^{48}$.

If $c_o > 0$, the RX sample frequency is lower than the TX sample frequency and the module produces less output samples than input samples, i.e. the output is halted from time to time while the input skips 1 sample during processing.

If $c_o < 0$, the RX sample frequency is higher than the TX sample frequency, the module produces more output samples than input samples. Therefore some samples are used twice and the input is blocked for 1 cycle from time to time.

Although there is a small FIFO buffer at the input, in the long run $c_o < 0$ can only work if the following relation holds for the DAC rate p/q (cf. section 4.3)

$$p/q < 1/(1 - c_o)\,. \tag{21}$$

If $p = q$, i.e. if the DAC sample rate is the same as the signal processing clock, it is not possible to use $c_o < 0$. There is no such restriction for $c_o > 0$.

In bypass mode, module *firToff* still processes normally, but is virtually inactive as *coeffToff* passes a single tap with value "1" as coefficient. Sample slip is avoided by ignoring clock offset values.

4.10.3 FW control message

Because of the 16 bit nature of the control interface, control message *CfgClkOffReq* needs $c_{o32,fxp} = [c_{o32,fxpHI} | c_{o32,fxpMI} | c_{o32,fxpLO}]$ for the clock offset, and writes these 3 parameters plus the bypass parameter to submodule *coeffToff* without changes. See also section 6.3.18 for details on message *CfgClkOffReq*.

4.10.4 Host control message handler

Function *msgVrfCfgClkOffReq.m* takes a relative offset given in ppm $|clkOffPpm| < 1000$ and a bypass selector (0 or 1) as parameters. In order to be consistent with the definition of frequency offset in section 4.9.1, i.e., a positive offset corresponding to a higher clock/sampling frequency at the receiver, $c_{o32,fxp}$ must be calculated as

$$c_{o32,fxp} = \left(\frac{1}{1 + clkOffPpm/10^6} - 1 \right) \cdot 32 \cdot 2^{48}\,. \tag{22}$$

Equation (22) turns a positive $clkOffPpm$ into a negative time step $c_{o32,fxp}$. The 48-bit value $c_{o32,fxp}$ is split into 3 16-bit values $c_{o32,fxpHI}$, $c_{o32,fxpMI}$, $c_{o32,fxpLO}$ and configures the VRF core via message *CfgClkOffReq*.

4.10.5 Example setup function

Function *setupVrfClkFreqOffset.m* gives an example how to setup a system with clock and carrier frequency coupled, i.e., derived from the same clock source. Possible violations of the DAC sampling rate in conjunction with positive clock offsets are checked within this function.

4.11 Channel - Gain

4.11.1 Functional description

Gain control converts the 16-bit internal data into 32 bit, thereby simulating the arbitrary dynamic range of the signal on the air. The output is calculated as

$$y = x \cdot g_f \cdot 2^{g_s} \tag{23}$$

where $g_f = 0.5..1$ is the gain factor, and $g_s = -32..+18$ is a gain shift. The output y is saturated to 32 bit after the operation.

4.11.2 HW operation

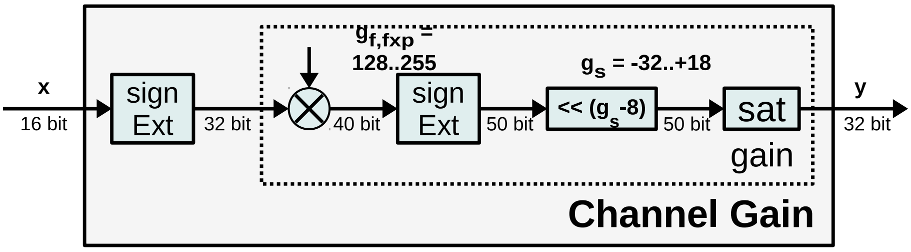

Fig. 19: Channel gain

Figure figure 19 depicts the HW operation. The 16-bit signal is first sign extended to 32 bit and then passed through the *gain* module. This module provides multiplication by $g_{f,fxp} = g_f \cdot 256 = 128..255$ and subsequent shift by $<< (g_s - 8)$ without internal overflow. The signal is finally saturated to 32 bit again. Because $20\log_{10}(129/128) = 0.067$, the accuracy of the gain setting is better than 0.1 dB.

4.11.3 FW control message

Message *CfgChGainReq* uses gain factor $g_{f,fxp}$ and gain shift g_s to set the corresponding HW registers. See also section 6.3.24 for details on message *CfgChGainReq*.

4.11.4 Host control message handler

Function *msgVrfChGainReq.m* uses the time domain SNR value as input in order to calculate the appropriate channel gain. It assumes the default signal RMS $\sigma_x = 3276.8$ (see section section 4.4) and the corresponding white gaussian noise with $\sigma_n = 796$ (see section section 4.12). The necessary factor to achieve a certain SNR is given by

$$f = 10^{snr/20} \cdot (\sigma_n/\sigma_x) \tag{24}$$

From f, the corresponding values $g_{f,fxp}$ and g_s are calculated as

$$g_s = \lceil \log_2(f) \rceil \tag{25}$$

$$g_{f,fxp} = \mathrm{round}(256 \cdot 2^{\log_2(f) - g_s}) \tag{26}$$

with possible correction if $g_{f,fxp} = 256$ in (26)

$$(g_{f,fxp}, g_s) = (256, n) \to (128, n+1) \tag{27}$$

Finally, $g_{f,fxp}$ and g_s are written to the VRF IP core via FW message *CfgChGainReq*. Given the range of g_s, it is possible to emulate SNRs from roughly -180..+120 dB.

4.12 RX - AWGN

4.12.1 Functional description

White gaussian noise (WGN) of fixed complex RMS $\sigma_n = 796$ is added to the signal. Thermal noise at the receiver input is modeled this way. Details of the distribution can be found in figure 20 and figure 21.

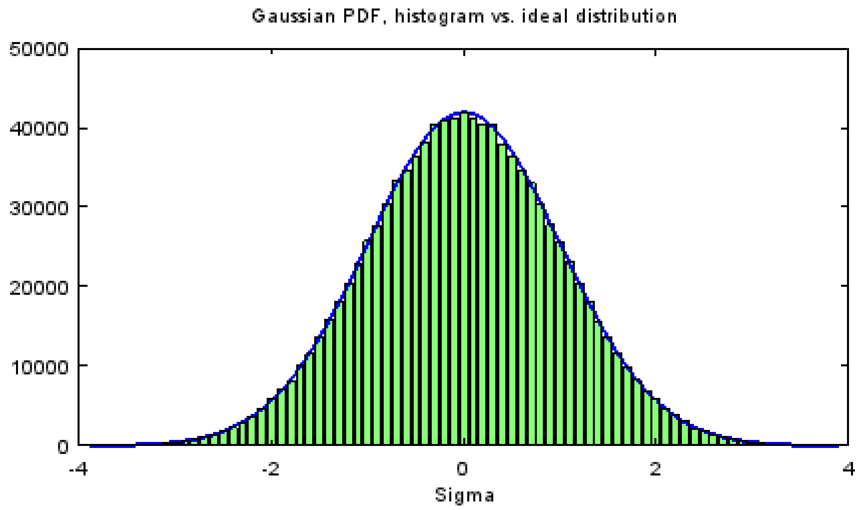

Fig. 20: AWGN histogram using 10^6 samples

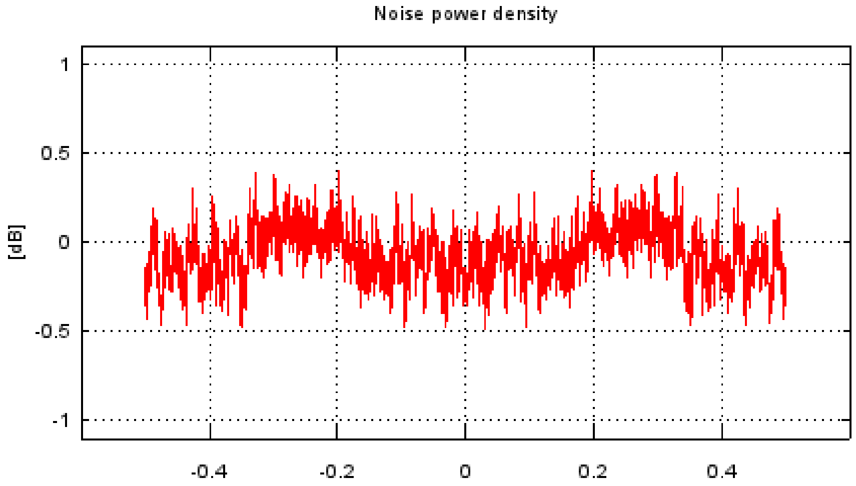

Fig. 21: AWGN power spectrum density using 10^6 samples

4.12.2 HW operation

The module consists of several linear feedback shift registers of different length and feedback polynomials combined in a way that WGN like signal results. Real + imaginary part are created using the same basic circuitry but with different state initialization values.

4.12.3 FW control message

There is no FW message controlling this module. WGN is added as soon as the RF model is switched on.

4.12.4 Host control message handler

There is no message control (see also FW message).

4.13 RX - Gain control

4.13.1 Functional description

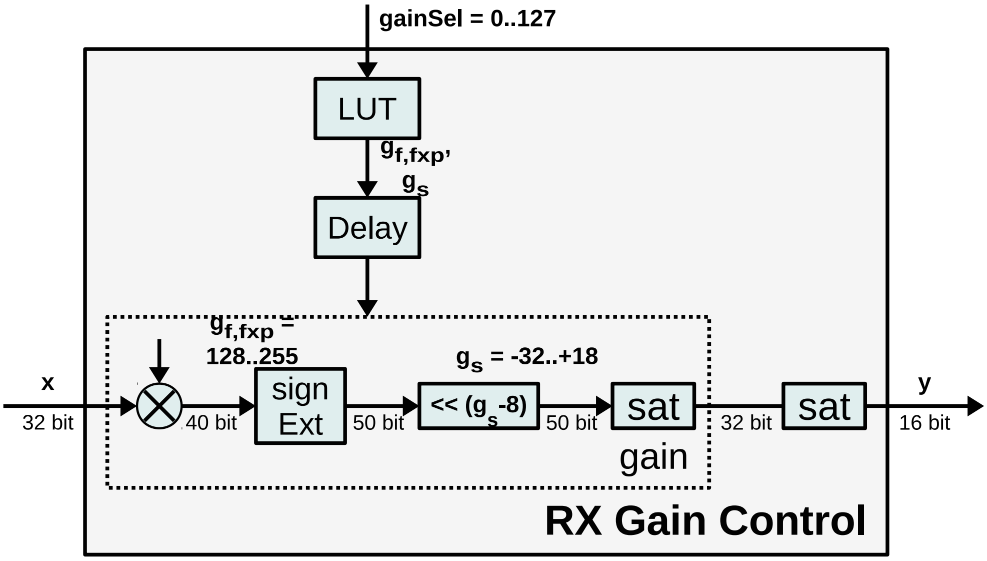

Fig. 22: Gain control

Gain control emulates the gain setting of the RF-RX path. The control input signal *gainSel* has 7 parallel lines that select one of 128 arbitrary gain steps from a LUT. Signal *gainSel* is controlled by the users RX signal processing core.

This LUT is freely configurable and of course it is also possible to use only a subset of the 128 gain steps provided at maximum. The user will typically define gains similar to the ones of the real RF hardware which he plans to use later, like 40 gain setting in steps of 2 dB.

4.13.2 HW operation

Figure 22 depicts the gain control operation. Similar to the description in the channel gain section (see section 4.11), RX gain control contains the *gain* submodule that provides almost arbitrary gains (range [-180..+100] dB).

The 7 parallel input lines *gainSel* are controlled by the users RX baseband core and serve as index of the LUT. The LUT outputs are gain factor $g_{f,fxp}$ and gain shift g_s. The LUT can be set arbitrarily via message control to emulate the behaviour of any real RF settings. In addition, a gain delay $g_D = 1..1023$ (baseband samples) can be defined. Setting $g_D = 0$ is valid but may also cause a delay of 1 in case a sample is transferred in the same cycles as the new gain value is applied. The data is eventually saturated to 16 bit for further processing in the RF-RX emulation.

4.13.3 FW control message

Message *CfgRxGainTblReq* uses values $[g_D,\ g_s(0), g_{f,fxp}(0), g_s(1), ..., g_{f,fxp}(127)]$ to set the gain delay HW register and the LUT, where $g_s = [-32..+18]$ are 6 bit, and $g_{f,fxp} = [128..255]$ are 8 bit. The message combines pairs into 14-bit values before writing to the LUT. See also section 6.3.12 for details on message *CfgRxGainTblReq*.

gainSel[5:0]						Gain
LNA		VGA				[dB]
1	1	1	1	1	1	-20
1	1	1	1	1	0	-23
1	1	1	1	0	1	-26
1	1	1	1	0	0	-29
1	0	1	1	1	1	-30
1	0	1	1	1	0	-33
1	0	1	1	0	1	-36
1	0	1	1	0	0	-39
0	0	1	1	1	1	-40
0	0	1	1	1	0	-43
0	0	1	1	0	1	-46
0	0	1	1	0	0	-49
0	0	1	0	1	1	-52
...	...	...	...	...	...	...
0	0	0	0	0	0	-85

Tbl. 2: Example RX gain table

4.13.4 Host control message handler

Function *msgVrfCfgRxGainTblReq.m* uses the scalar variable $g_D = gainDelay$, and vectors $gainDb(0..n_g-1)$, and $gainSel(0..n_g-1)$ as inputs, where n_g is the number of gain steps emulated (maximum 128). While g_D is transferred using the FW message as is, values g_S and $g_{f,fxp}$ are derived from $gainDb$ for each of the n_g values.

By definition $gainDb = 0$ is the gain that is needed to yield a noise level of $\sigma_{n,ADC} = 2^{11}$ at the 12-bit ADC output of the VRF IP core. In other words, if the WGN at the input of the RX gain control is amplified such that it fully drives the ADC, we define this gain as 0 dB.

The noise input has RMS $\sigma_n = 796$, cf. section 4.12, and the 16 bit output of module *gainCtrl* is shifted $>> 4$ later, so a gain offset in dB can be calculated as

$$gainOffsetDb = -20\log_{10}(796/2^4/2^{11}) \tag{28}$$

Consequently, the necessary (float) factor corresponding to $gainSel(k)$ is given by

$$f(k) = 10^{(gainDb(k)+gainOffsetDb)/20} \tag{29}$$

$$\tag{30}$$

Finally gain shift and fixed point gain factor are calculated as in section 4.11.4. Target index $l = gainSel(k)$ is used to make sure the LUT is addressed the right way

$$g_S(l) = \lceil \log_2(f(k)) \rceil \tag{31}$$

$$g_{f,fxp}(l) = \mathrm{round}(256 \cdot 2^{\log_2(f(k))-g_S(l)}) \tag{32}$$

Corner cases $[g_{f,fxp}, g_S] = [256, n] \rightarrow [128, n+1]$ are corrected as before in section 4.11.4. All unused LUT entries are set to 0 by default.

4.13.5 Example setup function

Function *setupRxGain.m* defines an example LUT for an RF with an LNA that has a gain steps of 0, 10, 20 dB controlled by bits [5:4], and a subsequent VGA with 16 steps of 3 dB controlled by bits [3:0] of *gainSel*. Furthermore, assume to have a HW setup such that the baseband noise level (no RX input signal from the channel) for the highest RF gain would be -20 dB full scale at the ADC. Your gain setup could then be represented by table 2.

In order to apply table 2, the control message handler *msgVrfCfgRxGainTblReq.m* must be called with parameters $gainDb = [-20, -23, -26, -29, -30, -33..., -85]$, $gainSel = [63, 62, 61, 60, 48, 47, ..., 0]$. This is included in function *setupRxGain.m*.

4.14 RX - DC offset

RX DC offset is implemented the same way as TX DC offset, see section 4.5 for details. However, it must be considered that the signal level at RX is not the default RMS = 3276.8 like at TX, but depends on RX and channel gain settings. The corresponding FW message is *CfgRxDcOffReq*, and the corresponding control message handler is *msgVrfRxDcOffReq.m*.

4.15 RX - IQ imbalance

RX IQ imbalance is implemented the same way as TX IQ imbalance, see section 4.6 for details. The corresponding FW message is *CfgRxIqImbReq*, and the corresponding control message handler is *msgVrfRxIqImbReq.m*.

4.16 RX - Saturation

Internal 16 bit RX data is eventually shifted right by 4 ($>>$ 4) and saturated to 12 bit before being passed to the ADC output of the VRF core.

5 Physical layer control

5.1 Overview

The TX physical layer IP core (PHY0), and the RX physical layer IP core (PHY1) must be connected to the *dac* inputs (PHY0-TX), and to the *adc* outputs / *gainSel* inputs (PHY1-RX) of the VRF, respectively. See also the example in figure 2.

If only the RF and channel emulation capabilities of the VRF IP core are needed, then these connections are sufficient. However, the VRF IP core offers additional capabilities to control the PHY IP cores as follows:

- Forward control messages to / from PHY IP cores.
- Send data to / read data from PHY TX/RX data buffers.
- Fill the TX data buffer via an internal data generator, read the RX data buffer via an internal data sink.
- Initiate transfer from the TX data buffer to the PHY0-TX IP core, and from the PHY1-RX IP core to the RX data buffer.
- Compare TX and RX data to detect frame errors.
- Use GPO/GPI pins to control the PHY IP cores, e.g. reset.

An overview is given in the upper part of figure 1. These features may or may not be used by the customer. If not required, the user should set the memory sizes of PHY0/1 buffers to small values in order to save FPGA resources, see also section 5.3.1.

5.2 Message forwarding

Message forwarding can be used to control the PHY IP cores via the same interface (*ctrl* or UART) that also controls the VRF. Data is sent / received via VRF control messages *Ctrl(Wr|Rd)Req* and the corresponding *Ctrl(Wr|Rd)Cfm*. See section 6.3.4, section 6.3.5, section 6.3.31, and section 6.3.32 for details on messages. The forwarding mechanism is described in detail section 6.2.

5.3 Data transmission + reception

5.3.1 Overview

The PHY control part contains TX + RX data buffers that can be be connected to the PHY IP cores if needed. By default, these buffers are 8 bit wide, i.e., they transfer data byte wise, and hold up to 4096 bytes.

Buffer size and width can be changed by adapting the parameters in file *./src/common/includes/def_Const_vrf.v*

```
parameter VrfDataWidth_C       = 8;
parameter VrfRxTxBufNum_C      = 4096;
parameter VrfRxTxBufAdrWidth_C = 12;
```

where, parameter *VrfDataWidth_C* is the data bit width, *VrfRxTxBufNum_C* is the number of words, *VrfRxTxBufAdrWidth_C* must be set to $\lceil \log_2(VrfRxTxBufNum\_C) \rceil$. The maximum data bit width is 16 bit, and the maximum number of words is $2^{16} = 65536$. If the PHY control features are not needed, set

```
parameter VrfDataWidth_C       = 1;
parameter VrfRxTxBufNum_C      = 2;
parameter VrfRxTxBufAdrWidth_C = 1;
```

in order to minimize the allocation of unused FPGA resources.

5.3.2 TX buffer filling via control interface

Message *DatBufTxWriteReq* allows to fill the TX data buffer via the control interface. If the buffer data bit width is less than 16 bit (default is 8 bit, cf. section 5.3.1), then only the lower bits are transferred to the buffer. See section 6.3.10 for details on the message.

5.3.3 TX buffer filling via data generator

The TX data buffer can also be filled via the internal data generator. Message *DatBufTxFillReq* is used to initiate the process. Data filling is possible by either simply counting up from a start value, or pseudo-random data generation with a linear feedback shift register. Additionally, the last 4 words of the generated data can be set to 0 independent of the type of data generation. See section 6.3.9 for details on the message.

5.3.4 Starting transmission of TX data

Data transfer from the TX buffer to the TX PHY does not start automatically with message *DatBufTxWriteReq* or *DatBufTxFillReq*. Instead, it must be initiated by message *DatBufTxStartReq* after filling the buffer. The buffered data can be used multiple times without refilling. After message *DatBufTxStartReq* has been confirmed, the data is available at output *datIfPhy0Out* and transferred whenever the TX PHY core indicates *datIfPhy0Out_ir = 1*. See also section 6.3.11 for details on the message.

5.3.5 Starting reception of RX data

RX data reception is started by message *DatBufRxStartReq*. If issued, the RX data buffer resets its internal counter to zero, is ready to receive new data, and overwrites previously received data when data is transferred from the RX PHY. The number of data words should normally be set in the message, but can also be set to maximum if unclear in advance. After *DatBufRxStartReq* has been confirmed, data is received at input *datIfPhy1In* whenever the RX PHY core indicates *datIfPhy1In_or=1*. See section 6.3.7 for details on the message.

5.3.6 RX buffer read/check via data sink

If data has been created using *DatBufTxFillReq*, then the integrity of the data in the RX buffer can be checked via message *DatBufRxCheckReq*. Parameters must be selected the same way as in message *DatBufTxFillReq* on the TX side. The corresponding confirm in *DatBufRxCheckCfm* returns the status of the data. See section 6.3.8 and section 6.3.35 for details on the messages.

5.3.7 RX buffer read via control interface

Data can be read from the RX buffer via message *DatBufRxReadReq*. Only the lower bits of the corresponding confirm message *DatBufRxReadCfm* are valid if the bit width is less then 16 bit (default 8 bit, cf. section 5.3.1). See section 6.3.6 and section 6.3.33 for details on the messages.

5.4 General purpose IO pins

There are 16 general purpose input / output pins each. They can be read / written via messages *GetGpiReq* / *CfgGpoReq*, see section 6.3.26 and section 6.3.25.

6 Messages

6.1 Overview

The IP core is controlled via the message interface. All messages can be used with the UART or the ctrl lines equivalently, the only exception being WLAN messages *TxImm(A|B)Req*.

The first word of a message is always the message ID. After the message ID, an arbitrary number of parameters can follow, see section 6.3 below. Message IDs are autogenerated and subject to change with a new release without special notice. In order to avoid problems when switching to an updated version of the IP core, one of the files *def_MsgId.h/m* should be used.

Matlab/Octave example implementations of message handlers can be found in the *./msg* directory. In some cases, these handlers transfer parameters directly to the IP core, in other cases, they perform intermediate calculation from more abstract parameters. The handlers make use of function *sendMsg.m* using the generic format

$$cfm = \mathrm{sendMsg}(id, req, nCfm)$$

where req is the request message, $nCfm$ is the length of the expected confirm message, and cfm is the confirm message itself. All values are 16 bit, *req* and *cfm* are row vectors. Variable id is a struct selecting a specific target IP core to deliver the message. It is explained in more detail in section 6.2.

6.2 Targets and forwarding

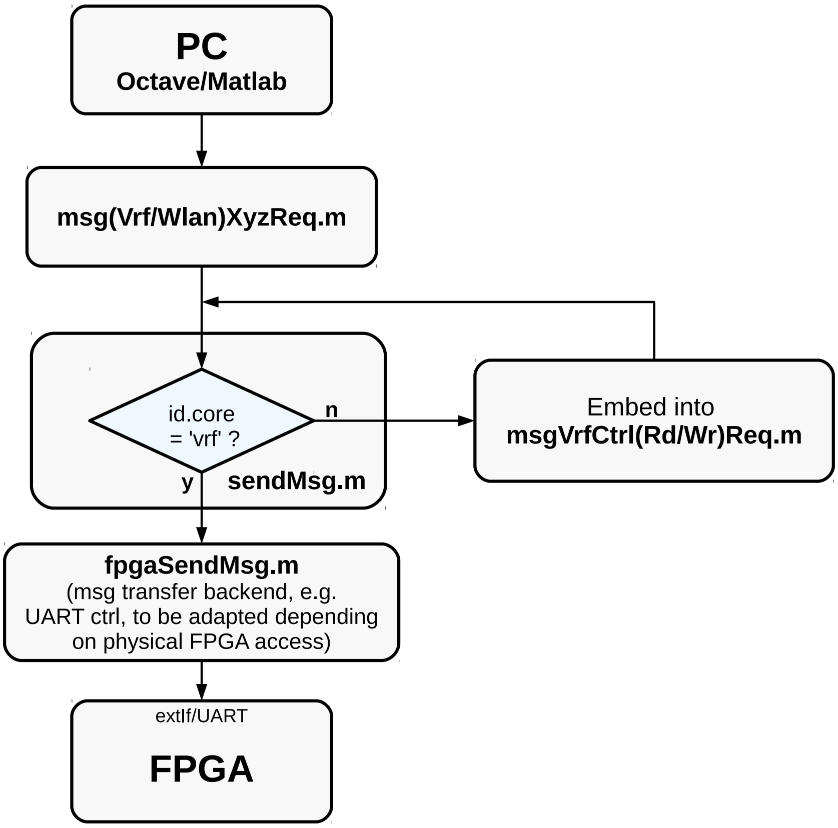

Fig. 23: Message handling and forwarding

Each control message handler passes variable id to function *sendMsg.m*. Variable id is a struct with arbitrary fields used by *sendMsg.m* to identify the target of the message. Function *sendMsg.m* is typically depending on the target the message is sent to (*wlanX* / *vrfX*). It might use the following struct fields:

core: Type of IP core to be addressed, e.g. 'vrf' to configure the VRF, or 'wlan' to configure WLAN PHY cores.

inst: Instance number, e.g., 0, 1, ..., referring the to a specific instance of a certain type of IP core if there is more than one. E.g., in figure 2 there are 2 WLAN IP cores with instance numbers 0 for TX and 1 for RX, respectively.

The *vrfX* example in figure 2 has 3 different targets, *core* = 'vrf' / *inst* = 0, *core* = 'wlan' / *inst* = 0, and *core* = 'wlan' / *inst* = 1. Figure 23 shows how messages are sent to different targets via a single control interface. Variable id may contain additional fields that are used with other (testing) versions of *sendMsg.m*. If *id.core* = 'wlan', function *sendMsg.m* calls *msgVrfCtrl(Rd|Wr)Req.m* with the WLAN message as payload, thereby redirecting the WLAN control message to the *ctrlIfPhy(0|1)(In|Out)* interface in figure 2.

If only a simple Verilog module like *wlanX* with a single WLAN core is used, then function *sendMsg.m* can access the UART (or whatever) interface directly, ignoring all settings of variable id.

6.3 Definitions

6.3.1 Message ID numbering

Message IDs might seem weird as they start at an arbitrary position with some "holes" in between. This is due to test/debug messages that are not distributed with the commercial version of the IP core. All message IDs are autogenerated and may change with a new version of the core. Please use the files *def_MsgId_vrf.h/m* in directory *./msg* or converted versions thereof.

6.3.2 MsgId 24 – ResetReq

Purpose
System reset request

Parameters
None

Description
ResetReq initiates a SW reset of the system. The SW reset might not work in case the system has crashed completely, e.g., due to misconfiguration. In this case, a HW reset via signal *resetIn* is necessary. Different from other messages, *ResetReq* does not have a corresponding confirm.

6.3.3 MsgId 25 – OnOffReq

Purpose
VRF on/off request

Parameters

1. 0=off, 1=on

Description
OnOffReq switches on/off all virtual RF modules. When modules are switched off, their internal state is reset, e.g. WGN generator, frequency shift phase, etc... . Configuration settings are not affected.

6.3.4 MsgId 26 – CtrlWrReq

Purpose
Write to physical layer control interface

Parameters

1. CtrlIf number M (0 or 1)
2. Number of data to follow N
3. Data 0, Data 1, ..., Data N-1

Description
CtrlWrReq writes N words of 16 bit data to interface *ctrlIfPhy[0|1]Out*. The data typically contains a request message sent to a connected PHY IP core.

6.3.5 MsgId 27 – CtrlRdReq

Purpose
Read from physical layer control interface

Parameters

1. CtrlIf number M
2. Number of data to read N

Description
CtrlRdReq reads N words of 16 bit data from the interface *ctrlIfPhy[0|1]In*. The data in *CtrlRdCfm* typically contains a confirm message read from a connected PHY IP core.

6.3.6 MsgId 28 – DatBufRxReadReq

Purpose
Read from RX data buffer

Parameters

1. Number of data words to read

Description
DatBufRxReadReq reads from the RX data buffer starting at buffer address 0. Data is returned in the corresponding *DatBufRxReadCfm*.

6.3.7 MsgId 29 – DatBufRxStartReq

Purpose
Start RX data buffer

Parameters

1. Number of data words to receive

Description
DatBufRxStartReq prepares writing of data from PHY1-RX to the RX data buffer. Data can be received at input *datIfPhy1In* and stored in the buffer after the corresponding *DatBufRxStartCfm*.

6.3.8 MsgId 30 – DatBufRxCheckReq

Purpose

Check RX data buffer

Parameters

1. Number of data
2. First data word
3. Select 0 = count up, 1 = random data
4. Select 0 = normal data, 1 = last 4 words are 0

Description

DatBufRxCheckReq checks if the contents of the RX data buffer is consistent with the data generated by message *DatBufTxFillReq* for transmission.

6.3.9 MsgId 31 – DatBufTxFillReq

Purpose

Fill TX data buffer using the internal generator

Parameters

1. Number of data to write
2. First data word
3. Select 0 = count up, 1 = random data
4. Select 0 = normal data, 1 = last 4 words are 0

Description

DatBufTxFillReq fills the TX data buffer using the internal data generator. Starting with the first data word, the generated data is either simply counted up, or a linear feedback shift register is used for pseudo random generation of the remaining data words.

6.3.10 MsgId 32 – DatBufTxWriteReq

Purpose

Write to TX data buffer

Parameters

1. Number of data to write
2. Data 1st word
3. Data 2nd word
4. Etc...

Description

DatBufTxWriteReq writes arbitrary values to the TX data buffer.

6.3.11 MsgId 33 – DatBufTxStartReq

Purpose

Start TX data buffer

Parameters

1. Number of data

Description

DatBufTxStartReq initiates writing of data from TX data buffer to a connected PHY0-TX core. Data is ready be read from the buffer and passed to output *datIfPhy0Out* after the corresponding *DatBufTxStartCfm*.

6.3.12 MsgId 34 – CfgRxGainTblReq

Purpose
RX gain delay and gain table configuration request

Parameters

1. Gain delay in samples
2. Gain shift (g_s) index 0
3. Gain factor ($g_{f,fxp}$) index 0
4. Indices 1..127 to follow ...

Description
CfgRxGainTblReq configures the AGC table of the virtual RF. 128 different gain configurations are set. The actual gain used during reception is selected by the 7 *gainSel* input lines.

The input data x of module *rxGain* is multiplied as

$$y = x \cdot g_{f,f_fxp} << (g_s - 8)$$

The output is saturated to 32 bit. For practical purposes, only gain factors between 128..255 are useful in order to achieve maximum resolution. The output y is shifted right $>>$ 4 and saturated to 12 bit during further processing before finally being passed to the ADC output of the VRF core.

The gain delay defines how many samples must pass after a change of the *gainSel* lines, before the new gain is applied.

Note that the control message handler *msgVrfCfgRxGainTblReq.m* uses parameters *gainDb* and *gainSel* as inputs and calculates the LUT containing g_{f,f_fxp} and g_s from these values. See section 4.13.4 for details.

6.3.13 MsgId 35 – CfgRxIqImbReq

Purpose
RX IQ imbalance configuration request

Parameters

1. RX IQ imbalance factor a_{fxp}
2. RX IQ imbalance factor b_{fxp}
3. RX IQ imbalance factor c_{fxp}

Description
See *MsgCfgTxIqImbReq* in section 6.3.21.

6.3.14 MsgId 36 – CfgRxDcOffReq

Purpose
RX DC offset configuration request

Parameters

1. RX DC offset real part
2. RX DC offset imag part

Description
See *MsgCfgTxDcOffReq* in section 6.3.20. However, different from TX, the signal level at RX depends on the current gain settings. RX DC offset must therefore be seen relative to this signal level.

6.3.15 MsgId 37 – CfgDacRateReq

Purpose
DAC sample rate configuration request

Parameters

1. DAC sample rate (p), lo 16 bit
2. DAC sample rate (p), hi 16 bit
3. System clock (q), lo 16 bit
4. System clock (q), hi 16 bit

Description
CfgDacRateReq sets the rate at which the VRF DAC input requests a sample from the TX baseband. The VRF sets the input ready signal (*dac_ir*) each p out of q cycles. Signal *dac_ir* is set as regular as possible, e.g. for $p = 2$ $q = 5$, the sequence would be "010010100101001..." . Parameters p and q can assume any value $1..2^{32} - 1$, but $p < q$ must hold. The default is $p = 1, q = 1$, requesting TX data in every clock cycle.

6.3.16 MsgId 38 – CfgMultiPathReq

Purpose
Multipath configuration request

Parameters

1. Number of coefficients ($n_c = 1..10$)
2. Coefficient 0 real part
3. Coefficient 0 imag part
4. Coefficient 0 delay
5. Coefficient 1 real part
6. ...

Description
CfgMultiPathReq sets the coefficients for the FIR filter that models the multipath fading. Coefficients are Q13, i.e. $2^{13} = 1$, and must not exceed a complex magnitude of 2. The coefficient delay can be between 0..29. The order of configuration parameters is $[n_c, c_{0,Re,fxp}, c_{0,Im,fxp}, d_0, c_{1,Re,fxp}, \dots d_{n_c-1}]$.

Note that this message has a variable number of parameters depending on the number of coefficients used. Independent of n_c all previous settings are automatically deleted if the message is used.

6.3.17 MsgId 39 – CfgFreqOffReq

Purpose
Frequency offset configuration request

Parameters

1. Frequency offset lower 16 bit ($f_{r,fxp,LO}$)
2. Frequency offset middle 16 bit ($f_{r,fxp,MI}$)
3. Frequency offset upper 16 bit ($f_{r,fxp,HI}$)

Description
CfgFreqOffReq sets the simulated frequency offset relative to the sample frequency such that

$$freqOffHz = f_{r,fxp}/2^{48} \cdot f_S$$

where f_S is the sampling frequency, and $f_{r,fxp} = [-2^{47} \,..\, +2^{47} - 1]$. Values $f_{r,fxp} > 0$ correspond to a higher carrier frequency at the RX.

In order to provide consistent settings between frequency offset and clock offset for a certain crystal deviation at the receiver with carrier frequency and baseband clock derived from the same source, control message handler *msgVrfCfgFreqOffReq.m* calculates $f_{r,fxp}$ from the given offset in ppm, the carrier frequency and the sampling frequency. See section 4.9.4 for details.

6.3.18 MsgId 40 – CfgClkOffReq

Purpose
Clock offset configuration request

Parameters

1. Clock offset lower 16 bit, $(c_{o32,fxp,LO})$
2. Clock offset middle 16 bit, $(c_{o32,fxp,MI})$
3. Clock offset upper 16 bit, $(c_{o32,fxp,HI})$

Description
CfgClkOffReq sets the simulated clock offset. Parameter $c_{o32} = c_{o32,fxp}/2^{48}$ defines the fraction of a sample by which the sampling time is shifted each 32 samples. Values $c_{o32} < 0$ correspond to a higher clock frequency at the RX, i.e. the sampling time shifts into the negative direction.

In order to provide consistent settings between frequency offset and clock offset for a certain crystal deviation at the receiver with carrier frequency and baseband clock derived from the same source, control message handler *msgVrfCfgClkOffReq.m* calculates $c_{o32,fxp}$ from the given offset in ppm. See section 4.10.4 for details.

6.3.19 MsgId 41 – CfgTxInpScReq

Purpose
TX input scaling configuration request

Parameters

1. TX scaling factor, $tx_f = 0..32767$

Description
CfgTxInpScReq is used to multiply the 12 bit input data x of module *txScale* as

$$y = x \cdot tx_f >> 8$$

TX factor must be set such that output y has exactly 20 dB backoff with respect to the complex valued full scaled data. In other words, given the DAC input data (= TX signal) backoff IBO, tx_f is calculated as

$$tx_f = 4096 \cdot 10^{(IBO-20)/20}$$

See also section 4.4 for details.

6.3.20 MsgId 42 – CfgTxDcOffReq

Purpose
TX DC offset configuration request

Parameters

1. TX DC offset real part
2. TX DC offset imag part

Description
CfgTxDcOffReq configures module *dcOff* to add values dcOff(Re/Im) to the I/Q path. As the signal level (RMS) is 3276.8, the DC offset must be seen relative to this. See also section 4.5 for details.

6.3.21 MsgId 43 – CfgTxIqImbReq

Purpose
TX IQ imbalance configuration request

Parameters

1. TX imbalance factor a_{fxp}
2. TX imbalance factor b_{fxp}
3. TX imbalance factor c_{fxp}

Description
CfgTxIqImbReq configures TX IQ imbalance to apply a matrix multiplication to the input signal x as follows

$$\begin{pmatrix} y_{Re} \\ y_{Im} \end{pmatrix} = \begin{pmatrix} a & c \\ 0 & b \end{pmatrix} \begin{pmatrix} x_{Re} \\ x_{Im} \end{pmatrix}$$

where $a = a_{fxp}/2^{14}$, $b = b_{fxp}/2^{14}$, and $c = c_{fxp}/2^{14}$. Parameters a, b, and c are calculated in control message handler *msgVrfCfgTxIqImbReq.m*, see section 4.6 for details.

6.3.22 MsgId 44 – CfgTxPaReq

Purpose
TX PA configuration request

Parameters

1. Bypass mode 0/1 = off/on

Description
CfgTxPaReq configures power amplifier modeling to be bypassed or not.

6.3.23 MsgId 45 – CfgTxPaLutReq

Purpose
TX PA LUT configuration request

Parameters

1. Amplitude factor $a_{0,fxp}$
2. Phase shift $\varphi_{0,fxp}$

3. Amplitude factor $a_{1,fxp}$
4. Phase shift $\varphi_{1,fxp}$
5. Continue until entry 512 ...

Description

CfgTxPaLutReq configures the LUTs for power amplifier modeling. 513 values for amplitude factor $a_{k,fxp} = 0..32767$ and phase shift $\varphi_{k,fxp} = -32768..+32767$, are transferred to the VRF core. Fixed point values correspond to real factors $a_k = a_{k,fxp}/2^{15}$, and real phase shifts $\varphi_k = \varphi_{k,fxp}/2^{15} \cdot 180°$. Index $k = 0..512$ is selected by the amplitude of the complex valued input signal such that if $|x| = k \cdot 64$ the corresponding LUT values are chosen. For input amplitudes other than $k \cdot 64$, the LUT outputs are interpolated linearly. See also section 4.7 for details.

6.3.24 MsgId 46 – CfgChGainReq

Purpose

Channel gain configuration request

Parameters

1. Channel gain factor $g_{f,fxp} = 0..255$
2. Channel gain shift, $g_s = -32..+18$

Description

CfgChGainReq is used to multiply the 16 bit input data x as

$$y = x \cdot g_{f,fxp} \cdot 2^{g_s-8}$$

The output is saturated to 32 bit. For practical purposes, only gain factors between 128..255 are useful in order to achieve maximum resolution. The complex 16 bit input is assumed to have a full scale backoff of 20 dB, corresponding to an RMS of 3276. The following WGN generator produces noise with complex RMS = 796, hence the SNR for zero gain ($g_f = 128$, $g_s = 1$) is about 12.3 dB. Message handler *msgVrfCfgChGainReq.m* uses the target SNR as input parameter and sets $g_{f,fxp}$, g_s accordingly. See section 4.11 for details.

6.3.25 MsgId 47 – CfgGpoReq

Purpose

General purpose output configuration request

Parameters

1. Dat
2. Mask

Description

CfgGpoReq writes the bits in *Dat* to the *gpo* output lines. Only bits where the corresponding *Mask* bit is set are written, all others keep their original values.

6.3.26 MsgId 48 – GetGpiReq

Purpose

General purpose input read request

Parameters

1. Mask

Description

GetGpiReq reads the bits of the *gpi* input lines. Only bits where the corresponding *Mask* bit is set are read, all others return 0.

6.3.27 MsgId 49 – VersionReq

Purpose
Version number read request

Parameters
None

Description
VersionReq reads the version number and evaluation flag, which are returned in *VersionCfm*.

6.3.28 MsgId 50 – LedBlinkReq

Purpose
Led the LEDs blink for a moment

Parameters

1. Blink period in cycles / 2^{20}

Description
LedBlinkReq lets all the LEDs blink 4 times.

6.3.29 MsgId 55 – BootCfm

Purpose
Confirm system reset/boot

Parameters
None

Description
BootCfm confirms system boot. The message is sent after the booting and must be read by the host. Other messages must not be sent before *BootCfm* indicates the end of the boot process.

6.3.30 MsgId 56 – OnOffCfm

Purpose
Confirm VRF on/off request

Parameters
None

Description
OnOffCfm confirms the *OnOffReq*.

6.3.31 MsgId 57 – CtrlWrCfm

Purpose
Confirm ctrl write request

Parameters
None

Description
ExIfWriteCfm confirms the *CtrlWrReq*.

6.3.32 MsgId 58 – CtrlRdCfm

Purpose
Confirm ctrl read request

Parameters

1. First word read from ctrl interface
2. Second word read from ctrl interface
3. Etc...

Description
CtrlRdCfm confirms *CtrlRdReq* and returns the data.

6.3.33 MsgId 59 – DatBufRxReadCfm

Purpose
Confirm RX data buffer read request

Parameters

1. First word read from RX data buffer
2. Second word read from RX data buffer
3. Etc...

Description
DatBufRxReadCfm confirms *DatBufRxReadReq* and returns the data.

6.3.34 MsgId 60 – DatBufRxStartCfm

Purpose
Confirm RX data buffer start request

Parameters
None

Description
DatBufRxStartCfm confirms *DatBufRxStartReq*.

6.3.35 MsgId 61 – DatBufRxCheckCfm

Purpose
Confirm RX data buffer check request

Parameters

1. Error flag: 0=OK, 1=error

Description
DatBufRxCheckCfm confirms *DatBufRxCheckReq* and returns the error flag.

6.3.36 MsgId 62 – DatBufTxFillCfm

Purpose
Confirm TX data buffer fill request

Parameters
None

Description
DatBufTxFillCfm confirms *DatBufTxFillReq*.

6.3.37 MsgId 63 – DatBufTxWriteCfm

Purpose
Confirm TX data buffer write request

Parameters
None

Description
DatBufTxWriteCfm confirms *DatBufTxWriteReq*.

6.3.38 MsgId 64 – DatBufTxStartCfm

Purpose
Confirm TX data buffer start request

Parameters
None

Description
DatBufTxStartCfm confirms *DatBufTxStartReq*.

6.3.39 MsgId 65 – CfgRxGainTblCfm

Purpose
Confirm RX gain table configuration request

Parameters
None

Description
CfgGainTblCfm confirms the RX gain table configuration.

6.3.40 MsgId 66 – CfgRxIqImbCfm

Purpose
Confirm RX IQ imbalance configuration request

Parameters

1. Status: 0 = OK, 1 = Message not supported

Description
CfgRxIqImbCfm confirms RX IQ imbalance configuration request.

6.3.41 MsgId 67 – CfgRxDcOffCfm

Purpose

Confirm RX DC offset configuration request

Parameters

1. Status: 0 = OK, 1 = Message not supported

Description

CfgRxDcOffCfm confirms RX DC offset configuration request.

6.3.42 MsgId 68 – CfgDacRateCfm

Purpose

Confirm DAC rate configuration request

Parameters

1. Status: 0 = OK, 1 = Message not supported

Description

CfgDacRateCfm confirms DAC rate configuration.

6.3.43 MsgId 69 – CfgMultiPathCfm

Purpose

Confirm multipath configuration request

Parameters

1. Status: 0 = OK, 1 = Message not supported

Description

CfgMultiPathCfm confirms frequency offset configuration.

6.3.44 MsgId 70 – CfgFreqOffCfm

Purpose

Confirm frequency offset configuration request

Parameters

1. Status: 0 = OK, 1 = Message not supported

Description

CfgFreqOffCfm confirms frequency offset configuration.

6.3.45 MsgId 71 – CfgClkOffCfm

Purpose

Confirm time offset configuration request

Parameters

1. Status: 0 = OK, 1 = Message not supported

Description

CfgClkOffCfm confirms time/clock offset configuration.

6.3.46 MsgId 72 – CfgTxInpScCfm

Purpose
Confirm TX input scale configuration request

Parameters
None

Description
CfgTxInpScCfm confirms TX input scaling configuration request.

6.3.47 MsgId 73 – CfgTxDcOffCfm

Purpose
Confirm TX DC offset configuration request

Parameters

1. Status: 0 = OK, 1 = Message not supported

Description
CfgTxDcOffCfm confirms TX DC offset configuration request.

6.3.48 MsgId 74 – CfgTxIqImbCfm

Purpose
Confirm TX IQ imbalance configuration request

Parameters

1. Status: 0 = OK, 1 = Message not supported

Description
CfgTxIqImbCfm confirms TX IQ imbalance configuration request.

6.3.49 MsgId 75 – CfgTxPaCfm

Purpose
Confirm TX PA configuration request

Parameters

1. Status: 0 = OK, 1 = Message not supported

Description
CfgTxPaCfm confirms PA configuration request.

6.3.50 MsgId 76 – CfgTxPaLutCfm

Purpose
Confirm TX PA LUT configuration request

Parameters

1. Status: 0 = OK, 1 = Message not supported

Description
CfgTxPaLutCfm confirms PA LUT configuration request.

6.3.51 MsgId 77 – CfgChGainCfm

Purpose
Confirm channel gain request

Parameters
None

Description
CfgChGainCfm confirms channel gain setting configuration request.

6.3.52 MsgId 78 – CfgGpoCfm

Purpose
General purpose output configuration confirm

Parameters

1. Updated GPO value

Description
CfgGpoCfm confirms execution of *CfgGpoReq* and returns the updated GPO value.

6.3.53 MsgId 79 – GetGpiCfm

Purpose
General purpose input read confirm

Parameters

1. GPI input data (masked)

Description
GetGpiCfm confirms execution of *GetGpiReq* and returns the masked input value.

6.3.54 MsgId 80 – VersionCfm

Purpose
Confirm version request and send version number

Parameters

1. Major version
2. Branch version
3. Tag version
4. Evaluation flag: 0 = full version, 1 = eval version

Description
VersionCfm returns the 3 digit version number and the evaluation flag in response to a *VersionReq*.

6.3.55 MsgId 81 – LedBlinkCfm

Purpose
Confirm LED blink request

Parameters
None

Description
LedBlinkCfm confirms execution of *LedBlinkReq*.

7 License

7.1 General

The Verilog sources, data files, message example applications – also referred to as the "code" – and the documentation distributed in this package are under copyright.

By downloading and using the code, you accept the license terms defined in this section.

7.2 Limited liability

You accept that the code comes without warranty for any particular purpose. The copyright owner will not be liable for any damage caused by the code.

7.3 Restrictions

You are not allowed to copy or redistribute the code.

You are not allowed to change the code with exception of minor modifications that do not change the original functionality, e.g., replacing memory models or fixing synthesis problems. You accept that these minor modifications do not lay foundation to any copyright on your side.

You are not allowed to remove the functional restrictions of the evaluation version.

7.4 Terms of use

This is an evaluation version with some functional and legal restrictions. If you consider using the VRF IP core for commercial, non-commercial, academic, military, private or whatever purposes, you are allowed to evaluate this version.

Evaluation allows you to

- inspect the sources
- synthesize the code for FPGA or similar
- run the vrfX example module
- embed the code into a larger system with your own physical layer IP cores

all exclusively for the purpose of testing and evaluating the IP core functionality.

Before you start using the core within your project in order to test and optimize your own physical layer IP, you must obtain a commercial license.