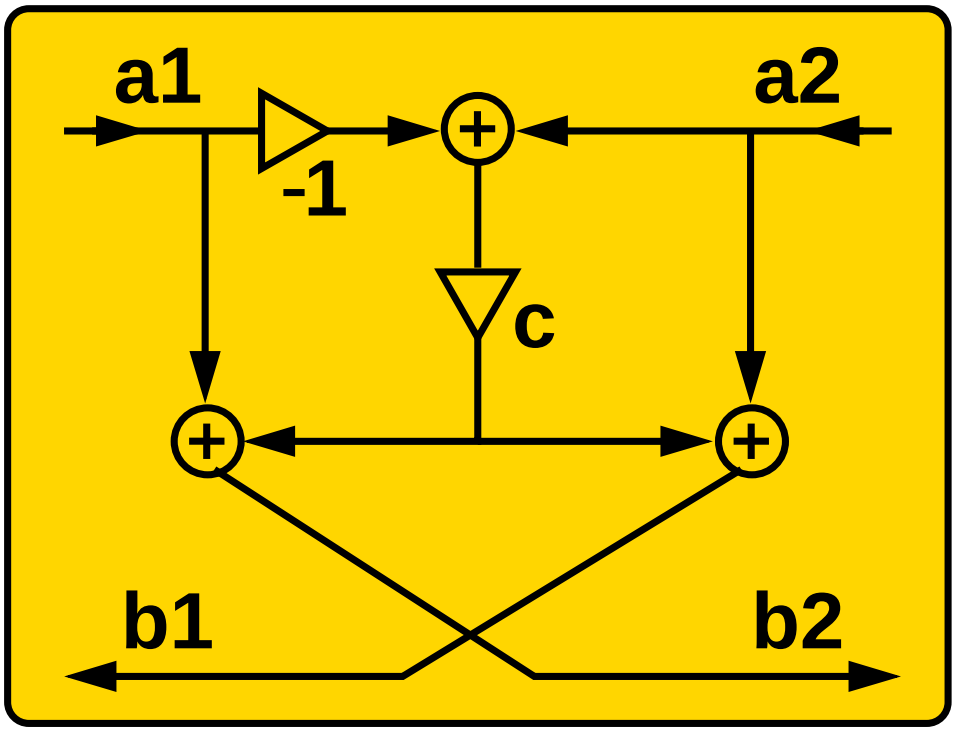
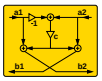
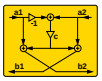


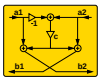
802.11a Physical Layer IP

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 802.11a Physical Layer Baseband IP core.

1.2 Delivery file structure

The delivery contains 6 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
reg: Module regression tests (full version only)
- reg/top: Top level tests
- reg/mod: Module level tests

src: Verilog sources
- src/common: Miscellaneous general purpose modules
- src/modules: Signal processing and control modules
- src/top: The top module *wlan.v* and a corresponding instance definition *wlan_0.v* that can be included into the Verilog file containing the IP block

tst: Test build examples + Matlab/Octave test scripts
- 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 test scripts or setup + test (single core)

1.3 Features

The BAY9 802.11a IP core allows to transmit and receive data packets according the sections 17.1-17.3 of [1]. The TX contains everything from the data byte input to the DAC signal outputs. For RX, the ADC input data is acquired, synchronized, decoded and provided at the data byte output. CRC-32, digital up/down conversion, and RF control are included in addition.

Main features summary:

- Fully IEEE 802.11a standard compliant, 802.11p is possible using half the clock frequency
- Support for all data rates 6-54 Mbit/s
- Written in Verilog without vendor specific IP cores
- Synthesis possible for Altera and Xilinx FPGAs without source code changes
- 80 MHz target frequency, runs on Virtex-4, Stratix II or faster devices
- Generic message interface for configuration, operation, and debugging
- Separate interfaces for control messages and data input/output
- RF control signals included (PA/TX/RX on/off, up to 16 parallel AGC lines, 3-wire serial bus)
- Generic AGC algorithm allows simple configuration via message interface
- Digital up and downconversion to/from 80 MS/s DAC/ADC interface for zero IF or 20 MHz low IF mode
- Message control via UART and internal data generator for PHY-only debugging directly from a PC

The core also contains an 802.11b transmitter, but no receiver yet.

1.4 Overview

The functional IP overview is shown in figure 1. The minimum set of I/O lines that needs to be connected for a functional test is marked in red. The Verilog top module is *wlan*.

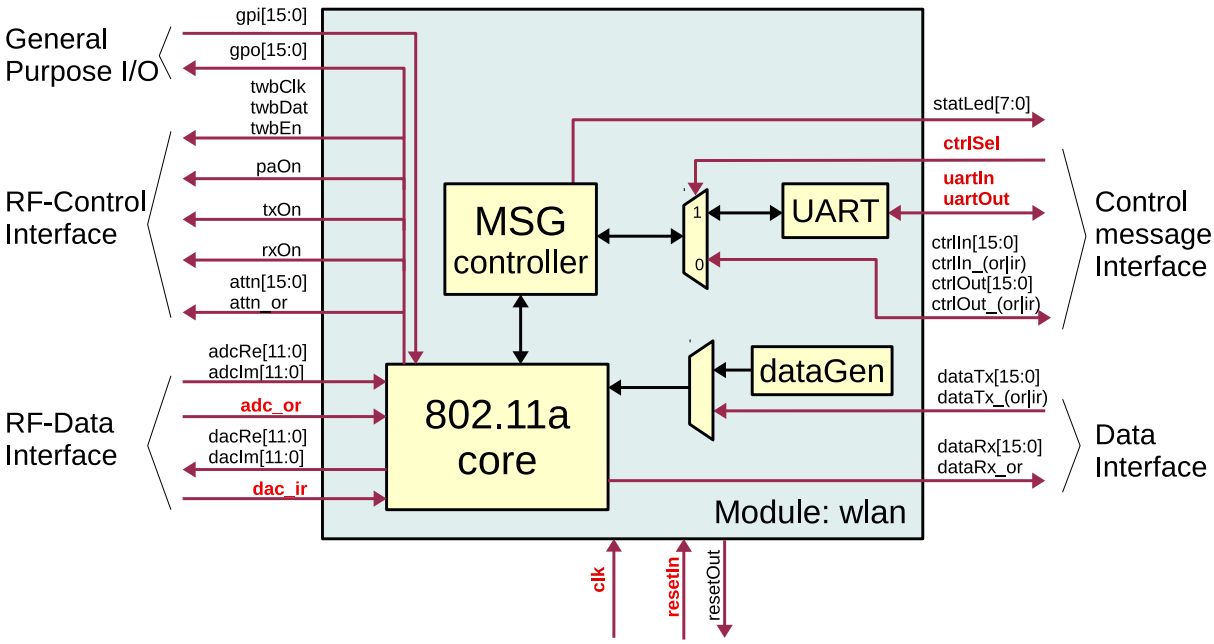


Fig. 1: 802.11a IP system overview

System:
System clock and reset are provided by the *clk* and *resetIn* signals. The *resetOut* signal indicates that the system is still in reset state after a HW or SW reset. Reset signals are active high. The system needs a minimum clock frequency of 80 MHz, but can operate at any higher clock frequency if required.

Host interface:
TX data bytes need to be provided at the 16-bit input *dataTx*, while RX output data is received at output *dataRx*. Alternatively, TX input data can be generated by the internal *dataGen* block for testing.

Control messages are sent or received either via the parallel 16-bit *ctrl(In/Out)* interface with handshake lines, or via *uart(In/Out)*. The *ctrlSel* input switches between the two control interfaces.

RF data interface:
Data signal input and output are handled via *adc(Re|Im)* and *dac(Re|Im)*. The interface operates at 80 MS/s, i.e. oversampling by a factor 4 is applied. ADC/DAC data is 12 bit and 2's complement, range [-2048..2047]. Data is read from *adc(Re|Im)* or written to *dac(Re|Im)* whenever signals *adc_or = 1* or *dac_ir = 1*, respectively. The handshake signals are needed for system clock frequencies above 80 MHz. For exactly 80 MHz, *adc_or* and *dac_ir* must be 1 permanently.

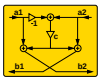
RF control interface:
The RF is controlled by *rxOn*, *txOn*, and *paOn*, to switch on/off the RX path, the TX path, and the power amplifier, respectively. The 3-wire serial bus used in many RF chips to control internal settings is mapped to *twb(Clk|Dat|En)*. AGC is provided by 16 parallel attenuation lines *attn* and a corresponding output ready signal.

Other:
Status LEDs indicate the internal state of the IP core. 16 general purpose output signals *gpo* can be programmed arbitrarily via the message interface. Similar, up to 16 general purpose input signals *gpi* can be read.

1.5 Evaluation vs. full featured version

The evaluation version differs from the full version as follows:

- Packet reception is restricted to 6 Mbit/s for arbitrary packet length, and 54 Mbit/s at length 1000 bytes. The TX path is fully functional.
- Most file header descriptions are removed from the sources in the evaluation version.
- Module regression tests are not provided.



2 Getting started

2.1 Overview

This section provides an example how to:

- Synthesize the WLAN IP core
- Boot and configure the core
- Test basic messages and TX
- Transmit data packages between 2 cores via a virtual RF and evaluate the results

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

- An example Verilog module *wlanX* where the WLAN core is embedded in a minimum setup
- Bash scripts to synthesize module *wlanX* using Altera Quartus, Xilinx Vivado, and Xilinx ISE
- Example control message handlers in Octave/Matlab to access the WLAN IP core
- An example Octave/Matlab backend function to send messages to the FPGA via UART

None of the above is needed to use the WLAN IP core. Rather, the core will typically be included into a customer specific system, 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 *wlanX* example module running. Module *wlanX* is shown in figure 3 and explained in more detail in section 2.4 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 2, 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 Minimum setup

Basic system test is possible by connecting only 7 signals, *clk*, *resetIn*, *ctrlSel*, *uartIn*, *uartOut*, *adc_or*, and *dac_ir*. An example is given in figure 2. Applying the system clock and setting *ctrlSel*=1 (UART control) allows booting and control of the internal logic. The UART signals can be connected directly to a PC (or any other host) with a UART interface. Board resets are typically active low, so this signal might need to be inverted to fit the active high *resetIn* input. DAC/ADC handshake must be set *dac_ir*=1, *adc_or*=1 in order to use TX + RX messages.

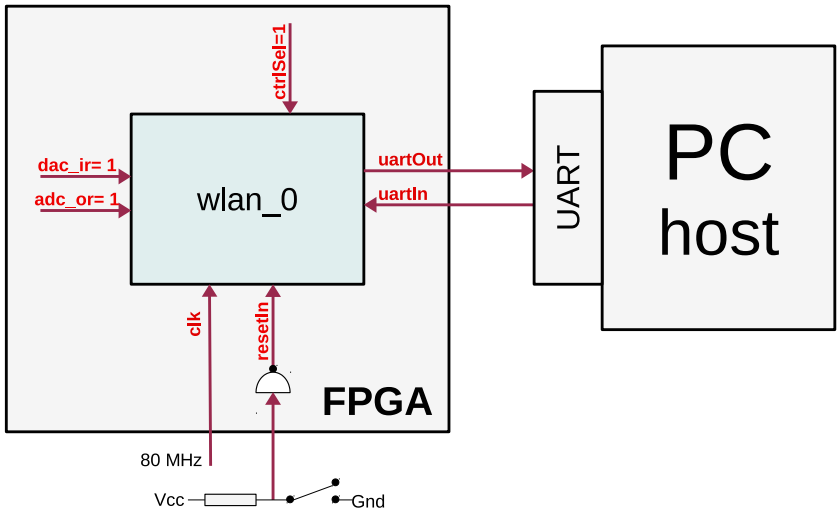


Fig. 2: Minimum test setup example

2.4 Example module *wlanX*

2.4.1 Description

For basic testing it is useful to embed the WLAN core into another system. An example setup is displayed in figure 3. Top level module *wlanX* adds a PLL, clock/reset lines, an LED output, a UART interface, and a mux/demux network to the WLAN IP core.

The mux/demux network connects all unused outputs to all unused inputs. While this is useless functionally, it prevents synthesis tools from removing internal logic during optimization, so the synthesis output provides real resource requirements and the achievable clock frequency.

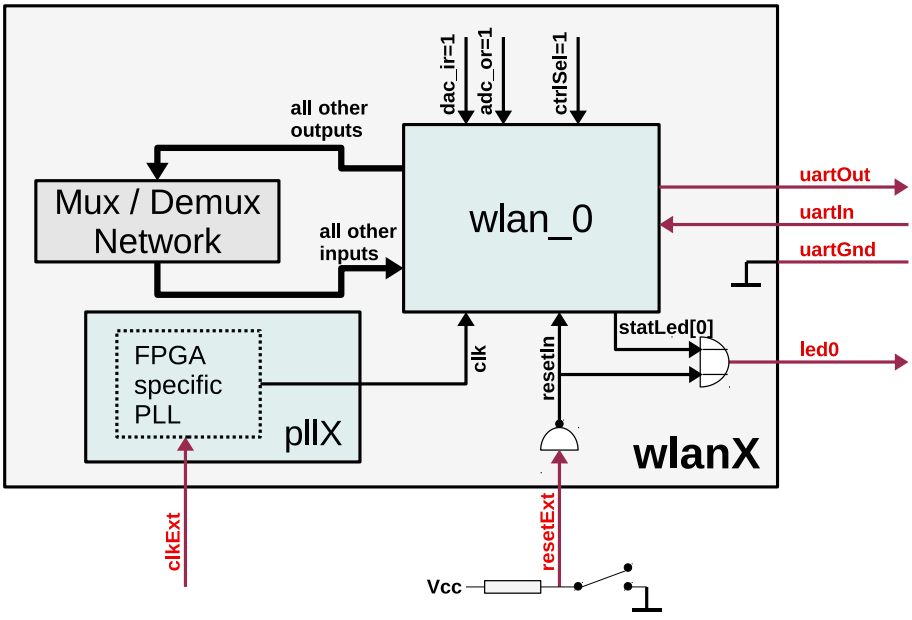
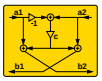


Fig. 3: Minimum example wrapper *wlanX* for top module *wlan_0*



2.4.2 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 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
- led0:	LED1-1

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
- led0:	Green LEDG[0]

Due to limitations of the Cyclone IV, the IP core runs at half the normal clock frequency in this setup (40 MHz instead of 80 MHz, 802.11p mode). Therefore, all timings must be multiplied by a factor 2 when compared to the description in this document.

2.4.3 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.

2.4.4 Reset

Signal *resetExt* of test module *wlanX* signal is active low. Note that this is different from the internal WLAN core signal *resetIn* which is active high, cf. figure 3. A short pulse *resetExt* = 0 resets the system. See also section 3.1 for details.

2.5 Booting and basic messages

2.5.1 Booting and version

After reset, the IP 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/msgWlanBootReq.m*:

```
$ cd ./tst/m
$ octave --traditional --quiet
>> def_MsgId_wlan;
>> msgWlanBootReq;
Booting WLAN OK;
>> msgWlanVersionReq;
Version A.B.C
>>
```

Successful boot is indicated by

- switching on *led0*
- a *BootCfm* message sent from the IP core via the (UART) control interface, cf. section 6.3.30

The *BootCfm* must be read by the host, which is done in *msgWlanBootReq.m*. In the example above, an additional *VersionReq* message (see section 6.3.25 and section 6.3.54) is sent.

After booting, the IP 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.

2.5.2 Basic TX test

TX testing is possible if *wlanX* is extended such that the *dac(Re|Im)* and/or the *paOn/txOn* lines are connected to a logic analyzer, oscilloscope or similar and monitored. Using module *wlanX* as provided, only the messages connected with a TX request can be tested. In order to start a TX, the following messages need to be sent:

- *CfgTxDataSrcReq*
  Selects the internal data generator and sets up the number of TX bytes, cf. section 6.3.17
- *TxReq* in timer mode 0
  Starts a transmit with the same number of bytes defined in *CfgTxDataSrcReq* and an arbitrary TX mode. The timing values are not important in timer mode 0, cf. section 6.3.5

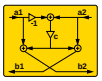
While message *CfgTxDataSrcReq* responds with the corresponding confirm only, the *TxReq* is followed by a *TxStartCfm* and a *TxEndCfm*, see figure 7 in section 4.2.

The default settings after booting use zero IF and -10 dB TX backOff, the *paOn/txOn* lines and the BB output start as fast as possible after reception. It is possible to change this behavior with messages *CfgTxScalingReq* (section 6.3.15), *CfgBandSelReq* (section 6.3.18), and *CfgTxTimingReq* (section 6.3.16).

2.5.3 Basic RX test

Without applying a valid 802.11a signal at the input, a complete RX test is not possible, internal TX-RX loopback is not available due to shared use of the FFT.

If an *RxReq* (section 6.3.6) is sent by the host without an input signal being present, the core confirms the start with *RxStartCfm* (section 6.3.33), then returns with an *RxAcqEndCfm* (section 6.3.34) after the given time (timer mode 1 + 2), or in response to an *RxAcqStopReq* (timer mode 0). See figure 10 in section 4.3 for details.



2.5.4 IP core init + TX test

In order to boot and initialize module *wlanX*, function *setupWlan.m* can be invoked. After a short blinking of LED0 the following output can be seen:

```
>> setupWlan
-------------------------------------
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
```

The following simple test the creates 10 TX frames with random length + modulation scheme:

```
>> testTx11g

+--------+---+---+------+
|   No   | S | M |  Len |
+--------+---+---+------+
|      0 | 1 | 7 | 2681 |
|      1 | 0 | 0 |  172 |
|      2 | 0 | 2 | 2089 |
|      3 | 0 | 1 | 3377 |
|      4 | 1 | 7 |  916 |
|      5 | 0 | 5 | 1645 |
|      6 | 1 | 1 | 3038 |
|      7 | 0 | 1 | 3828 |
|      8 | 0 | 6 | 3102 |
|      9 | 1 | 5 |  699 |
```

Parameter S refers to the subsystem. $S = 0$ is 802.11a, $S = 1$ is 802.11b. Parameter M is the mode, $M = 0..7$ corresponds to 6, 9, ..., 54 MBit/s for 802.11a, $M = 1..7$ refer to mode 1l, 2s, 2l, 5.5s, 5.5l, 11s, 11l of 802.11b, where "s" and "l" stand for short and long header.

2.6 Test build and run including Virtual RF

In order to test the real world behaviour of RX and TX, it is possible to integrate the 802.11a IP core with a virtual RF (VRF) emulator. An overview is shown in figure 4. Module *vrfX* consists of

- 2 instances of the WLAN 802.11a IP cores (*wlan_0 + wlan_1*) used for TX and RX, respectively
- A reduced version of a Virtual RF core providing
  - Channel gain setting
  - Additive white gaussian noise
  - RF gain control based on an arbitrarily defined gain table

The necessary files are not contained in this package. Please download the evaluation version of the VRF IP core from *http://www.bay9.de/products_vrf.html* and follow the documentation therein.

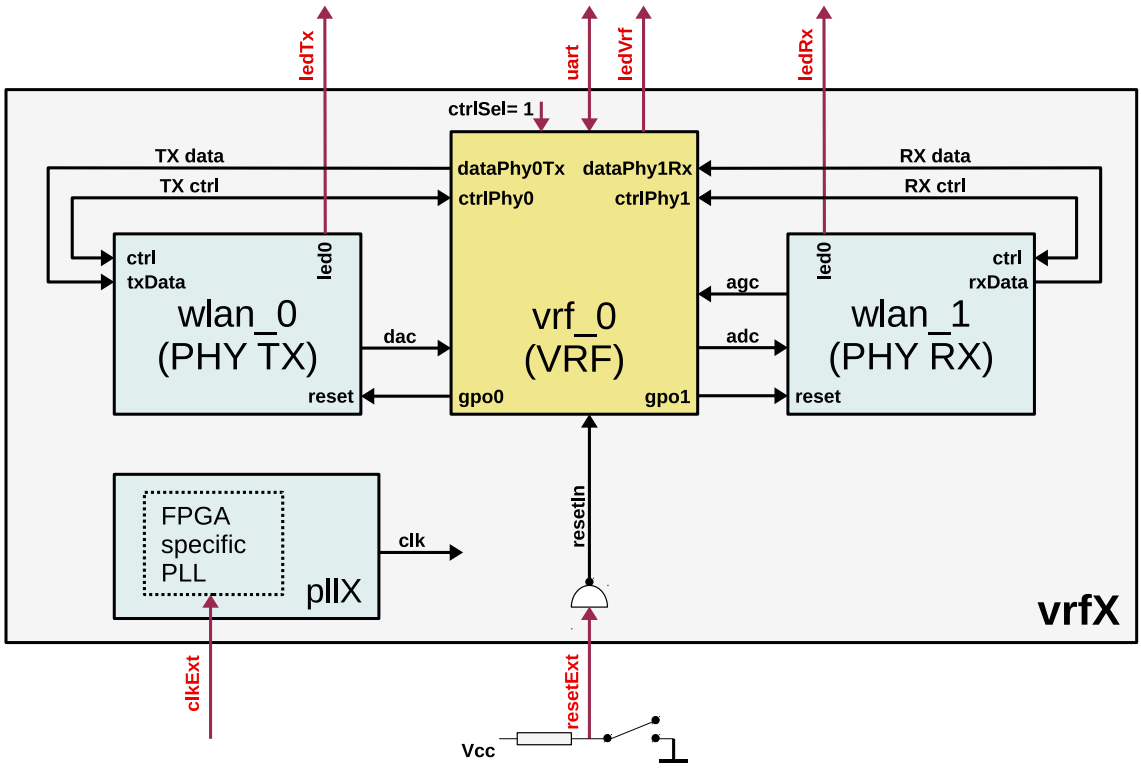
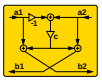


Fig. 4: Virtual RF wrapper *vrfX* including *vrf_0* and IP core instances *wlan_0/1*



3 Description – General aspects

3.1 Reset

3.1.1 Overview

The WLAN 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 *wlanX* in figure 3 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 WLAN 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 6.3.30) is sent from the WLAN 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 6.3.30

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 2.5.1 provides an example how to boot via UART using Octave/Matlab function *msgWlanBootReq.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 *WlanUartDivider_C* in file *./src/common/instances/def_Const_wlan.v* allows to change the speed. It must be set to

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

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.

3.4 Control messages and configuration

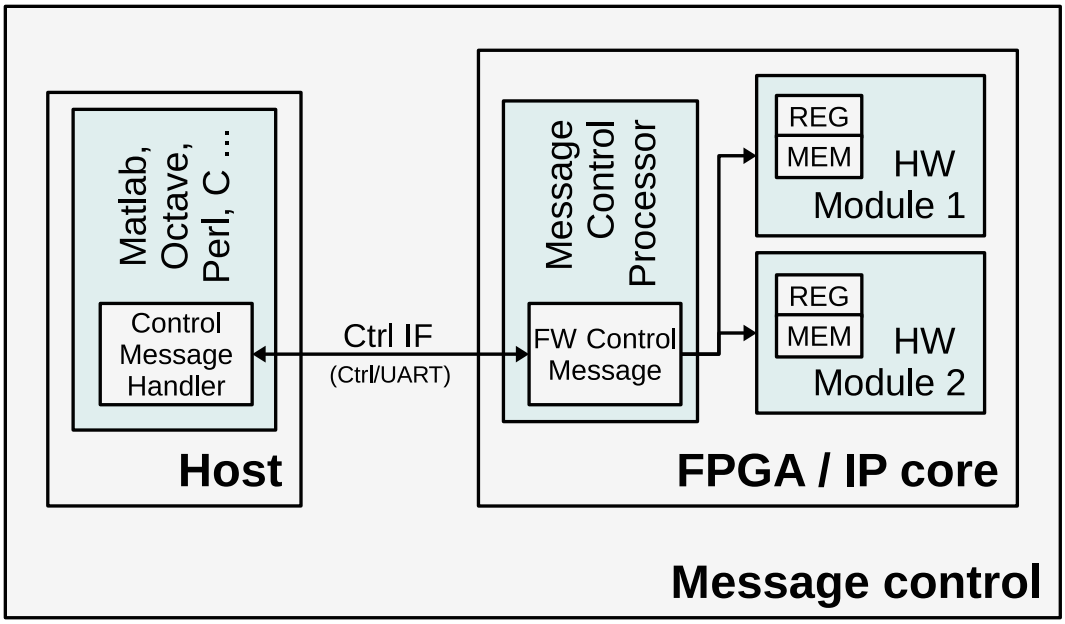


Fig. 5: Configuration via control messages

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 5. 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 WLAN 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 WLAN 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*.

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 6.

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.

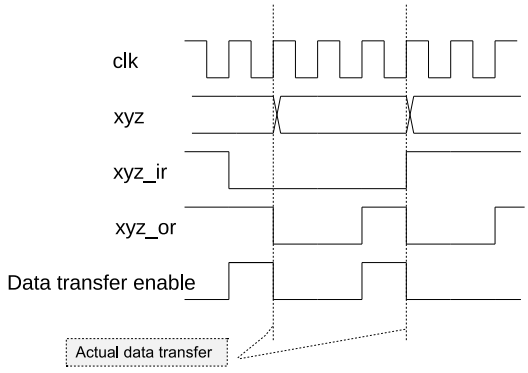


Fig. 6: Data transfer

3.5.4 Summary

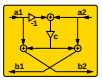
In order to connect an IP core input (*xyzIn*) with a corresponding output (*xyzOut*) of a connecting external module assumed to have the same type of interface (or v.v.), connection must be made as

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

Data transfer signaling is depicted in figure 6. 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 all other cases, one of the signals is omitted. E.g., for the ADC connection, there is only the *adc_or* input available. While the IP core logic can wait for valid data from the ADC (if the system clock is higher than the interface clock) indicated by *adc_or*=1, the ADC cannot wait for the IP core. Details are given in the sections below.



3.6 Data interface

3.6.1 Overview

Input and output data is transferred 16-bit wise via the interfaces *dataTx* and *dataRx*, respectively. The first byte is in the lower 8 bit of the 16-bit word. In case an odd number of bytes is transferred, the upper 8 bit of the last 16-bit word are ignored at TX / invalid at RX.

3.6.2 TX data input

The TX input interface requests input data shortly after the *TxReq* message has been sent and indicates this by setting *dataTx_ir* = 1, see section 4.2 for details of the TX control flow. Data is read at the following positive clock edge. Different from the *ctrlIn* control interface, *dataTx* does not wait for data. Instead, it assumes data to be available for reading when *dataTx_ir* goes high. The host must supply this data once it has started TX processing.

Handshake line *dataTx_or* is currently unused.

In order to create a correct CRC-32 of the payload data, the last 4 bytes must be zero. Transmission works correctly even if the last 4 bytes are non-zero, however, the CRC-result of message *RxEndCfm* in section 6.3.36 will be wrong.

3.6.3 RX data output

The RX output interface starts transmitting data some microseconds after a valid header detection (*RxHdrCfm* message) has been sent to the host and indicates this by setting *dataRx_or* = 1, see section 4.3 for details of the control flow. Data is written at the following positive clock edge. Different from the *ctrlOut* control interface, *dataRx* does not wait to write the data. Instead, it assumes the host to be able to accept data as soon as it is available for writing, i.e., when *dataRx_or* goes high. The host must accept this data once it has received the *RxHdrCfm* message.

3.7 ADC/DAC interface

Signals *adc(Re|Im) / dac(Re|Im)* operate at 80 MS/s. Internal interpolation / decimation filtering by a factor 4 with respect to the 802.11a baseband sample rate of 20 MS/s is provided to simplify analog filtering. For the default minimum system clock of 80 MHz, *adc(Re|Im) / dac(Re|Im)* read/write a data sample in each clock cycle. The handshake lines *adc_or* and *dac_ir* must be set to 1 continuously in this case.

Because internal processing is data driven, arbitrary system clocks higher than 80 MHz can be used. In this case, the ADCs/DACs must be connected through a FIFO or similar to signals *adc(Re|Im) / dac(Re|Im)* and will not read/write a data sample in each clock cycle. Valid reads/writes from/to the IP cores DAC outputs / ADC inputs must be indicated by setting *dac_ir* = 1 / *adc_or* = 1.

4 Frame transmission and reception

4.1 TX/RX message control flow and timing

4.1.1 Overwiew

IP core processing is initiated by sending a message via the control interface. There are 3 types of messages: control, TX, and RX. All messages are described in detail in section 6. While control requests have a corresponding confirm that is returned directly after the execution of the message, TX and RX requests involve a state machine described in section 4.2 and section 4.3, respectively. The host must implement the counterpart of these state machines.

In any case, sending a new request is only allowed after the previous request has been processed by the system and confirmed. There is no way to interrupt a request with exception of *RxAcqStopReq*, see section 4.3.

For all messages, example message handler functions are implemented in Matlab/Octave. They can be found in directory *./msg*. In case of control messages, these functions handle the request and the corresponding confirm. In order to deal with the complexity of the RX/TX state machines in different situations, separate request and confirm message handler functions are provided for TX and RX, while combined functions *msg...Run.m* handle the state machines. Details are described in section 4.2 and section 4.3.

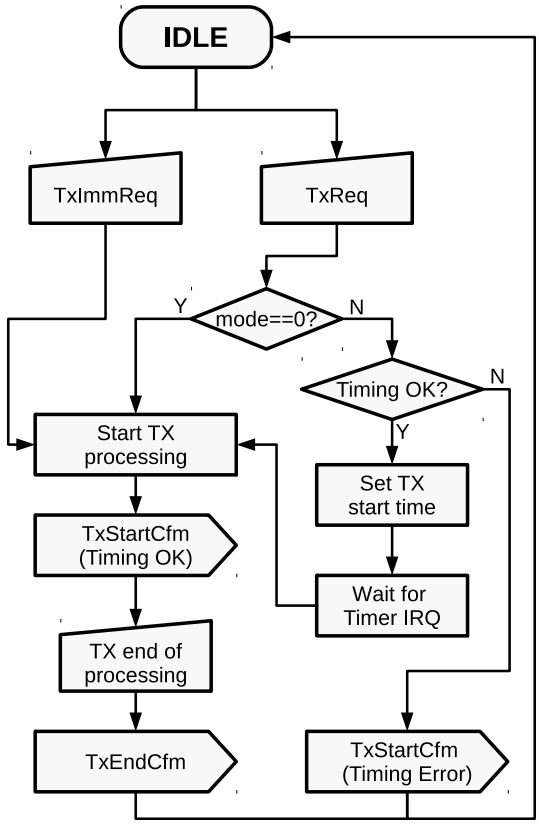


Fig. 7: TX processing control flow

4.2 TX request

General remark

The description in this chapter refers to the *TxImmReq*, which means using one of *TxImmAReq* and *TxImmBReq*, reflecting the inclusion of the 11b-transmitter functionality with separate TX immediate messages. Despite the

TX11b being included into the system, its use is strongly discouraged. The feature is not yet tested properly, and the corresponding receiver is still under development.

4.2.1 Control flow

The message control flow for TX is depicted in figure 7. The host initiates a transmission by sending a *TxReq* or *TxImmReq* message. The state machine (for normal *TxReq*, not *TxImmReq*) is handled by function *msgTxRun.m*.

In case of *TxImmReq* or *TxReq* in timer mode 0, the IP core starts the TX sequence as fast as possible, i.e., after 0.25μs for *TxImmReq* and after 1μs for *TxReq*. The core responds with a *TxStartCfm* and "Timing OK" indication. It then waits for the end of TX processing and additionally returns a *TxEndCfm* after the frame has been sent.

In case of a *TxImmReq*, the TX sequence is started immediately upon the reception of the message ID word and the host must complete the message (length, mode) directly after that, i.e., as fast as possible within the next cycles. For that reason, *TxImmReq* cannot be used via the UART. Parameter transmission at UART speed would take too much time.

For *TxReq*, the system first waits for the complete message to be received and starts the TX sequence afterwards.

If the host uses *TxReq* in timer mode 1 or 2 (i.e., start after or at a given time, cf. section 6.3.5), the timing is checked first. If the calculated start time is sufficiently in the future (≥ 31 baseband timer ticks or 1.55μs), the TX procedure continues as described above at the requested time. Otherwise, a *TxStartCfm* with a "Timing Error" is returned, but no *TxEndCfm*.

Figure 8 shows the begin of the TX start sequences in the different cases.

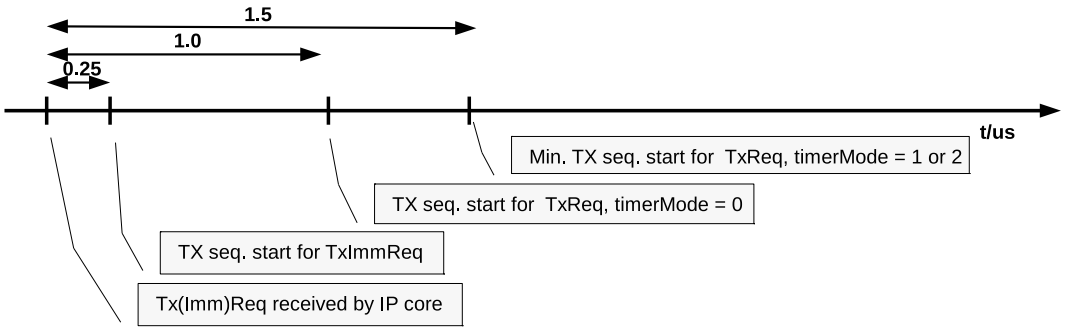


Fig. 8: TX start timing after TX request message

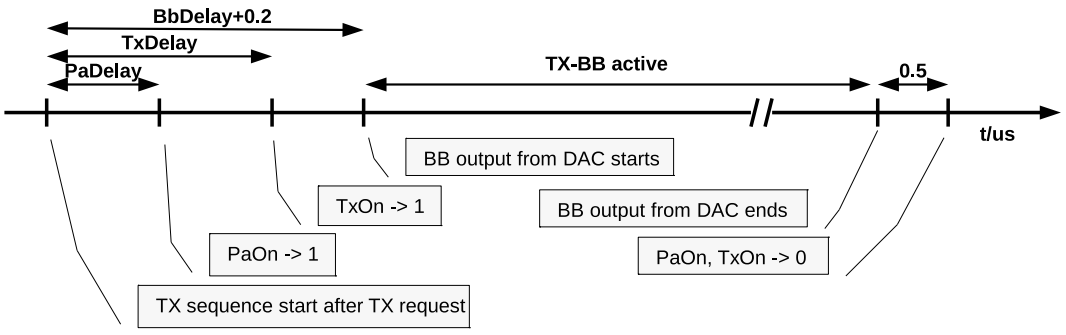


Fig. 9: TX sequence start and end timing

4.2.2 Start sequence timing

A transmission begins with a TX start sequence as shown in figure 9. It depends on parameters *PaDelay*, *TxDelay*, and *BbDelay* set by the *CfgTxTimingReq* message. If all parameters are set to 1, the *txOn* and *paOn* lines go high immediately, and the DAC BB output starts about 0.2μs later. All 3 timings can be delayed individually by using the *CfgTxTimingReq* message, cf. section 6.3.16. Once adjusted, these additionaly delays are fixed and independent of the timing given in the *TxReq* message.

4.2.3 Number of bytes

Although arbitrary transmissions from 1-4095 bytes are possible in principle, the last 4 bytes must always be zero in order to get a correct CRC result at the receiver. If less than 5 bytes are transmitted, the CRC will show an error even if data is received correctly.

4.3 RX request

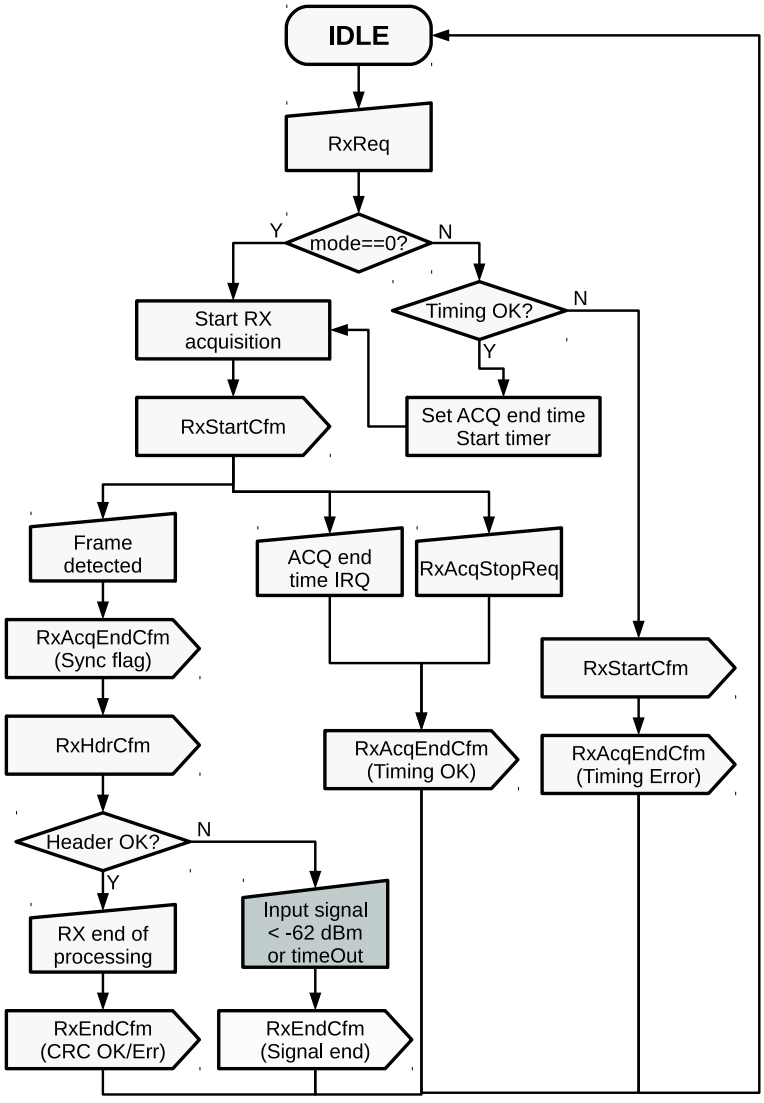


Fig. 10: RX processing control flow

4.3.1 Control flow

The RX processing control flow is depicted in figure 10. The IP core first evaluates the timer mode of the incoming *RxReq* message , i.e., whether the request is timed (timer mode = 1 or 2) or not (timer mode = 0).

In the first case, the timing is evaluated. If the end time is less than 5μs in the future, the system returns immediately with an *RxStartCfm* and an *RxAcqEndCfm* message, the latter indicating a timing error. Otherwise, the end time is set, the timer IRQ is enabled, and the RX acquisition is started. For an untimed *RxReq* message, the acquisition is started immediately.

If no frame is detected, the acquisition is stopped by the timer IRQ (timer mode 1 or 2) or if the host sends an *RxAcqStopReq* (timer mode 0).

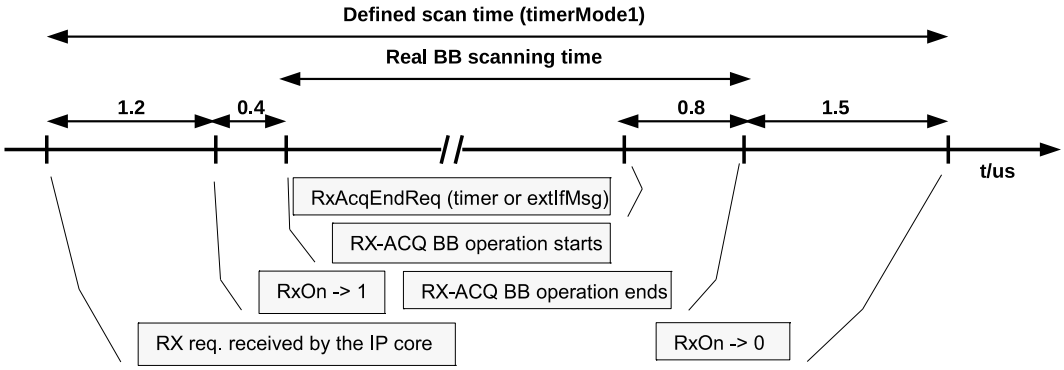


Fig. 11: RX timing for mode=1 (search for given time), no frame detected

In case a frame is detected, the core first returns an *RxAcqEndCfm* (sync indication) immediately after detection, and an *RxHdrCfm* containing all relevant frame data after header decoding. Correct header demodulation is checked via the header parity bit and Viterbi decoding metrics.

If the header has been received correctly, the frame is processed normally and the *RxEndCfm* is sent when the CRC result is available.

If the header was corrupted, the IP core can wait for the input signal to drop below -62 dBm before returning. The threshold is checked once per microsecond. This feature is required by the IEEE standard. However, because it is not necessary in all situations, it is switched off by default and must be enabled explicitly via message *CfgCcaReq*, cf. section 6.3.13. The final *RxEndCfm* is sent in any case with a special *header failed* flag instead of the CRC result.

The RX state machine is handled by function *msgRxRun.m*, which itself consists of *msgRxRun1.m* / *msgRxRun2.m* for start of the RX and reception handling, respectively. Functions *msgRxRun1|2.m* call the actual message handlers *msgRxReq.m*, *msgRxStartCfm.m*, *msgRxAcqEndCfm.m*, *msgRxHdrCfm.m*, and *msgRxEndCfm.m*.

4.3.2 Timing

If the RX acquisition is supposed to search for a given time (timer mode = 1) and if no frame is detected, the corresponding timing is shown in figure 11. About 4μs are needed internally to setup the acquisition and stop it at the end. This time is subtracted from the defined search time. Therefore, the core does not really search for the time defined in the *RxReq* message, but rather returns after the time given, counted from the moment it has received the message.

Similar, if the end time is specified (timer mode = 2), the system returns at the time specified in the *RxReq* message, and stops internal operation a bit earlier.

The end timing in case a frame is detected is shown in figure 12. The processing delay is varies in the range 5-10μs depending on frame length and modulation/coding scheme. Processing delay is defined as the time needed from the moment the last sample of a frame enters the RX-BB buffer until the *RxEndCfm* including the CRC result is sent to the host. About 1μs later, the *rxOn* signal returns to 0.

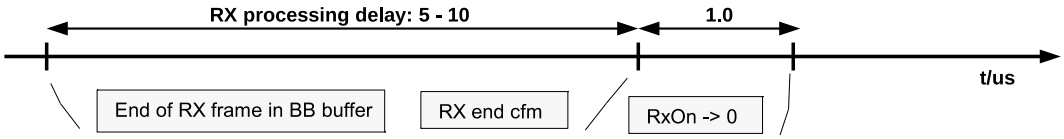


Fig. 12: RX end timing after frame detection

4.4 TX/RX configuration overview

4.4.1 AGC

AGC configuration is done via messages *CfgAgcReq* and *CfgAgcTblReq*. Normal operation needs to select AGC mode 2 with an initial attenuation of 0dB in *CfgAgcReq*, the AGC attenuation and pin table settings are described in detail in section 5.3. The possibility to use a fixed attenuation value (AGC mode 1) should be used only for testing.

4.4.2 Time tracking

The *CfgTtReq* message is used to set some parameters needed by the internal tracking algorithms. They are depending on the carrier frequency f_c. The calculations defined in section 6.3.14 must be executed in floating point. Alternatively, if floating point calculation is not available on the host, the values can be taken from a precalculated table containing the data for all possibly used carrier frequencies.

4.4.3 TX baseband output level

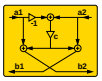
The TX-BB output level can be set by message *CfgTxScalingReq*, see section 6.3.15. The default value of 10dB backoff is optimum in terms of EVM. Saturation occurs for higher values. Lower values do not necessarily decrease the EVM as long as the DAC resolution is sufficient.

4.4.4 TX start timing

Message *CfgTxTimingReq* can be used to delay the pin activation for *paOn* and *txOn*, and/or to delay the baseband DAC output if required by the RF. Details are described in section 4.2 and figure 9. In most cases, the default behaviour which enables the pins and starting the BB DAC output as fast as possible should be sufficient.

4.4.5 TX data source selection

TX data can be created internally using *CfgTxDataSrcReq* or taken from input *dataTx*, see section 6.3.17. The default setting is to use data from input *dataTx*. If the internal data generation is selected, the message must be sent again before each TX request, and the number of bytes of the *TxDataSrcReq* and the *Tx(Imm)Req* message must be equal.



4.4.6 Band and ADC selection

The ADCs and DACs operate at 80 MS/s and are able to transmit a complex valued signal at -20/0/+20 MHz. Band reversal is possible as well. The behavior is selected by the first two parameters of the *CfgBandSelReq* message.

If properly filtered on the RF side, only a single DAC/ADC is needed at the BB. Any input at *adcIm* will be ignored if single ADC operation is selected by the third parameter of *CfgBandSelReq*, see section 6.3.18.

4.4.7 System clock and timer

The internal timer counts at the 802.11a baseband sample rate of 20 MS/s, therefore timer prescaling is set to $1/4$ by default, corresponding to the default system clock of 80 MHz. Reception and transmission of frames can be based on the 32 bit timer.

In case a higher system clock frequency is used, the timer prescaling must be set accordingly through message *CfgTimerPrescaleReq*, cf. section 6.3.23. As parameters p (numerator) and q (denominator) of message *CfgTimerPrescaleReq* are 32 bit each, the most simple is to set $p = 20\,000\,000$, while setting $q =$ system clock in Hz.

The current timer value can be read via message *GetTimeReq*, see section 6.3.24.

4.5 IQ sample debug buffering

4.5.1 Overview and configuration

For debugging purposes, IQ samples can be buffered using message *IqBufModeReq*, see section 6.3.27. Instead of writing IQ samples directly from TX baseband to the TX-DAC outputs, or from the RX-ADC inputs to the RX baseband, samples are stored intermediately in a debug buffer. The buffered data can then be used the feed the TX-DAC or the RX baseband later.

It is also possible to write/read the IQ sample buffer via messages *IqBufWriteReq* / *IqBufReadReq*, cf. section 6.3.28 and section 6.3.29, respectively. This way, arbitrary frames can be sent via TX-DACs and/or fed into the RX baseband. Captured frames can be analyzed in Matlab or similar tools.

Most buffer modes have a *NumSmpl* parameter that may limit using the complete buffer space. In addition, when capturing RX frames, data buffering starts only if the input exceeds a certain threshold, this way providing a simple detection mechanism.

The buffer size is determined by parameters *WlanIqBufNumSmpl_C* and *WlanIqBufAdrWidth_C* in file *./src/common/instances/def_Const_wlan.v*. The address width must be set to $\log_2$(number of samples). Both parameters are set to small values by default in order to save FPGA resources when debug buffering is not needed.

4.5.2 IQ buffer modes

Mode 0 – Reset:
Mode 0 resets the debug buffering feature and switches the IP core back to normal operation. It is necessary to use mode 0 each time after one of modes 1..5 have been used, before reusing a mode or switching to another one. Resetting does not delete the buffer contents.

Mode 1 – RX-RF $\rightarrow$ Buffer:
Mode 1 allows to capture RX frames from ADC. Capturing starts when the absolute value of the real/imag part

of an RX input sample exceeds the threshold set by message *IqBufModeReq*. The threshold is an unsigned 11 bit integer value.

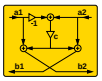
Mode 2 – TX-BB → Buffer:
Mode 2 captures a frame from the TX baseband. The buffer must be setup first, then the TX request message can be sent.

Mode 3 – Buffer → RX-BB:
Mode 3 writes the buffer contents to the RX baseband. The buffer must be setup first, then the RX request message can be sent.

Mode 4/5 – Buffer → TX-RF:
Mode 4 starts a single write of the buffer contents to the TX DACs. In mode 5, the buffer contents is given out repeatedly.

Mode 6 – Check RX trigger:
Different from the other modes, mode 6 does not initiate a buffer transfer, but checks if a trigger event has occured in mode 1 (RX-RF → Buffer). The status is returned in the 2nd confirm parameter, see in *IqBufModeCfm* in section 6.3.56.

Possible system crash:
Beware that the system might crash if the debug mode is not used properly. E.g., if you try to capture a TX baseband frame, but the number of samples is insufficient to buffer the whole frame, then the TX request might not return. Likewise, if you feed a valid RX frame header into the RX baseband from the buffer, but the frame as a whole is insufficient in length, then the RX request might not return. Note that both, TX and RX, need some extra samples at the end for proper operation.



5 RF interface

5.1 Prerequisites

5.1.1 AGC response time

The response to AGC line changes must be less then than 0.5 μs. After that time, the input level must not vary anymore by more than 1 dB.

5.1.2 Oscillator stability

Any instability of the carrier frequency after switching from RX to TX, especially while transmitting the short and long symbols preamble, can influence the ability of the another receiver to estimate the frequency offset correctly. Similar, the carrier frequency must be stable while the RF attenuation lines are switched during AGC operation in order to guarantee proper frequency estimation during reception.

The short term carrier frequency deviation due to transients (e.g. switching of RX/TX/PA and / or AGC control signals) must be less than 10 kHz during short symbols, and less than 2.5 kHz during long symbols.

Note that this has nothing do with the range of the carrier frequency offset correction capability. Offset correction is possible up to ± 1 subcarrier. Time tracking works up to ± 50 ppm.

5.1.3 Baseband and RF oscillator coupling

The carrier frequency and the baseband clock must be derived from the same source. Otherwise, time tracking does not work properly and the reception of long frames might be corrupted. Time tracking parameters are derived from the accumulated frequency/phase at the receiver and therefore dependent on the actual carrier frequency, cf. section 6.3.14. The IP core supports carrier frequencies between 20 MHz and 20 GHz.

5.2 RF pin connections

5.2.1 Rx/Tx/PaOn

The signals *rxOn*, *txOn*, and *paOn* are intended to activate the RF-RX path, RF-TX path, and the power amplifier, respectively. All signals are active high. See section 4.2 and section 4.3 for the exact pin timings in relation to frame transmission and reception.

5.2.2 Attenuation

Up to 16 parallel *attn* signals are available to control the RF attenuation. The pin settings can be configured via message *CfgAgcTblReq*, see section 5.3 for details.

The *attn_or* signal goes high for the first cycle after a change of the *attn* outputs. This feature can be used to initiate a parallel to serial conversion or similar if required by the RF. If the RF is connected to the parallel *attn* outputs directly, registering is not needed, as the signals are registered internally and keep their value until the next change occurs. The *attn_or* output can be left unconnected in this case.

5.2.3 Three wire bus

Programming the RF is possible via the 3-wire serial bus interface using output signals *twb(Dat|Clk|En)*, see also figure 1. Pin control follows the typical scheme, applying data to *twbDat*, sending a short pulse at *twbClk* while the data is valid, and finishing the transmission with a short pulse at the *twbEn* pin, see figure 13 for details. The clock frequency is about 4 MHz with pulses being 37.5 ns wide. Three wire serial bus operation is started by message *CfgTwbReq*, cf. section 6.3.20.

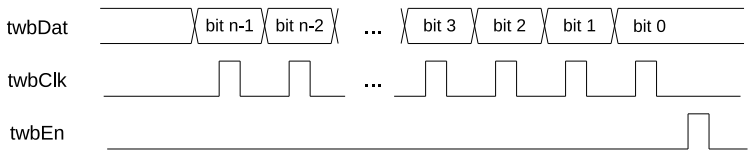


Fig. 13: Three wire serial bus control

Note that 3-WB control is firmware based and relatively slow. Hence, it cannot be used for AGC or other time-critical control issues.

5.3 AGC calibration

5.3.1 Overview

The internal AGC table consists of 70 entries, ideally corresponding to attenuation settings from 0 dB (highest RF gain) to 69 dB (lowest RF gain), LUT[0..69]. All 70 LUT entries must be set independent of the number of settings the real RF provides. Each of the LUT entries uses the best matching real RF gain setting. If the RF provides gain settings in steps that are > 1 dB, then some settings will be assigned to multiple LUT entries. If the RF has very fine grained gain settings, then some of them will not be used.

RF gain table calibration is a 2-step procedure

- First, the RF gain setting mapped to LUT[0] must be found
- Second, the remaining RF gain steps are mapped to LUT[1..69]

5.3.2 Calibration of attenuation step 0

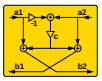
The RF gain corresponding to LUT[0], i.e., the highest gain to be set by the IP core, must be found first. This is not necessarily the highest gain the RF provides.

Instead, one of the gains the RF provides needs to be selected such that the baseband noise input level at this gain reaches about -30 dB FS (full scale, possibly a bit higher). This corresponds to an input amplitude RMS of about $10^{-30/20} \cdot 2^9 = 16.2$ for the 10 bit / 20 MS/s baseband input data.

Remark: Although the external ADC interface has 12 bit, internal operation after RX filtering works with 10 bit only at the 20 MS/s BB input. Due to the unknown noise bandwidth and the digital RX filters, it is not possible to define the target level directly at the ADC inputs.

In order to find the RF gain setting for the desired input level, one needs to step through all possible RF gains (starting with the highest) and check the BB input level. The level can be obtained by lowering the acquisition threshold, see section 6.3.19, so that noise is "detected", and then running the normal *RxReq*, which reports the level in the confirm message, see section 6.3.35. In detail:

- Message *CfgAgcTblReq* is used to fill the LUT with pin combinations for all possible gain settings of the RF chip. Assignment is arbitrary, but in the easiest case one would assign LUT[0] = highest gain, LUT[1] = second highest gain, etc ... The only thing important is to be able to set the RF gain manually for testing using AGC mode 1 with *CfgAgcReq*. The attenuation values required by message *CfgAgcTblReq* can all be set to 0, they are not needed here.



- The RX acquisition detection threshold must be lowered via message *CfgAcqThrReq*, see section 6.3.19. This yields almost immediate "detection" of WGN input.
- The system must be set to the highest gain using message *CfgAgcReq* for a fixed attenuation, i.e., setting AGC mode 1, starting with LUT[0].
- RX request message must be sent continuously, and the BB amplitude returned with the *RxHdrCfm* messages must be monitored. If the average level is below 16, the HW setup should be changed to provide a higher gain. Otherwise, the tests are continued with lower RF gains until the measured BB input level is just a bit over 15.

The RF gain setting found by the above procedure is used as highest gain (LUT[0]) addressed by the IP core. Ideally, it should also be (one of) the highest possible gains the RF provides in order to achieve a good noise figure.

In terms of performance, it is no problem to select another RF gain for LUT[0] with BB input noise level > 15 as long as the noise level drives the ADCs reasonably. Reception works without degradation, however, the dynamic range of the AGC is more limited at the upper end. In other words, if dynamic the range of the AGC is not an issue, the accuracy of the procedure is not critical, one can just use any roughly suitable RF gain for LUT[0].

An example implementation of the above procedure is implemented in *./tst/m/test/setupWlanAgc.m*. Function *setupWlanAgc* is executed during the setup with the virtual RF core, see description in section 2.6 and the reference therein.

5.3.3 AGC table configuration

The RF gain setting found by the procedure in section 5.3.2 has 0 dB attenuation by definition and is placed at LUT[0]. RF settings with higher gain are not used. All RF settings with lower gain are defined relative to LUT[0]. They must be mapped as close as possible to the remaining attenuation steps 1-69 of the AGC table.

E.g., assume the RF hardware has further gain steps that are [2.3, 4.7, 7.4, 9.9, 12.0, ...] dB lower. The mapping to the LUT then looks as shown in table 1.

Attn (ideal)	0	1	2	3	4	5	6	7	8	9	10	11	...
Actual RF attn	0.0	0.0	2.3	2.3	4.7	4.7	4.7	7.4	7.4	9.9	9.9	12.0	...
Attn table entry	0	0	2	2	5	5	5	7	7	10	10	12	...

Tbl. 1: RF attenuation mapping to baseband AGC table

The actual RF gain is first mapped to the closest hypothetical attenuation, e.g., 0.0 dB is mapped to 0 and 1, 2.3 dB is mapped to 2 and 3, 4.7 dB is mapped to 4, 5, and 6, and so on. The attenuation values are then rounded to the next integer. These values, together with the corresponding pin settings, are used as table entries for message *CfgAgcTblReq*, see section 6.3.9.

5.4 DC offset correction

5.4.1 Overview

Baseband DC offset correction is possible for RX and TX. In both cases, the correction values are 12 bit complex and simply added to the baseband values directly before feeding the DAC or receiving data from the ADC, respectively.

In addition to the configuration of correction values (cf. section 5.4.2 and section 5.4.3), DC offset correction must be switched on explicitly using message *CfgDcOffCorrReq*, see section 6.3.10.

5.4.2 TX

TX DC offset correction uses a single correction value which is set via message *CfgDcOffCorrTxReq*, see section 6.3.11.

5.4.3 RX

RX DC offset correction is LUT based and depends on the current AGC settings. Similar to the AGC configuration, RX DC offset correction features 70 lookup values, one for each gain setting. The values are configured via *CfgDcOffCorrRxReq*, see section 6.3.12. LUT values are automatically selected when a certain gain is applied, no matter if the AGC is enabled, or if gain is selected manually.

Calibration of the RX DC offset correction values is possible by manual gain selection in combination with data IQ buffer reception, cf. section 4.5, mode 1.

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

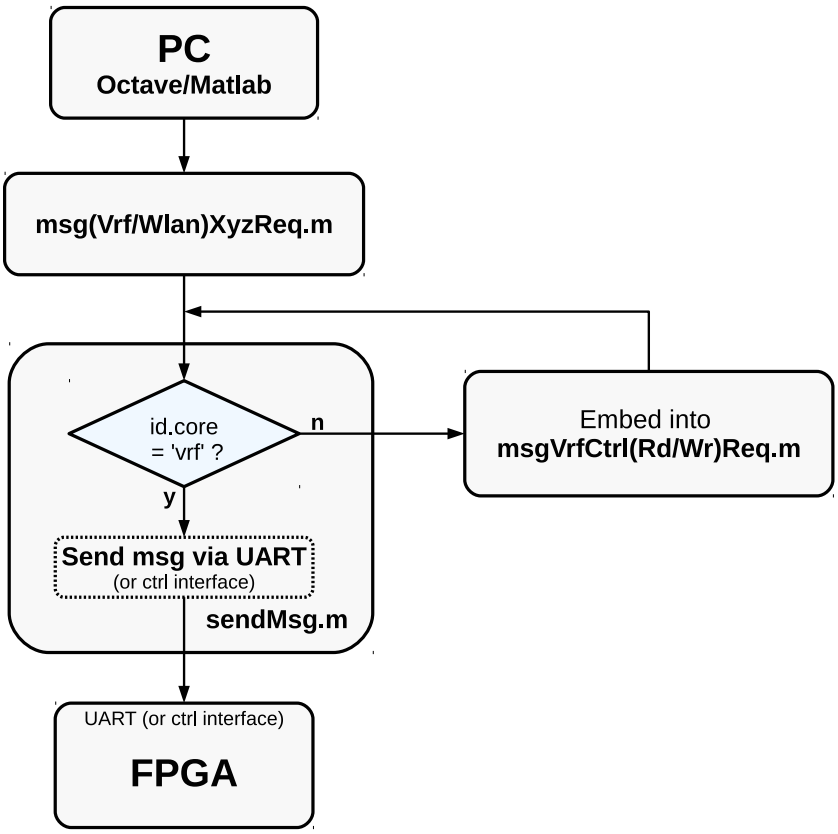


Fig. 14: Message handling and forwarding

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 4 there are 2 WLAN IP cores with instance numbers 0 for TX and 1 for RX, respectively.

The *vrfX* example in figure 4 has 3 different targets, *core* = 'vrf' / *inst* = 0, *core* = 'wlan' / *inst* = 0, and *core* = 'wlan' / *inst* = 1. Figure 14 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 4.

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.

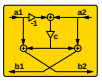
6.3.3 MsgId 25 – TxImmAReq

Purpose
Start TX-11a immediately request

Parameters

1. TX mode
   0-7 $\rightarrow$ 11a: 6..54 MBit/s
2. TX length in bytes, 1-4095

Description
TxImmAReq starts a TX immediately after the message ID is received without waiting for the mode and length parameters. The host must provide mode and length within 2μs after the message ID, otherwise the system will crash. Therefore, *TxImmAReq* must not be used with the UART control interface. See section 4.2 for details on timing and control flow.



6.3.4 MsgId 26 – TxImmBReq

Purpose
Start TX-11b immediately request

Parameters

1. TX mode
   1-7 → 11b: 1l, 2s, 2l, 5.5s, 5.5l, 11s, 11l
2. TX length in bytes, 1-4095

Description
TxImmBReq starts a TX immediately after the message ID is received without waiting for the mode and length parameters. The host must provide mode and length within 0.2 μs after the message ID, otherwise the system will crash. Therefore, *TxImmBReq* must not be used with the UART control interface. See section 4.2 for details on timing and control flow.

6.3.5 MsgId 27 – TxReq

Purpose
Start TX based on timer request

Parameters

1. TX mode
   0-7, bit15=0 → 11a: 6..54 MBit/s
   1-7, bit15=1 → 11b: 1l, 2s, 2l, 5.5s, 5.5l, 11s, 11l
2. TX length in bytes, 1-4095
3. Timer mode:
   0: start as fast as possible
   1: start after given delay
   2: start at given time
4. Lower 16 bit of timing
5. Upper 16 bit of timing

Description
TxReq starts a timer based TX. Different from *TxImmAReq*, it starts the transmission only after all parameters are received completely by the IP core.

In timer mode 0, the TX sequence starts after about 1 μs. Timing parameters 4+5 are ignored in this case.

In timer mode 1, the TX starts after the given number of timer ticks, while in timer mode 2, the system waits until the given time is reached before starting TX. The earliest start time is 30 timer ticks in the future, counted from the moment the message is sent to the IP core. If the given start time is less than 30 ticks in the future or if the delay is less than 30, the system returns with a timing error instead of starting TX, see section 6.3.31 and 4.2 for details on timing and TX state machine.

One timer tick is equivalent to the BB sample rate of 20 MS/s.

6.3.6 MsgId 28 – RxReq

Purpose
Start RX acquisition request

Parameters

1. Timer mode:
   0: search until explicitly stopped by *RxAcqStopReq*
   1: search for given time
   2: search until given time is reached
2. Lower 16 bit of the timing
3. Upper 16 bit of the timing

Description
RxReq starts the RX acquisition. In any case, the acquisition ends when an 802.11a frame is received. Depending on the timer mode, it may also stop without receiving a frame.

In timer mode 0, the acquisition can be stopped by sending an *RxAcqStopReq*. The *RxAcqStopReq* is ignored, if a frame has been detected before the request is received the system.

In timer mode 1, the acquisition runs for the given time, while in timer mode 2, the acquisition runs until the given time is reached. The system then returns automatically to idle mode if a frame has not been received. The minimum useful time to run the acquisition is about 5 μs. If used with less, the system returns immediately without searching for signals.

The *RxAcqStopReq* must not be used in timer mode 1 or 2 due to possible IRQ conflicts. Details of the state machine and timing are given in section 4.3.

One timer tick is equivalent to the BB sample rate of 20 MS/s.

6.3.7 MsgId 29 – RxAcqStopReq

Purpose
Stop RX acquisition request

Parameters
None

Description
RxAcqStopReq stops the RX acquisition. The RX-ACQ should have been started before in timer mode 0. The request is ignored if the system is not in acquisition mode anymore, e.g. because a frame has been detected.

6.3.8 MsgId 30 – CfgAgcReq

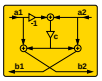
Purpose
AGC configuration request

Parameters

1. Run mode, 1 = fixed gain setting, 2 = normal AGC operation
2. Fixed (mode 1) / initial (mode 2) attenuation step, 0-69

Description
CfgAgcReq switches between a fixed gain setting (mode=1) and normal AGC operation (mode=2). The fixed / intial attenuation parameter selects the fixed (mode=1), or initial (mode=2) attenuation step, where one step typically corresponds to 1 dB.

Mode 1 is typically used for testing. In mode 2, the initial attenuation must be set to 0 if the AGC table has been configured properly, cf. section 5.3 and 6.3.9. After reset, AGC is switched off completely using the entry LUT[0].



6.3.9 MsgId 31 – CfgAgcTblReq

Purpose
AGC table configuration request

Parameters

1. Attenuation of the step 0
2. AGC pin output of step 0
3. Steps 1..69 to follow ...

Description
CfgAgcTblReq configures the AGC attenuation table and the AGC pin settings as required by the RF. This message has 140 parameters, one pin setting and one AGC attenuation for each AGC step.

The attenuation values corresponds to real attenuation steps of the RF, see section 5.3 for details. The pin settings are mapped directly to the 16 attenuation output lines.

6.3.10 MsgId 32 – CfgDcOffCorrReq

Purpose
DC offset correction configuration request

Parameters

1. 0/1 = DC offset correction off/on

Description
CfgDcOffCorrReq switched the DC offset compensation off or on. The correction is off by default.

6.3.11 MsgId 33 – CfgDcOffCorrTxReq

Purpose
TX DC offset value configuration request

Parameters

1. DC offset correction value (real part)
2. DC offset correction value (imag part)

Description
CfgDcOffCorrTxReq configures the TX DC offset. Different from RX DC offset compensation, there is only a single value.

6.3.12 MsgId 34 – CfgDcOffCorrRxReq

Purpose
RX DC offset table configuration request

Parameters

1. DC offset correction for attenuation step 0 (real part)
2. DC offset correction for attenuation step 0 (imag part)
3. Steps 1..69 to follow ...

Description
CfgDcOffCorrRxReq configures the AGC dependent RX DC offset table. This message has 140 parameters, real and imag part of the DC offset correction value for each AGC attenuation step. The values are 12 bit signed.

6.3.13 MsgId 35 – CfgCcaReq

Purpose
Clear channel assessment configuration request

Parameters

1. Timeout in microseconds
2. Offset calibration in dB [-15..+15]

Description
CfgCcaReq configures the way the IP core waits for a clear channel after a failure during header decoding. The timeout in microseconds determines how long the system waits for the end of a signal. It should be set a bit longer than the maximum expected frame length. Timeout = 0 is used to switch off waiting completely.

According to the 802.11a standard, the system needs to check the signal being less than -62 dBm before returning. In order to determine the corresponding threshold, the IP core assumes the RX input level being -94 dBm at gain step 0 and a baseband amplitude of 16. This is roughly correct for a noise figure of 7 dB and RF calibration according to section 5.3. In case the RF differs significantly from the assumption, the offset calibration parameter can be used to shift the threshold by up to ± 15 dB.

The feature is switched off by default, i.e. the IP core returns directly after a header error.

6.3.14 MsgId 36 – CfgTtReq

Purpose
Time tracking configuration request

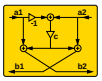
Parameters

1. Time tracking factor mantissa
2. Time tracking factor shift
3. Time tracking jump low word
4. Time tracking jump high word

Description
CfgTtReq sets the time tracking parameters. The parameters are used to derive time tracking correction phases from the accumulated phase shift due to frequency offset. Therefore, depending on the carrier frequency f_c, they are calculated as:

$$\begin{aligned} tt_{Exp} &= \lfloor \log_2(f_{sc}/f_c) \rfloor + 18 \\ tt_{Shift} &= 2^{tt_{Exp}} \\ tt_{Mant} &= \left\lfloor f_{sc}/f_c/2^{tt_{Exp}-32} \right\rfloor \\ tt_{Jump} &= \mathrm{round}(2^{16} \cdot f_c/(64 f_{sc})) \end{aligned} \tag{2}$$

where $20\,\mathrm{MHz} \leq f_c \leq 20\,\mathrm{GHz}$ is the carrier frequency and $f_{sc} = 312.5\,\mathrm{kHz}$ is the subcarrier spacing. Parameter tt_{Exp} is only needed for intermediate calculation, the actual shift is executed via multiplication on the controller. Parameter tt_{Jump} is 32 bit and must be split into low/high 16 bit. Time tracking can be switched off by setting all parameters to 0 (default).



6.3.15 MsgId 37 – CfgTxScalingReq

Purpose
TX scaling configuration request

Parameters

1. TX scaling factor, 0..32767

Description
CfgTxScalingReq sets a linear TX baseband scaling factor between 0 and 32767. The DAC backoff with respect to a full scale signal (RMS of the complex signal = 2047) is calculated as $backOffDb = -20\log_{10}(scaleTx/25693)$. The default value is 8192 corresponding to 10 dB DAC backoff. A backoff below 10 dB (i.e., $scaleTx > 8192$) yields signal saturation/distortion and hence performance loss at higher order modulation schemes.

6.3.16 MsgId 38 – CfgTxTimingReq

Purpose
TX timing configuration request

Parameters

1. Delay to switch on PA pin
2. Delay to switch on TX pin
3. Delay to start BB processing

Description
CfgTxTimingReq allows configuration of TX start timing sequence, i.e., to modify the timing relation between switching on the *txOn / paOn* pins and the start of the TX-BB DAC output. The delay is given in BB samples of 20 MHz. All values must be in the range 1-65535, the default is 1. See section 4.2 for details. If the clock frequency differs from the DAC/ADC sample frequency, message *CfgTimerPrescaleReq*, section 6.3.23 must be used in addition to ensure timing is working correctly.

6.3.17 MsgId 39 – CfgTxDataSrcReq

Purpose
TX data source configuration request

Parameters

1. Source selection, 0=external, 1=internal
2. Number of bytes of internal data source (1..4095)
3. Start value for internal data source (0..255)

Description
CfgTxDataSrcReq switches between internally generated data (for testing) and externally supplied input data (default, normal operation). When generated internally, the 8-bit values count up the defined number of bytes from the given start value. The last 4 bytes are always 0 in order to create a correct CRC-32. The message must be sent before each TX request in this case.

6.3.18 MsgId 40 – CfgBandSelReq

Purpose
Band selection / reversal and ADC configuration request

Parameters
1. 3/0/1 → IF = -20/0/+20 MHz
2. 0/1 → Band reversal off/on
3. 0/1 → Dual/Single ADC use

Description
CfgBandSelReq configures the band selection and ADC usage. The DACs and ADCs are always running at 80 MHz, so in principle, a single ADC/DAC is sufficient for low-IF operation. Both ADCs/DACs can be used, however, in order to reduce filter requirements on the RF side. If single ADC is selected, the *adcIm* input is ignored. Zero IF, band reversal off, and dual ADC use are selected by default.

6.3.19 MsgId 41 – CfgAcqThrReq

Purpose
Acquisition threshold configuration request

Parameters
1. ThresholdIdx = -1/0 → low/normal threshold

Description
CfgAcqThrReq selects one of several predefined acquistion thresholds by setting the *ThresholdIdx*.
-1 → Increased sensitivity, for AGC calibration only (see section 5.3.3), many false alarms
0 → Normal sensitivity (default)

6.3.20 MsgId 42 – CfgTwbReq

Purpose
Three wire serial bus configuration request

Parameters
1. DatLo
2. DatHi
3. NumBits

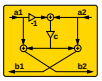
Description
CfgTwbReq writes the lower *NumBits* bits of the combined data word [*DatHi DatLo*] to the RF via the 3-wire serial interface. Details are explained in section 5.2.3.

6.3.21 MsgId 43 – 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. All GPOs are 0 by default.



6.3.22 MsgId 44 – 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 values are set to 0.

6.3.23 MsgId 45 – CfgTimerPrescaleReq

Purpose
Configure timer prescale factor

Parameters

1. Prescale numerator p, lower 16 bit
2. Prescale numerator p, upper 16 bit
3. Prescale denominator q, lower 16 bit
4. Prescale denominator q, upper 16 bit

Description
CfgTimerPrescaleReq sets the timer prescaler. The internal timer runs with frequency $p/q \cdot FpgaClk$. In order to receive the correct timing information, the prescale factors need to be set such that the timer counts the BB frequency (20 MHz), i.e., setting $p = BbClk$, $q = FpgaClk$ or simplified versions thereof. The default after reset is $p = 1$, $q = 4$ corresponding to the minimum FPGA system clock of 80 MHz. *CfgTimerPrescaleReq* stops, resets, and restarts the timer.

6.3.24 MsgId 46 – GetTimeReq

Purpose
Get system time request

Parameters
None

Description
GetTimeReq reads the current value of the 32 bit timer.

6.3.25 MsgId 47 – VersionReq

Purpose
Version number read request

Parameters
None

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

6.3.26 MsgId 48 – 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.

6.3.27 MsgId 49 – IqBufModeReq

Purpose
Set IQ sample buffer mode

Parameters
1. 0 = Reset/Off (normal operation)
   1 = RX-RF → Buffer
   2 = TX-BB → Buffer
   3 = Buffer → RX-BB
   4 = Buffer → TX-RF
   5 = Like 4, but continuous cyclic output until switched off
   6 = Check trigger state after RX-RF → Buffer
2. Number of samples to write/read
3. Detection threshold for mode 1, range 0..2047

Description
IqBufModeReq configures the IQ sample buffer operation. This message is only for debug purposes and not needed during normal operation.

6.3.28 MsgId 50 – IqBufWriteReq

Purpose
Write data to IQ sample buffer

Parameters
1. Number of samples to write
2. x(0), lower 16 bit
3. x(0), upper 8 bit
4. x(1), lower 16 bit
5. ...

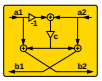
Description
IqBufWriteReq writes data to the IQ sample buffer. The data is 12 bit complex signed, with [xIm(3:0) xRe(11:0)] in the lower word, and xIm(11:4) in the upper word.

6.3.29 MsgId 51 – IqBufReadReq

Purpose
Read data from IQ sample buffer

Parameters
1. Number of samples to read

Description
IqBufReadReq initiates reading data from the IQ sample buffer.



6.3.30 MsgId 63 – BootCfm

Purpose
Confirm system 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.

6.3.31 MsgId 64 – TxStartCfm

Purpose
Confirm TX request

Parameters

1. Status: 0=OK, 1=Timing error

Description
TxStartCfm confirms the *TxReq* or *TxImmAReq*. The status parameters indicates a timing error in case of *TxReq* with mode=1/2 if the delay was to small (mode=1) or the start time was in the past (mode=2). See section 4.2 for details on timing and state machine.

6.3.32 MsgId 65 – TxEndCfm

Purpose
Confirm TX end

Parameters
None

Description
TxStartCfm confirms the end of a TX. See section 4.2 for details on timing and state machine.

6.3.33 MsgId 66 – RxStartCfm

Purpose
Confirm RX acquisition start request

Parameters
None

Description
RxStartCfm confirms the start of the RX acquisition.

6.3.34 MsgId 67 – RxAcqEndCfm

Purpose
Confirm RX acquisition end

Parameters

1. Status: 0=Stopped, 1=RX setup timing error, 2=Synchronization start

Description
RxAcqEndCfm confirms the end of the RX acquisition without frame detection, or the synchronization of a frame. The message is triggered either by an *RxAcqStopReq*, by the internal timer, or if a frame is detected. A timing error indicates that the system returned without starting the acquisition. See section 4.3 for details on timing and state machine.

6.3.35 MsgId 68 – RxHdrCfm

Purpose
Confirm header decoding

Parameters

1. 0=sync+header OK, 1=header error
2. RX mode, 0-7 $\rightarrow$ 6-54 MBit/s
3. RX length in bytes
4. Frequency offset, $f_{sc} <=> 2^{13}$
5. RX-BB amplitude (0..256)
6. AGC final attenuation setting
7. Frame start time low
8. Frame start time high

Description
RxHdrCfm confirms the 802.11a header decoding and returns miscellaneous physical layer parameters such as RX mode and length, frequency offset, RX baseband amplitude (measured in the input buffer), the final AGC setting, and the frame start time.

The frame start is given in terms of the internal timer value, see section 4.4.7. It indicates the moment when the first sample was written into the baseband input buffer.

A header error indicates that something went wrong during header decoding. Therefore, the values for RX mode and length, frequency offset, and frame start are possibly incorrect.

6.3.36 MsgId 69 – RxEndCfm

Purpose
Confirm PSDU decoding end

Parameters

1. 0=CRC OK, 1=CRC Error, 2=header error

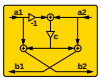
Description
RxEndCfm confirms the end of the RX reception and indicates the CRC result. The CRC is only correctly indicated if the last 4 bytes of the payload data are zero. In case of a header error, the message is sent after waiting for the end of the noisy signal.

6.3.37 MsgId 70 – CfgAgcCfm

Purpose
Confirm AGC table configuration request

Parameters
None

Description
CfgAgcCfm confirms execution of *CfgAgcReq*.



6.3.38 MsgId 71 – CfgAgcTblCfm

Purpose
Confirm AGC table configuration request

Parameters
None

Description
CfgAgcTblCfm confirms execution of *CfgAgcTblReq*.

6.3.39 MsgId 72 – CfgDcOffCorrCfm

Purpose
Confirm DC offset configuration request

Parameters
None

Description
CfgDcOffCorrCfm confirms execution of *CfgDcOffCorrReq*.

6.3.40 MsgId 73 – CfgDcOffCorrTxCfm

Purpose
Confirm TX DC offset table configuration request

Parameters
None

Description
CfgDcOffCorrTxCfm confirms execution of *CfgDcOffCorrTxReq*.

6.3.41 MsgId 74 – CfgDcOffCorrRxCfm

Purpose
Confirm RX DC offset table configuration request

Parameters
None

Description
CfgDcOffCorrRxCfm confirms execution of *CfgDcOffCorrRxReq*.

6.3.42 MsgId 75 – CfgCcaCfm

Purpose
Confirm clear channel assessment configuration request

Parameters
None

Description
CfgCcaCfm confirms execution of *CfgCcaReq*.

6.3.43 MsgId 76 – CfgTtCfm

Purpose
Confirm time tracking configuration request

Parameters
None

Description
CfgTtCfm confirms execution of *CfgTtReq*.

6.3.44 MsgId 77 – CfgTxScalingCfm

Purpose
Confirm TX scaling configuration request

Parameters
None

Description
CfgTxScalingCfm confirms execution of *CfgTxScalingReq*.

6.3.45 MsgId 78 – CfgTxTimingCfm

Purpose
Confirm TX timing configuration request

Parameters
None

Description
CfgTxTimingCfm confirms execution of *CfgTxTimingReq*.

6.3.46 MsgId 79 – CfgTxDataSrcCfm

Purpose
Confirm TX data source configuration request

Parameters
None

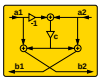
Description
CfgTxDataSrcCfm confirms execution of *CfgTxDataSrcReq*.

6.3.47 MsgId 80 – CfgBandSelCfm

Purpose
Confirm band selection and reversal configuration request

Parameters
None

Description
CfgBandSelCfm confirms execution of *CfgBandSelReq*.



6.3.48 MsgId 81 – CfgAcqThrCfm

Purpose
Acquisition threshold configuration confirm

Parameters
None

Description
CfgAcqThrCfm confirms execution of *CfgAcqThrReq*.

6.3.49 MsgId 82 – CfgTwbCfm

Purpose
Three wire bus configuration confirm

Parameters
None

Description
CfgTwbCfm confirms execution of *CfgTwbReq*.

6.3.50 MsgId 83 – CfgGpoCfm

Purpose
General purpose output configuration confirm

Parameters

1. Updated GPO value

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

6.3.51 MsgId 84 – 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.

6.3.52 MsgId 85 – CfgTimerPrescaleCfm

Purpose
Confirm timer prescale setting

Parameters
None

Description
CfgTimerPrescaleCfm confirms execution of *CfgTimerPrescaleReq*.

6.3.53 MsgId 86 – GetTimeCfm

Purpose
Confirm get timing request and return the global 32 bit timer

Parameters

1. Lower 16 bit of global 32 bit timer
2. Upper 16 bit of global 32 bit timer

Description
GetTimeCfm returns the current internal timer value in response to a *GetTimeReq*. See section 4.4.7 for details on the timer.

6.3.54 MsgId 87 – 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 88 – LedBlinkCfm

Purpose
Confirm LED blink request

Parameters
None

Description
LedBlinkCfm confirms execution of *LedBlinkReq*.

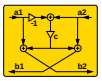
6.3.56 MsgId 89 – IqBufModeCfm

Purpose
Confirm setting IQ buffer mode

Parameters

1. RX trigger flag, 0 = not triggered, 1 = triggered

Description
IqBufModeCfm confirms the execution of IQ buffer mode setting. The RX detection flag only applies to mode 6, when the request checks if RX capturing has been triggered.



6.3.57 MsgId 90 – IqBufWriteCfm

Purpose
Confirm writing data to the IQ sample buffer

Parameters
None

Description
IqBufWriteCfm confirms writing data to the IQ sample buffer.

6.3.58 MsgId 91 – IqBufReadCfm

Purpose
Confirm reading data from the IQ sample buffer

Parameters

1. x(0), lower 16 bit
2. x(0), upper 8 bit
3. x(1), lower 16 bit
4. ...

Description
IqBufReadCfm confirms reading data from the IQ sample buffer. The data is 12 bit complex signed, with [xIm(3:0) xRe(11:0)] in the lower word, and xIm(11:4) in the upper word.

7 Test benches

7.1 Prerequisites

Simulation has been tested under Linux using the free Icarus Verilog simulator, see *http://iverilog.icarus.com*. Postprocessing extracts lines from files using *sed*, and file comparison is done using the *diff* command. These tools should be installed with any standard Linux distribution or Cygwin.

7.2 Top level test benches

7.2.1 Overview

Top level test benches can be found in the *./reg/top* subdirectory. A test bench simulates messages via the *ctrl* interface and provides inputs / collects outputs for either TX or RX frames. All steps like reset, booting, configuration, and TX/RX frames are covered.

7.2.2 Test scope and concept

In order to stimulate the WLAN IP core, its interfaces are connected to standard buffers containing the data written to inputs or read from outputs. For the ADC input / DAC output, additional IO-rate setting is provided to allow emulation of the IP core running at higher clock frequency than the ADC/DAC sample rate. Due to the half duplex nature of the IP core, buffers are shared for data byte input/output, and for ADC/DAC sample input/output. The setup overview is shown in figure 15. The UART and the X16 controller are not used during Verilog simulation, see section 7.4.

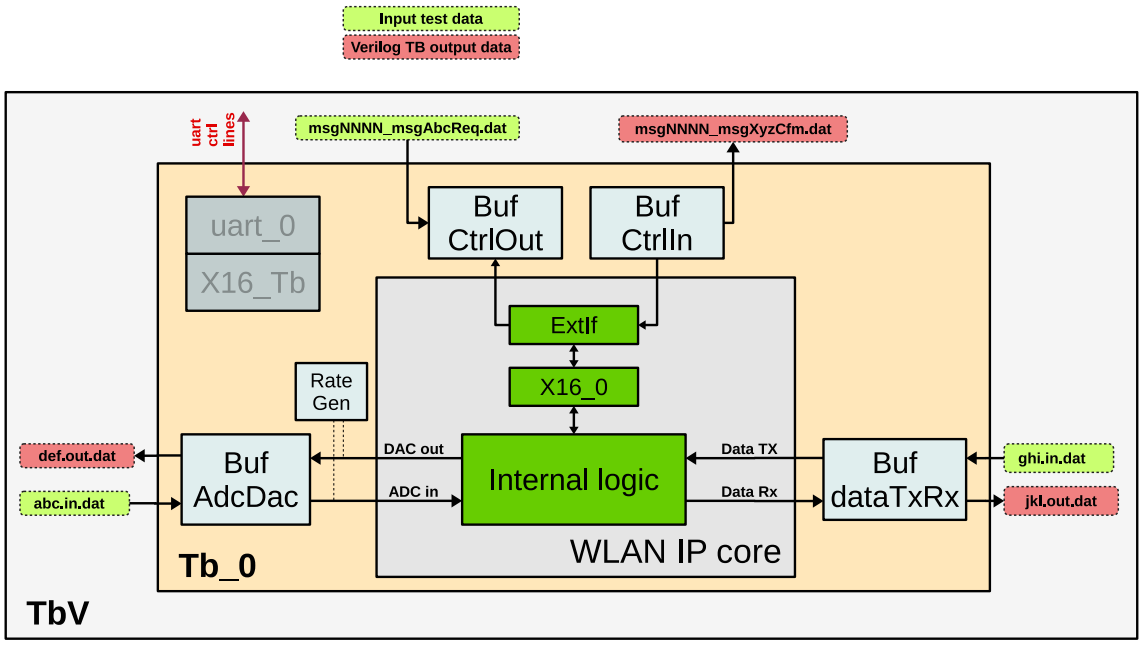


Fig. 15: Top test bench overview

7.2.3 Operation

During Verilog simulation, the *AdcDac* and *DataTxRx* buffer memories are initialized by the outer test bench wrapper *TbV* from files *abc|ghi.in.dat*. At the end of the simulation, output stub memories are read and dumped

to files *def|jkl.out.dat*. Request messages are written to / confirm message are read from the IP core via buffers *CtrlIn* / *CtrlOut*, respectively. Corresponding files are *msgMMMM_msgXyzReq/Cfm.dat*. All data files are located in the *./dat* subdirectory.

There are no references for the output files. Instead, for most tests, a Matlab/Octave script *tbNNNN_eval.m* provides a partial evaluation or visualization of the output. In most cases it is necessary to inspect the traces in *./out/TbV.vcd* to verify the results.

7.2.4 Invocation

In order to run a Verilog simulation, change to directory *./reg/top/tbNNNN* and run *./runTbv*.

7.3 Module level test benches

7.3.1 Overview

Module test benches can be found in the *./reg/mod* subdirectory. The test bench concept is signal processing oriented, test benches for other modules, e.g. *irqCtrl*, *uart16*, etc... are not provided. In some cases (modules *mod11a*, *tx11a*, and *rx11a*) there is a separate test bench used in order to include the FFT otherwise shared between RX and TX.

7.3.2 Test scope and concept

The module under test (here: *xyz*) is connected to stubs writing to and reading from inputs and outputs, see figure 16 for details. These stubs are included in *Tb_0*. Additional control is provided in *TbV.v*. The separation splits the test bench into an inner part that can be synthesized (*Tb_0*), and an outer part which cannot (*TbV*). *TbV* can be replaced to run the test bench in real time on the FPGA board, see section 7.4.

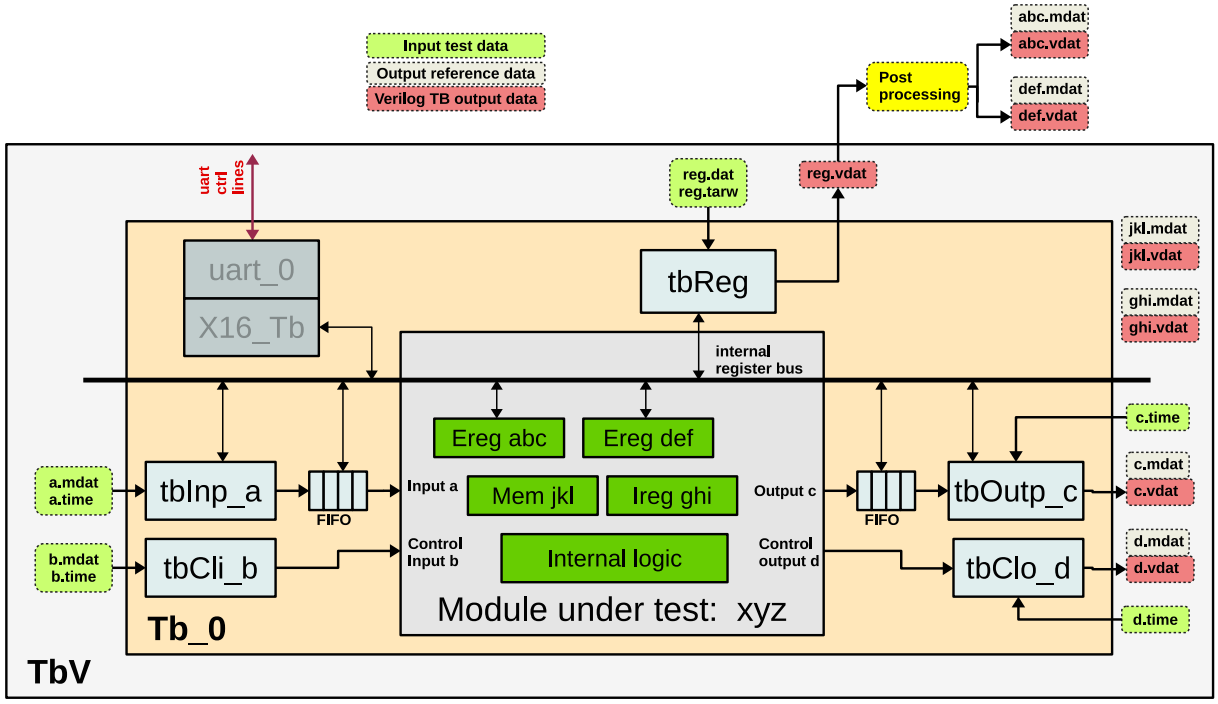


Fig. 16: Test bench overview

Each module consists of a number of standard interfaces and components plus additional module specific internal logic. Inputs and outputs can be either for data transfer (e.g. Input a, Output c) with full handshaking and possible FIFO connection to the module under test, or control lines (e.g. Control Input b, Control Output d) in figure 16. A dedicated test stub is connected to each type of interface, *tbInp*, *tbCli*, *tbOutp*, and *tbClo*. The appendix of the stub instance matches the interface name. Although figure 16 shows exactly one interface of each type, the number is arbitrary and varies from module to module.

Furthermore, the stub *tbReg* is connected to the module under test, reading from and writing to the internal bus in order access externally accessible control registers (Eregs *abc*, *def*), and possibly write and read messages (not shown in figure 16). A module can further contain internal state registers (Iregs) and memories that may or may not be tested depending on the test bench configuration.

Files *a.mdat*, *a.time*, *b.mdat*, ... etc define the data and the clock cycle when the data is written from the stub to module *xyz* inputs. Likewise, stub *tbReg* is configured by files *reg.dat* and *reg.tarw*, defining data, time, register address, and if data is read from or written to the internal bus. At the outputs, files *c.time* and *d.time* define the clock cycles when data is read by stubs*tbOut* and *tbClo*.

The simulation collects output data in stubs *tbOut* , *tbClo*, and *tbReg* for later evaluation.

7.3.3 Operation

During Verilog simulation, the stub memories are initialized by the outer test bench wrapper *TbV*. At the end of the simulation, output stub memories are read and dumped into files *c.vdat*, *d.vdat*, and *reg.vdat*, respectively. These files are located in the *out* subdirectory. For each *.vdat* file, there is a corresponding *.mdat* file containing the reference data.

File *reg.vdat* is postprocessed, i.e. the lines containing read data from the internal register bus are extracted and dumped into separate files for different eregs (here *abc.vdat* and *def.vdat*) and possibly messages (not shown in figure 16).

Internal states and memories can be dumped directly by the Verilog wrapper, although this is used in a few test cases only.

7.3.4 Invocation

In order to run a Verilog simulation, change to directory *./reg/mod/modules/xyz/tbNNNN* and run *./runTbv*. In order to run all test benches available, change to directory *./reg/mod* and run *./runVtbAll*.

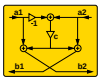
7.3.5 Dumping a VCD file

VCD file dumping is disabled by default. It can be enabled by editing *./reg/mod/modules/xyz/tbNNNN/tbSetup/TbV.v* and enabling the $dumpfile / $dumpVars commands.

7.4 FPGA testing

By replacing the outer wrapper *TbV*, all test benches can be converted to a synthesizable version. The test benches include an X16 controller and a UART which are unused during Verilog testing. For FPGA real time testing, UART lines are connected to FPGA pins and from there to a PC, similar to the setup in figure 2. Memories of debug stubs are read and written via the UART and the X16 controller. Inputs/outputs, control inputs/outputs, ereg values and messages can be tested this way, but no internal states or memory contents.

All modules test benches and most of the top level test benches have been run in real time on the FPGA directly for extended verification. However, this possibility is not made available in this package.



8 Frequently asked questions

8.1 Applications

I need a proprietary communication system similar to 802.11a, can you provide this based on your IP core?
That's one of the ideas behind this product. Wireless LAN is million dollar consumer business. There is no plan to get into that. But if you need something a bit special, describe your application and we'll see how to adapt the IP core.

I would like to build my private WLAN just for fun, can I get a full featured version for free?
Yes, I'd like to see fun projects using this core and I would be happy to support you. Just send me an email.

Will there ever be an 802.11b transceiver, making the 802.11g complete?
Probably yes.

Will there ever be an 802.11n transceiver?
Probably not.

8.2 Matlab/Octave/C references

Are there any reference simulations in Matlab/C or similar?
There are references implemented in Octave 4.0.2. They should be Matlab compatible although that has not been tested. There is no C-code behind.

Do you plan to make references available?
No. There is quite some tooling and scripting behind that would require significant documentation effort before other people could use it.

8.3 Verilog implementation + synthesis

Why are there so many unused wires, especially at the module interfaces?
Headers etc. are autogenerated and inputs/outputs are using standard interfaces similar to the one described in section 3.3.2. In many cases, not all interface handshake signals are used, but there is no need to remove them manually. Synthesis will do this for you.

Why is this old fashioned Verilog header style used?
The project was once started like that and because module headers and the corresponding instance definitions are autogenerated, it doesn't really matter.

What is this strange X16 module?
X16 is a small home grown controller used to configure the HW modules and to handle the messages.

Why are there so many warnings during compilation?
You can safely ignore all of them. Below there is a list of typical warnings from Altera-Quartus, for Xilinx-ISE/Vivado it's similar.

"...truncated value with size 31/32 to match size of target (N)"
This is when a decimal value (integer by default) is assigned to a register or wire. Many constants etc... are autogenerated from Matlab/Octave definitions and don't have a bit width definition.

"... object "xyz" assigned a value but never read"
All interfaces between modules are autogenerated. Depending on the application, some of the handshake lines might not be used, so they are left unconnected.

"... Net 'abc' at xyz has no driver or initial value, using a default initial value '0'"
Same as before, typically these are unused interface handshake lines.

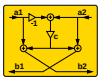
"... case item expression never matches the case expression"
This is a warning bug occurring when indexed parts of 2-dimensional register arrays are used in the case statement. The synthesis does work.

"... Converted the fan-out from the tri-state buffer ... into an OR gate"
Well, FPGAs seem to prefer OR gates, no need to worry.

8.4 RF interface

Can I connect an RF with a serial interface for AGC etc... ?
Yes, but you have to be aware of the strict timing requirements of 802.11a or use a proprietary MAC protocol.

Do you support any other interfaces?
The interface is kept as generic as possible. If you need any parallel-to-serial conversion or similar, you have to add it yourself.



9 License

9.1 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.

9.2 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.

9.3 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 compile 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.

Use of additional test code, Matlab/Octave scripts etc... is only permitted to evaluate the IP core.

9.4 Non-private use

Non-private use includes any application of this code related to your job or other professional activity, commercial, non-commercial, academic, military or whatever.

Non-private users are allowed to evaluate this code. Evaluation comprises all steps necessary to assess its usability for your project such as

- inspecting the sources
- simulation of the code
- synthesizing the code for FPGA or similar
- embedding the code into a larger system
- bring-up of a test system including RF and MAC

Before you start using the code for your project purpose, you must obtain a commercial license.

9.5 Private use

You are allowed to use this code free of charge for private purposes.

References

[1] IEEE Computer Society
IEEE Std 802.11^{TM}-2007
IEEE, 3 Park Avenue, New York, NY 10016-5997, USA