The CBSR communication module on-board the satellite is designed to minimize the volume and power consumption, while maximizing its performance at the same time. Both the transmitter (on-board the satellite) and the receiver (in the ground station) are implemented using Software Defined Radio technique. However, in this document only the electrical and mechanical design of the satellite communication module is discussed.

Contents

List of acronyms

BB	Base-band
CBSR	C-band Satellite Radio
DAC	Digital to Analog Converter
DMA	Direct Memory Access
FSM	Finite State Machine
NCBR	Narodowe Centrum Badań I Rozwoju
PA	Power Amplifier
PN	Pseudorandom Noise
PUT	Poznań University of Technology
SDR	Software Defined Radio
SR	SatRevolution S.A.
SoM	System-on-Module
SoC	System-on-Chip
t.b.d	to be determined

1. Module Characteristics

1.1. General Parameters

The CBSR transmitter module has been designed to fit on-board the Cubesat nanosatellite developed by SatRevolution S.A. The mechanical construction and the electrical parameters of the module have been agreed by SR and PUT within a joint R&D project financed by NCBR.

Based on the system specification presented in [1], the electrical parameters of the module are the following:

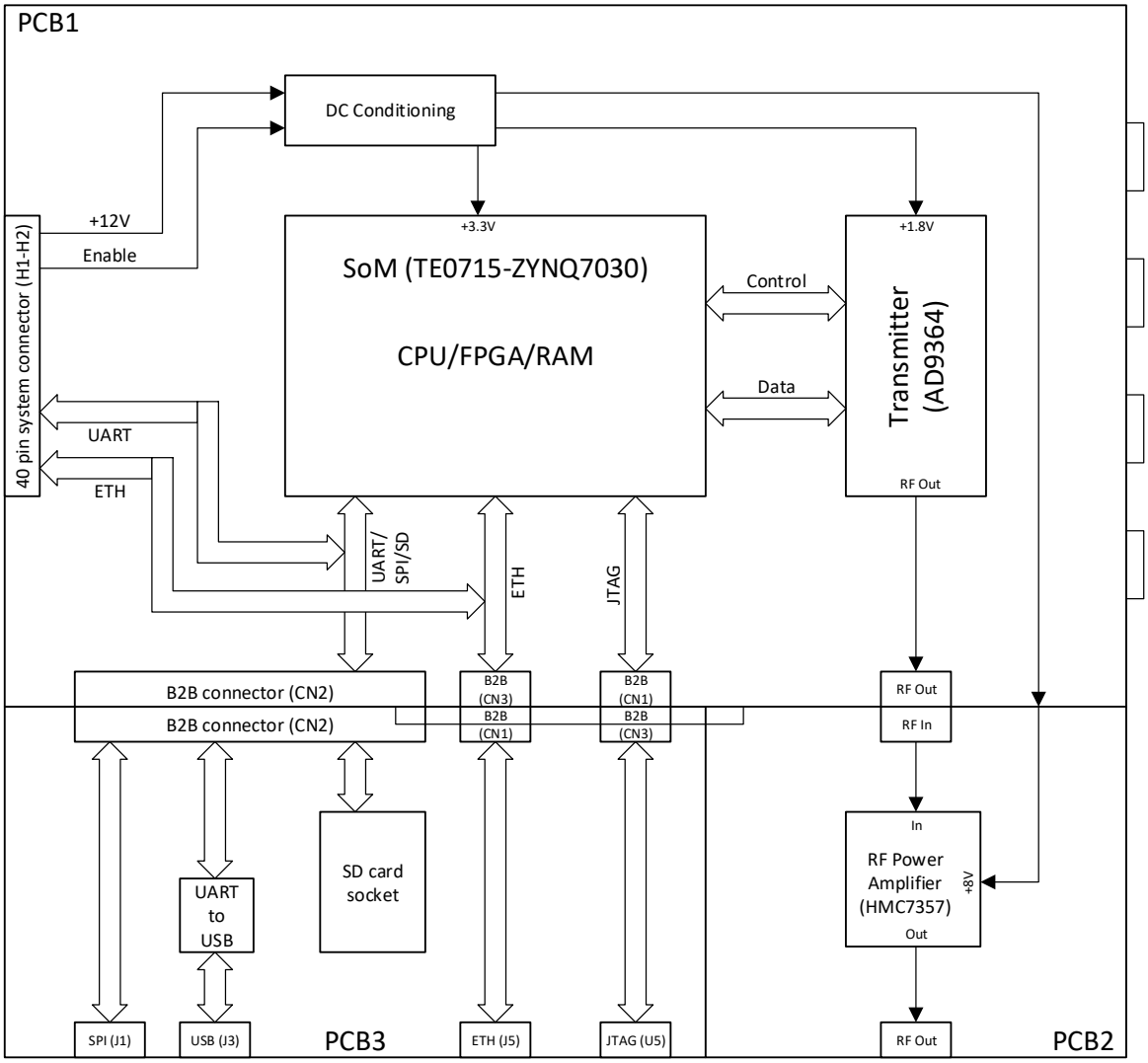
- carrier frequency – user selectable between 5500 MHz and 6000 MHz, default 5840 MHz
- carrier frequency step – 2.4 Hz
- frequency stability – < 1ppm, -40°C ÷ +85°C
- channel bandwidth – user selectable: 1 MHz, 1.25 MHz, 5 MHz, 10 MHz, 20 MHz
- transmit power – 2W (+33dBm) max. @50 Ω, user selectable in 0.25dB steps
- modulation type - digital (quadrature) - OQPSK
- channel coding – Turbo, user selectable code rate: 0.19 – 0.91
- power supply – 12 V, 2.5 A max.
- power consumption:
    - off mode – t.b.d. (100uA?) (BB-OFF, PA-OFF)
    - sleep mode – t.b.d. (BB-ON – sleep mode, PA-OFF)
    - idle mode – t.b.d. (1A?) (BB-ON, PA-OFF)
    - transmit mode – t.b.d. (2.2A?) (BB-ON, PA-ON)
- off mode to idle mode time – approx. 5 s
- data/control interface – RS-232, Ethernet

1.2. Implementation and Block Diagram

The transmitter has been implemented using SDR technique. Most of the its functionality (physical layer) is implemented in FPGA, only the control block and layers above the physical layer are implemented in software. The transmitter module runs its own lightweight, Linux based, operating system.

The transmitter functionality can be modified by software upgrades only – no hardware modifications are required. This can be done even in space (during the mission) providing that the uplink can be used for uploading the bitstream to the satellite.

The block diagram of the transmitter module is presented in Fig. 1.1. Base-band signal processing and system management sections (see 2.1), low-power analog (RF) section (see 2.2) as well as power supply section (see 2.3) are located on the main PCB (PCB1). The RF power amplifier (see 2.3) is located on a separate PCB (PCB2), however the two PCBs form a single module covered with an aluminum radiator. For debugging and testing purposes the module can be connected in the lab to PCB3 hosting the auxiliary section (see 2.4), using zero insertion force ribbon connectors.



Debugging only!

Fig. 1.1 Block diagram of the transmitter module

2. Module Electrical Design

2.1. Base-band Signal Processing and System Management Sections (a.k.a Digital Section)

An efficient SDR implementation of the transmitter requires application of an FPGA device. For this purpose a Xilinx ZYNQ System-on-Chip device, which combines dual-core ARM Cortex-A9 processor with the FPGA and a choice of interfaces was selected [2]. Most of the BB processing blocks are implemented in the FPGA, while the ARM processor is used for system control and management functions.

The design of a system based on ZYNQ SoC from scratch is time consuming and error-prone, since the device requires many external components, e.g. Flash memory, SDRAM, interface drivers, clock and power supplies. Therefore an off-the-shelf, ready-to-use System-on-Module from Trenz Electronic GmbH is used in the transmitter module.

The key features of the TE0715-30-1I3 SoM [3] are listed below:

- Industrial-grade Xilinx Zynq SoC XC7Z030
- Rugged for shock and high vibration
- 10/100/1000 Mbps Ethernet transceiver PHY
- MAC address EEPROM
- 32-bit wide 1GB DDR3 SDRAM
- 32 MByte quad SPI Flash memory
- Programmable clock generator
- Transceiver clock (default 125 MHz)
- Plug-on module with 2 × 100-pin and 1 × 60-pin high-speed hermaphroditic strips
- 132 FPGA I/Os (65 LVDS pairs possible) and 14 PS MIO available on B2B connectors
- 4 GTP/GTX (high-performance transceiver) lanes
- GTP/GTX (high-performance transceiver) clock input
- USB 2.0 high-speed ULPI transceiver
- On-board high-efficiency DC-DC converters (1.0 V, 1.5 V, 1.8 V power rails)
- System management CPLD
- Temperature compensated RTC (real-time clock)
- User LED
- Evenly-spread supply pins for good signal integrity
- 50x40 mm module size

The TE0715 block diagram is depicted in Fig. 2.1.

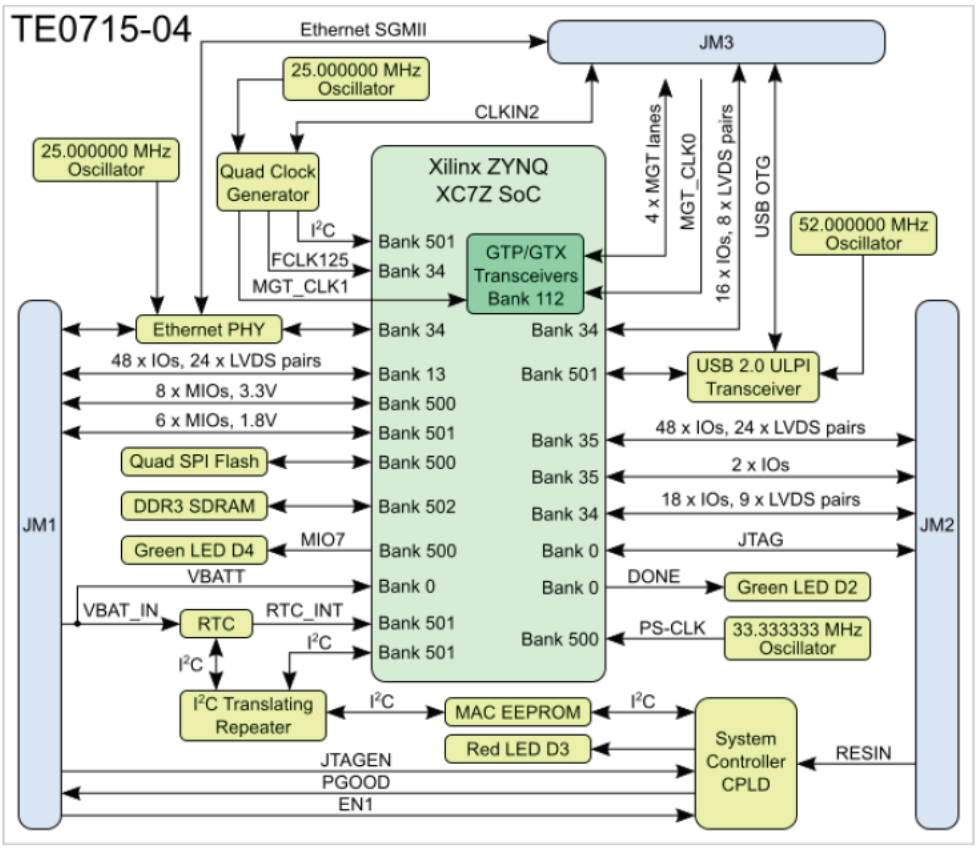


Fig. 2.1 TE0715 block diagram

The module (daughter-board) uses 2x100 pins and 1x60 pins board-to-board connectors to access the following signals from the mother-board:

- Bank 14, Bank 34, Bank 35 Zynq GPIOs
- Bank 500, Bank 501 Zynq MIOs
- Bank 112 Zynq MGTs
- JTAG interface
- System Controller I/O pins
- SD card interface
- ETH interface
- USB interface

2.2. RF Section (a.k.a Analog Section)

The RF part of the transmitter is based on the Analog Devices integrated transceiver AD9364. It is a high performance, highly integrated radio frequency (RF) Agile Transceiver™ designed for use e.g. in 3G and 4G base station applications. Its programmability and wideband capability make it ideal for a broad range of transceiver applications [4].

The key features of the AD9364 are listed below:

- RF 1 × 1 transceiver with integrated 12-bit DACs and ADCs

- band: 70 MHz to 6.0 GHz
- supports time division duplex (TDD) and frequency division duplex (FDD) operation
- tunable channel bandwidth (BW): <200 kHz to 56 MHz
- 3-band receiver: 3 differential or 6 single-ended inputs
- superior receiver sensitivity with a noise figure of <2.5 dB
- Rx gain control - real-time monitor and control signals for manual gain Independent automatic gain control
- 2-band differential output transmitter
- highly linear broadband transmitter, Tx EVM: ≤−40 dB Tx, noise: ≤−157 dBm/Hz noise floor, Tx monitor: ≥66 dB dynamic range with 1 dB accuracy
- integrated fractional-N synthesizers: 2.4 Hz maximum local oscillator (LO) step size
- multichip synchronization
- CMOS/LVDS digital interface

The AD9364 block diagram is depicted in Fig. 2.2.

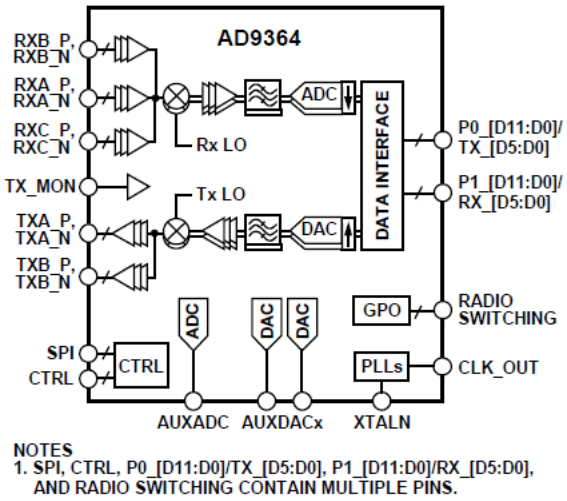


Fig. 2.2 AD9364 block diagram

AD9364 is connected to the SoC via data and control lanes, including SPI interface for transceiver configuration and control. In the CBSR communication module only the transmit path of AD9364 transceiver is used since the radio link is uni-directional (space to ground). However, the module is “reception-ready”, i.e. it can be easily modified to act as a receiver and implement bi-directional link.

The RF signal from AD9364 is amplified by an Analog Devices HMC7357 integrated power amplifier to improve the link power budget [5]. HMC7357LP5GE is a three-stage GaAs pHEMT MMIC 2 watt power amplifier that operates between 5.5 and 8.5 GHz. The amplifier provides 29 dB of gain and +35 dBm of saturated output power at 34% PAE from a +8V supply. With an excellent output IP3 of +41.5 dBm, the HMC7357LP5GE is ideal for linear applications such as high capacity point-to-point and point-to-multi-point radios or VSAT/SATCOM applications demanding +35 dBm of efficient saturated output power.

2.3. Power Supply Section

The CBSR communication module requires high-quality power supplies. The 8 V, 3.3 V, 1.8 V and 1.3 V supplies are derived from a single 12 V power supply on-board the satellite. The power section uses the following devices:

- LM5060 – High-Side Protection Controller With Low Quiescent Current
- TPS54623 – Synchronous Step-Down SWIFT™ Converter With Light Load Efficiency and Hiccup Overcurrent Protection
- ADP1755 – low dropout CMOS linear regulator
- MIC37301 – low-dropout linear voltage regulator
- TLV76750 – Precision Linear Voltage Regulator

Some of the power rails can be controlled independently by the FPGA to turn-off the sections currently not in use. The transmitter module can be also put in the 'off mode' by an external ENABLE control signal. This helps to save the energy which is a scarce resource on-board the satellite.

2.4. Interface Section

The CBSR communication module is interfaced to other satellite sub-systems via two 52-pin PC-104 stack-thru connectors. The following signal are available on the system connector:

- +12V – 3 lines
- GND – 11 lines
- Ethernet interface – 4 lines
- RS-232 interface – 2 lines (module control)
- RS-232 interface – 2 lines (debug)
- ENABLE – 1 line

All control lines uses 3.3V single-ended signaling.

2.5. Auxilary Section

Due to the limited module size some of the peripherals necessary for system development and debugging have been moved to an external PCB which is removed when the module is installed on-board the satellite. The PCB is connected to the main module using three zero-insertion-force ribbon connectors.

The PCB contain the following interfaces/connectors:

- JTAG
- SPI – a "mirror" of the SPI available on the system connector
- CAN
- ETH
- 2 x USB – emulation of ZYNQ SoC UART interfaces
- SD – full size SD card slot
- system reset button

2.6. Schematics

The detailed schematics of all sections can be found in Appendix A.

3. Module Mechanical Design

The CBSR communication module uses 3 multi-layer PCBs, which are stacked together to form a single, easy-to-use module.

The main PCB, which hosts the digital sections, part of the RF section (integrated transceiver) and power section is an 95.89x90.17 mm, 8-layer PCB, including 3 board-to-board connectors for the SoM as well as the PC-104 system connectors. The other part of the RF section (power amplifier) is located on a separate PCB, which can contain a low noise amplifier if a bi-directional version of the CBSR communication module is developed in the future.

The module is covered with a block of aluminum for heat dissipation and EM shielding. The block holds the SMA connectors for attaching an antenna system. Patch-type transmit antenna has been selected which provides 6dBi gain to improve the link budget.

Details of the mechanical construction (including PCBs) can be found in Appendix B as well as in external *step* and *gerber* files.

4. Transmitter implementation – PL part (FPGA)

4.1. Radio frame structure

The transmission is organized into radio frames depicted in Fig. 4.1. In consists of a preamble and tunable number of subframes. We can distinguish three parts in the preamble. The first part is the *G_AMB* used for AGC purposes. The second part is the *T_AMB* used for time synchronization. Finally, the *F_AMB* is used for frequency offset estimation. Each subframe in a frame corresponds to a single codeword coming from a turbo coder. Each subframe consists of n partial codewords (*PCWORD*) and $n+1$ midambles used for phase offset estimation (*P_AMB*). Where n is calculated based on the currently selected coding rate

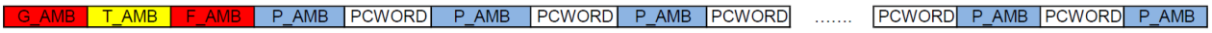


Fig. 4.1 Radio frame structure

4.2. Simulink model

The transmitter implementation in FPGA is prepared with the help of Matlab/Simulink and its HDL Coder. In Fig 4.2 the transmitter block inputs and outputs are presented, the relevant ones are described in Table 4.1.

Fig. 4.2 Transmitter block

Table 4.1 Inputs and outputs of transmitter block

Name	Type	Description
input_words	Input	32-bit words loaded from software via DMA
words_valid	Input	Indication whether data in input_words line is valid
ENABLE_REG	Input/ Output	Register that stores i.a. a value that enables/disables the transmitter. When its state is high, the signal can be transmitted, when it's low the transmitter will stop sending samples its output (only after the internal data buffer has been emptied)
RADIO_CONFIG_REG	Input	A register used to set: • the coding rate of the turbo coder. There are 7 coding rates settings from 0 to 6 (described in later sections) • the transmission mode. There are 4 modes available (values 0-3, described in later sections) • the roll-off factor of the shaping RRC filter. There are 2 settings available i.e. 0 and 1. • the length of the frequency offset estimation part of the preamble. There are 3 possible values from 0 to 2 (described in later sections)
NUM_RF_SUBFRAMES_REG	Input/ Output	A line used to set the number of subframes in each radio frame.
tx_out_I	Output	Transmitter in-phase component samples
tx_out_Q	Output	Transmitter quadrature component samples
valid_out	Output	Indication whether output samples are valid
tx_load_req	Output	A line used to request data from the software. The high output indicates that data can be sent to the block.
SUBFRAME_COUNT_REG	Output	The number of transmitted subframes

In Fig. 4.3 the general structure of the transmitter is presented. Since the scheme is quite complex the main parts of the transmitter are shown and described in detail in the subsequent sections.

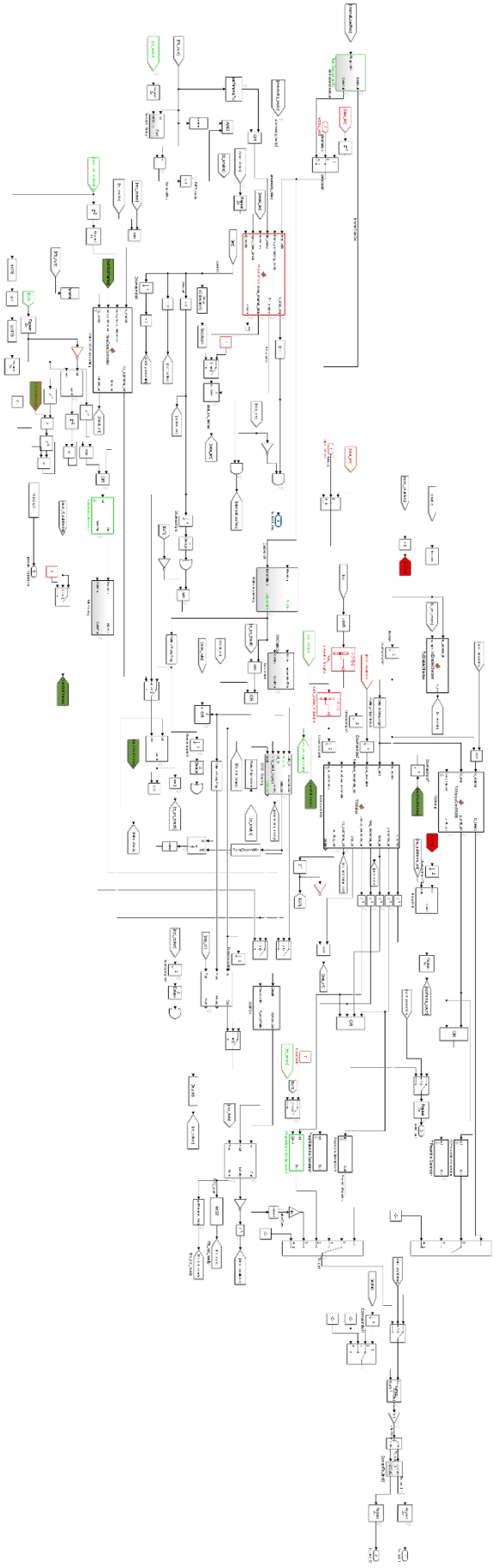


Fig. 4.3 Overview of the transmitter Simulink schematics

4.3. Data loading and fake frame generation

The first part of the transmitter is shown in Fig. 4.4. It is responsible for requesting data from software or the internal data source. It also prepares the data for the subsequent subsystems i.e. it converts received 32-bit words into bits accepted by scrambling and coding subsystems described in subsection 4.2.1.

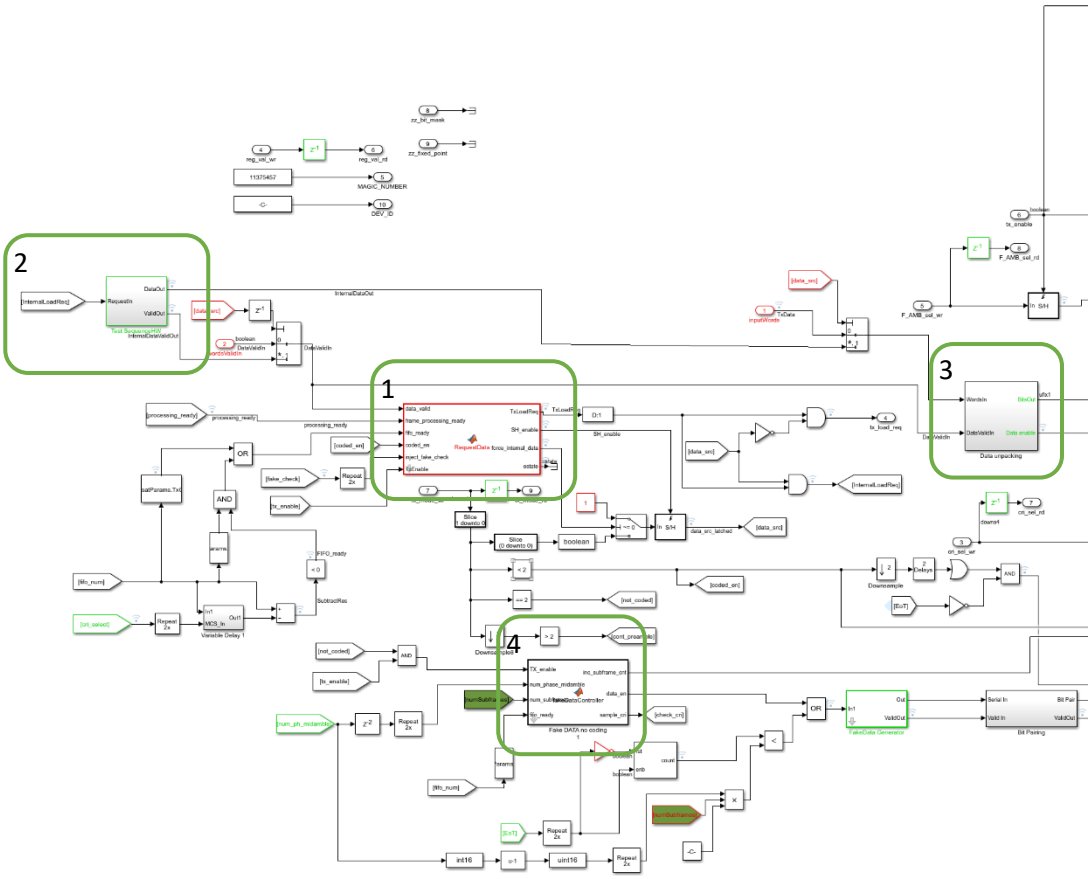


Fig. 4.4 Data loading and fake frame generation part of transmitter schematics

In the proposed transmitter three modes of data transmission are foreseen:

1. Transmission of channel coded data from software
2. Transmission of channel coded data from the internal data generator
3. Transmission of repeatable uncoded data from the internal data generator

The realization of all of the modes is performed with the help of blocks 1 to 4 shown in Fig. 4.4.

Block number 1 (Fig. 4.5) is a Matlab function that implements a Finite State Machine (FSM) responsible for requesting data from software or an internal data source. The block has 6 inputs and 4 outputs which are described in Table 4.2.

Fig. 4.5 Finite state machine for requesting data from software or internal data generator

Table 4.2 Inputs and outputs of *RequestData* block

Name	Type	Description
data_valid	Input	Indication whether there is valid data
frame_processing_ready	Input	Indication whether the channel coder is ready to accept new data
fifo_ready	Input	Indication whether data symbols buffer is ready to accept new data
coded_en	Input	Indication whether coded data transmission mode is selected
inject_fake_check	Input	Flag used to trigger the use of fake (internally generated) data until new data from software arrives
tx_enable	Input	Flag indicating whether the transmission is still enabled
tx_load_req	Output	Signal responsible for requesting new data, either from software or internal data generator. The data is requested until we receive the required number of words i.e. 187
SH_enable	Output	Signal used to hold current data source selection
force_internal_data	Output	Signal used to override the software data source in case there is no valid data coming from the software
ostate	Output	Indication of the current state of the FSM (used for debugging)

Block number 2 is the internal data generator and its structure is depicted in Fig. 4.6 it uses a PN sequence generator with the following polynomial $x^{20} + x^{17} + 1$. The data output of the generator is tailored to fit the 32-bit words coming from the software. Each time there is a high *RequestIn* signal a new word is generated that can be fed to the block responsible for unpacking words to bits (block number 3). Block number 3, depending on transmission mode selection, can accept either data coming from software or internal data as described above. The structure of the data unpacking block is depicted in Fig. 4.7 It uses a simple state machine responsible for reading 32-bit words from the FIFO queue in cycles of 32 samples. The word is then converted to a stream of bits that can be fed to the next blocks in the processing chain.

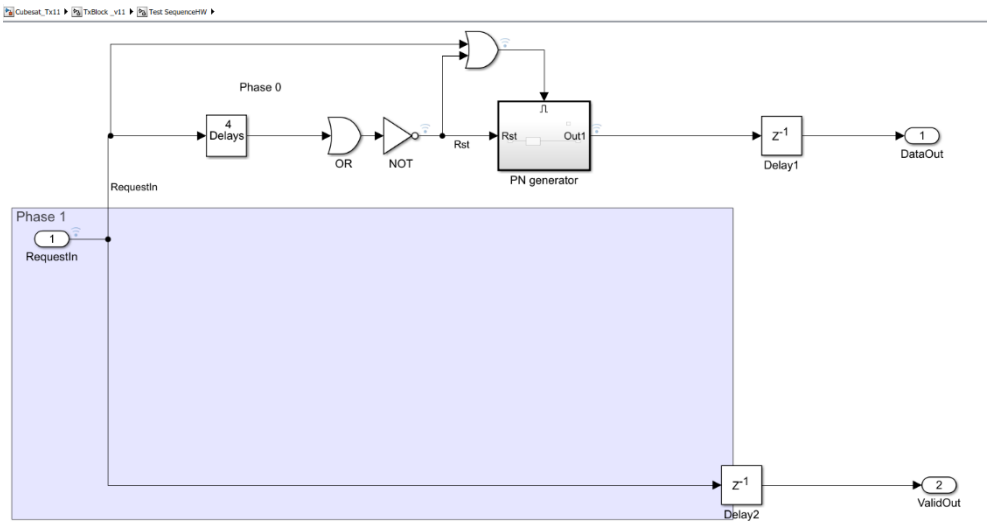


Fig. 4.6 Internal data generator

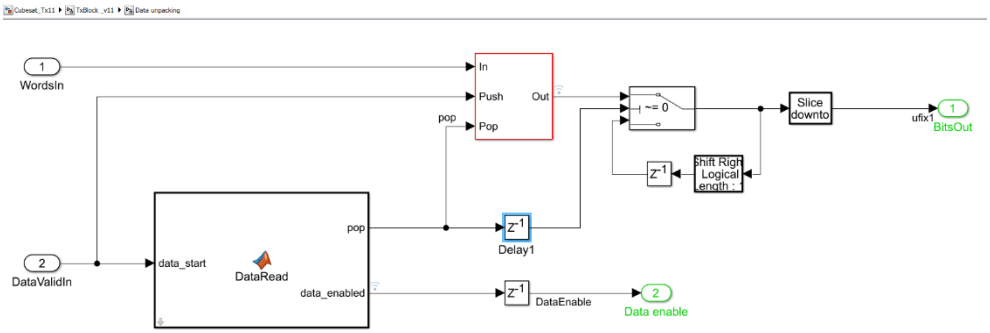


Fig. 4.7 Data unpacking subsystem

The final block (number 4, Fig. 4.8) is a controller used to generate bits for the uncoded transmission mode. It has 4 inputs and 3 outputs which are described in Table 4.3

Fig. 4.8 Controller for handling fake uncoded data

Table 4.3 Inputs and outputs of fakeDataController block

Name	Type	Description
tx_enable	Input	Flag indicating whether the transmission is still enabled
num_phase_midamble	Input	Number of midambles used for phase offset estimation used in each subframe
num_subframes	Input	Number of subframes in radio frame
fifo_ready	Input	Indication whether data symbols buffer is ready to accept new data
inc_subframe_cnt	Output	Signal used to increment a subframe counter of the transmitter
data_en	Output	Signal used for enabling reading fake uncoded bits from a lookup table
sample_cri	Output	Signal used to sample the state of the *cri_sel* input of the transmitter in uncoded mode of transmission

The *fakeDataController* and *RequestData* blocks can be operational only when the *TxEnable* signal is high. The *tx_enable* signal coming to the transmitter block can be controlled from software that doesn't know the internal state of the transmitter. If the software changed the state of the *tx_enable* line when the transmitter is still processing data it could corrupt the data and the structure of the frames and subframes for the next transmission session. To avoid this problem a simple function was created which block is depicted in Fig. 4.9. Its task is to sample the state of the *tx_enable* of the transmitter only when the subframe counter outputs the value 0, meaning this is the beginning of the radio frame.

Fig. 4.9 Controller of the *tx_enable* state used for requesting data

4.3.1. Data scrambling, CRC insertion and channel coding

The second part of the transmitter (blocks 1 and 2 shown in Fig. 4.10) is responsible for preparing the data for the OQPSK modulator firstly by scrambling the incoming bits and secondly by adding CRC and channel coding it with a Turbo coder.

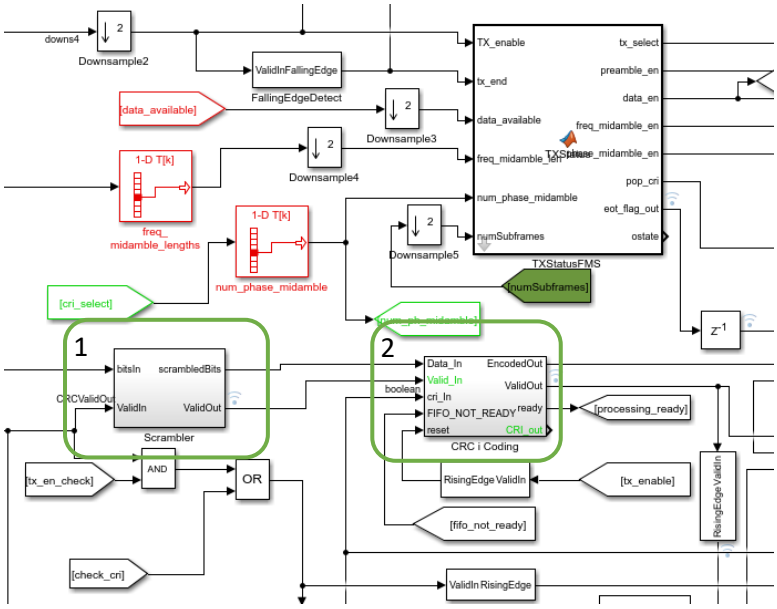


Fig. 4.10 Data scrambling and coding part of transmitter schematics

The scrambling block (denoted by 1 in Fig 4.10) is using a PN sequence generator to modify the data bits fed to it by performing an XOR operation as shown in Fig. 4.11. The PN sequence generator is using the same polynomial as the generator used in internal data source (i.e. $x^{20} + x^{17} + 1$) but they are independent of each other. The difference here is that we use only a single bit from the generated sequence instead of 32 bits in the case of the internal data source generator.

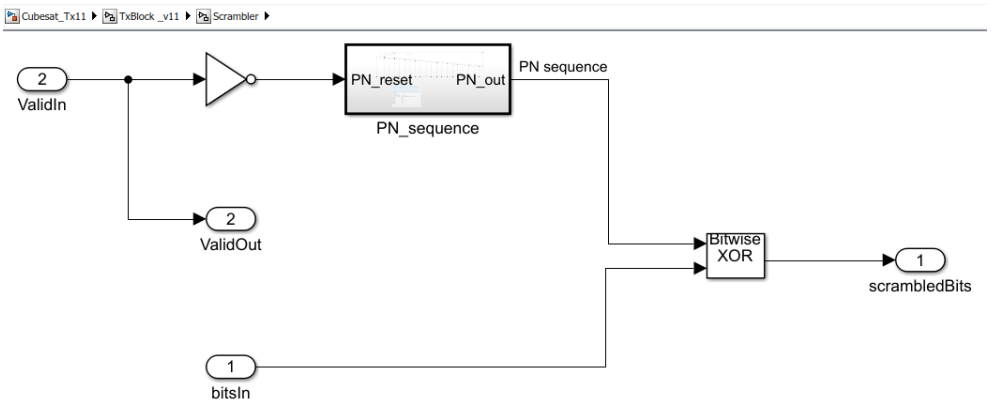


Fig. 4.11 Data scrambling subsystem

The CRC and coding block is depicted in Fig. 4.12. It consists of two main parts. First, there is a CRC calculation block that adds a 32-bit CRC to the subframe data, extending its length to 6016 bits accepted by the turbo coder. The CRC sequence is generated using the following polynomial:

$$x^{32} + x^{31} + x^{24} + x^{22} + x^{16} + x^{14} + x^{8} + x^{7} + x^{5} + x^{3} + x + 1$$

The output of the CRC calculation block is fed to one of two turbo coders in an alternating manner. We use two coders due to the processing delay of the coding process which could cause that the data supply for the subframe assembly is not sufficient. In order to avoid overlapping of output data,

a dedicated control mechanism has been implemented. The output codeword length varies with different coding rate values, possible lengths are shown in Table 4.4.

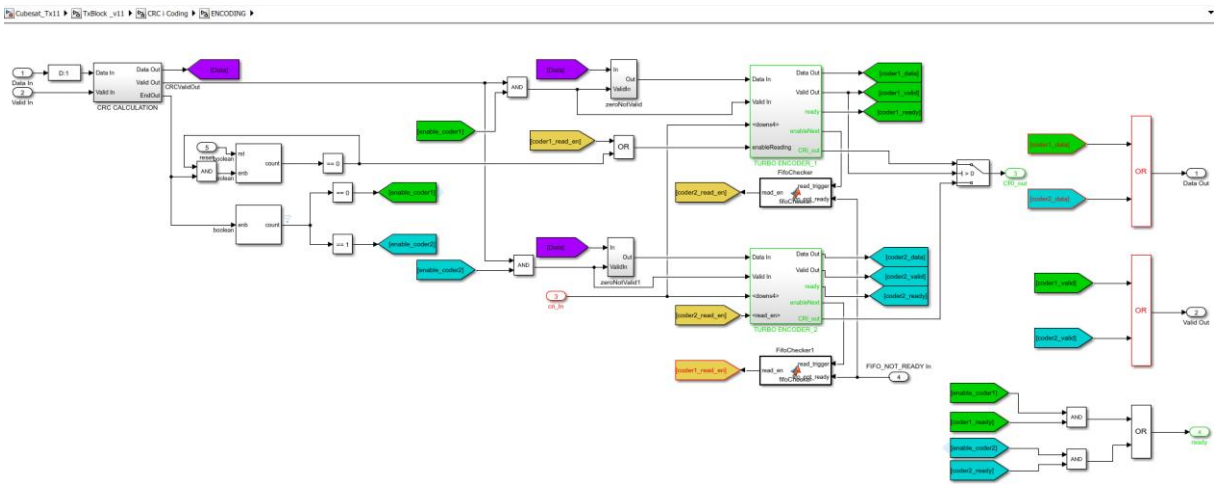


Fig. 4.12 CRC and channel coding subsystem

Table 4.4 CRI settings and coder output lengths

CRI value	Coding rate	Codeword length
0	0.91	6600
1	0.83	7260
2	0.76	7920
3	0.57	10560
4	0.38	15840
5	0.285	21120
6	0.19	31680

4.3.2. OQPSK modulation

The bits coming from the coding block are grouped to formed bit pairs which are mapped to OQPSK samples in the OQPSK modulation block denoted with 1 in Fig. 4.13. Its internal structure is shown in Fig. 4.14. Its task is to convert incoming bit pairs to OQPSK symbols and upsample it by the factor of 2. Each bit from the pair is upsampled. The bit in the imaginary branch of the symbol is delayed by half the initial sampling rate. Combining together the delayed and not delayed branches into a single complex sample produces an OQPSK symbol.

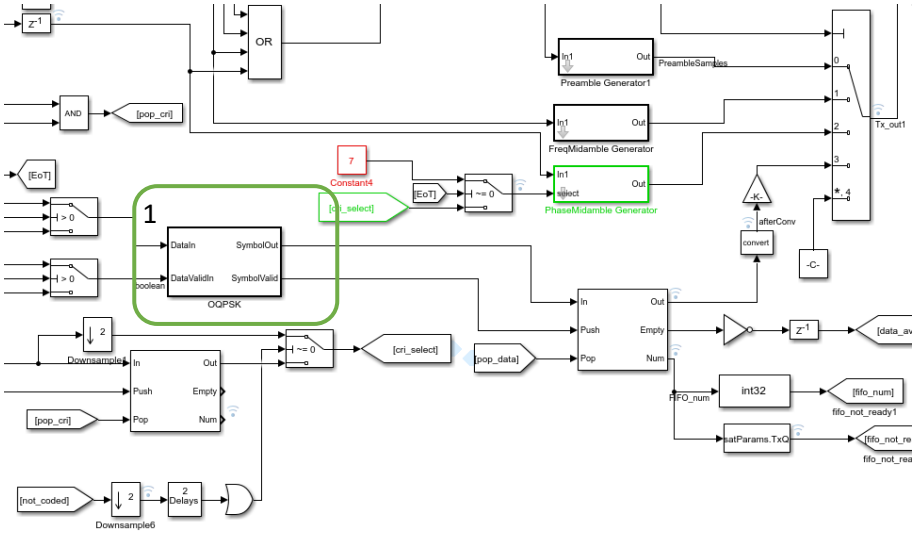


Fig. 4.13 OQPSK modulation part of the transmitter schematics

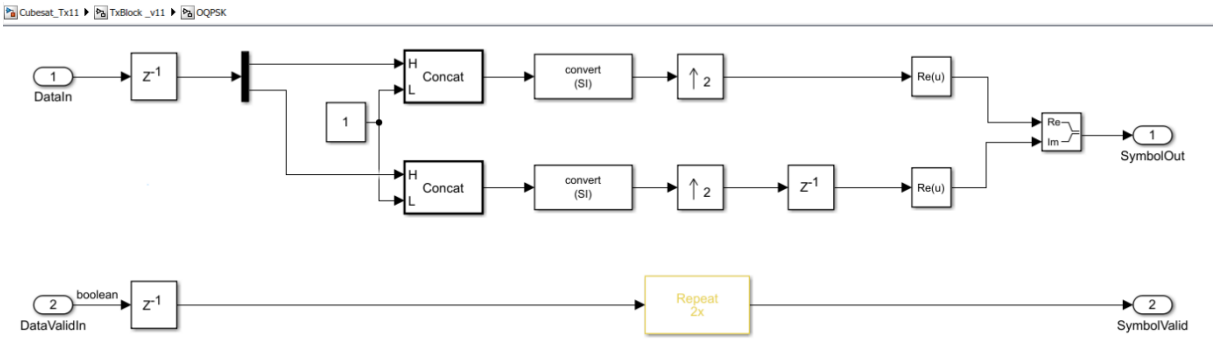


Fig. 4.14 OQPSK modulation

4.3.3. Radio frame creation

Each sample produced by the data preparation part of the transmitter descriped above is stored in a FIFO queue (block number 1 depicted in Fig. 4.15) which is controlled by a FSM implemented in the *TXStatus* block (number 2 in Fig. 4.15). The *TXStatus* is responsible for the correct assembly of subframes and radio frames. It has 6 inputs and 8 outputs as depicted in Fig. 4.16, the description of the input/output lines is given in Table 4.5

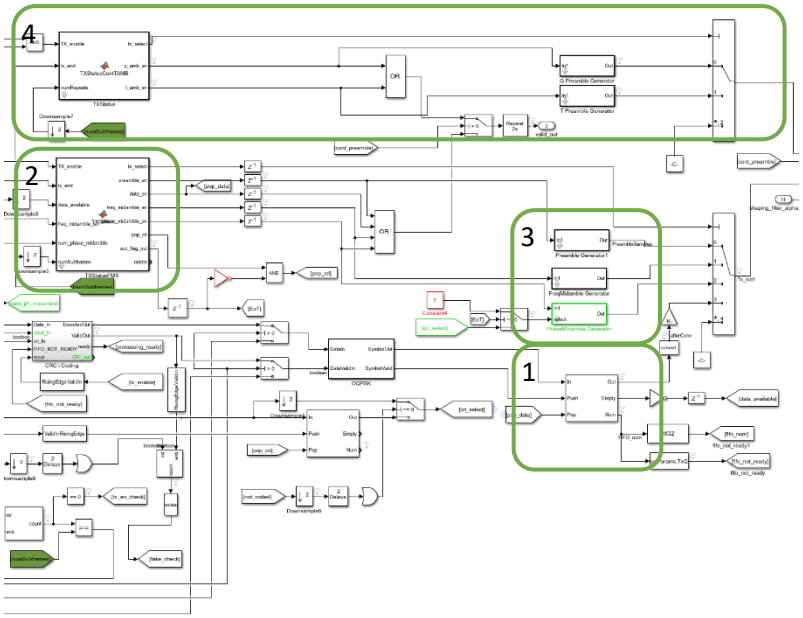


Fig. 4.15 Radio frame creation part of the transmitter schematics

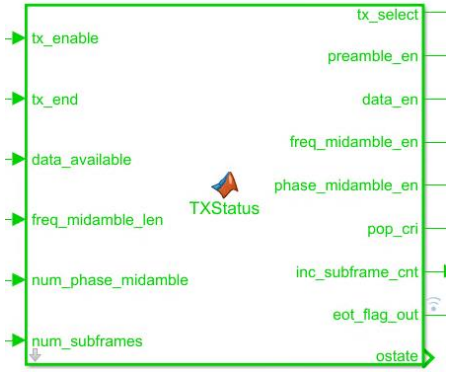


Fig. 4.16 Finite state machine used for assembling radio frames

Table 4.5 Inputs and outputs of TXStatus block

Name	Type	Description
tx_enable	Input	Indication whether the transmission is still enabled in software
tx_end	Input	Flag indicating that the high level of *tx_enable* signal has ended
data_available	Input	Indication whether OQPSK symbols are present in the symbol buffer
freq_midamble_len	Input	Length of the part of a preamble used in frequency offset estimation. There are 3 values possible: 544,1056 and 2080 (they are oversampled by 2)depending on the *F_AMB_sel* setting
num_phase_midamble	Input	A number of midambles used for phase offset estimation used in each subframe. The number of midambles is calculated based on coding rate and partial codeword length (330 OQPSK symbols)

num_subframes	Input	Number of subframes in the radio frame
tx_select	Output	Signal used to select appropriate part of the radio frame fed into the shaping filter
preamble_en	Output	Signal used to read initial preamble symbols from a lookup table. The length of the preamble is 768 symbols (they are oversampled by 2). The preamble consists of two parts. The First 256 samples are used for AGC purposes (*G_AMB*) and the remaining 512 symbols are used for time synchronization (*T_AMB*)
data_en	Output	Signal used to read OQPSK symbols from FIFO queue
freq_midamble_en	Output	Signal used to read from a lookup table preamble symbols used for frequency offset estimation. The length of *F_AMB* is dependent on the *freq_midamble_len*.
phase_midamble_en	Output	Signal used to read from a lookup table midamble symbols used for phase offset estimation. The length of the phase midamble is 166 (oversampled by 2)
pop_cri	Output	Signal used to get the currently used CRI value
inc_subframe_cnt	Output	A line used for counting transmitted subframes
eot_flag_out	Output	Signal used to generate an EOT frame
ostate	Output	Indication of the current state of the FSM (used for debugging)

Preamble and midambles are read from lookup tables stored in memory. In Fig. 4.17 a preamble generator is presented. Its operation is rather straightforward i.e. the enable signal on the input port enables the counter which generates an index from which we read the value from the lookup table for the output. The generators for the frequency offset estimation part of the preamble and midamble generator are identical in structure but differ in contents of the lookup table. The phase midamble apart from being used for phase offset estimation is used to distinguish the coding rate in the receiver. Each coding rate setting has its own form of midamble that is created by cyclically shifting base Zadoff-Chu sequence used for phase estimation purposes.

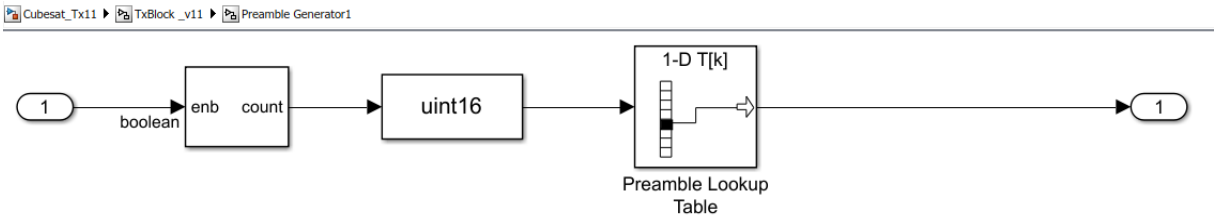


Fig. 4.17 Preamble generator subsystem

Apart from the 3 modes of transmission described in section 4.1.1, there is a fourth mode in which the transmitter will continuously transmit preambles (*G_AMB* with $n = number\ of\ subframes$ repeats of *T_AMB*). To control this mode of transmission FSM depicted in Fig. 4.18 was implemented. Its inputs and outputs are described in Table 4.6

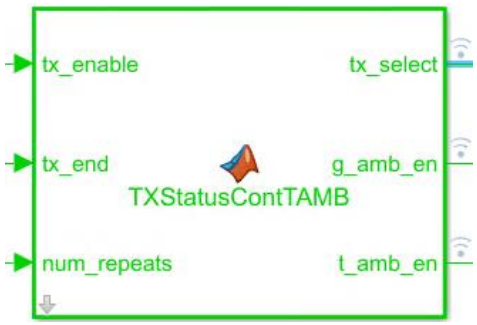


Fig. 4.18 Finite state machine used for assembling radio frames consisting of repeating preambles

Table 4.6 Inputs and outputs of TXStatusContTAMB block

Name	Type	Description
tx_enable	Input	Indication whether the transmission is still enabled in software
tx_end	Input	Flag indicating that the high level of *tx_enable* signal has ended
num_repeats	Input	The number of T_AMB preamble repeats, set by the *num_subframes* value
tx_select	Output	Signal used to select appropriate part of the radio frame fed into the shaping filter
g_amb_en	Output	Signal used to read *G_AMB* preamble symbols from a lookup table. The length of *G_AMB* is 256 symbols (they are oversampled by 2).
t_amb_en	Output	Signal used to read *T_AMB* preamble symbols from a lookup table. The length of *T_AMB* is 512 symbols (they are oversampled by 2).

4.3.4. Pulse shaping and transmitter output

The final step in the processing chain is pulse shaping and data type conversion of the samples to unsigned integer format accepted by the DAC depicted in Fig. 4.19. The pulse shaping is performed with a tunable root raised cosine filter. The tunable parameter of the filter is its roll-off factor α which can be set by the *shaping_filter_alpha_sel* input of the transmitter. There are two possible values of the α parameters i.e. 0.22 when the *shaping_filter_alpha_sel* inputs value is 1 and 0.35 otherwise. The input of the pulse shaping filter are the samples of the radio frames assembled in previous steps of the processing chain.

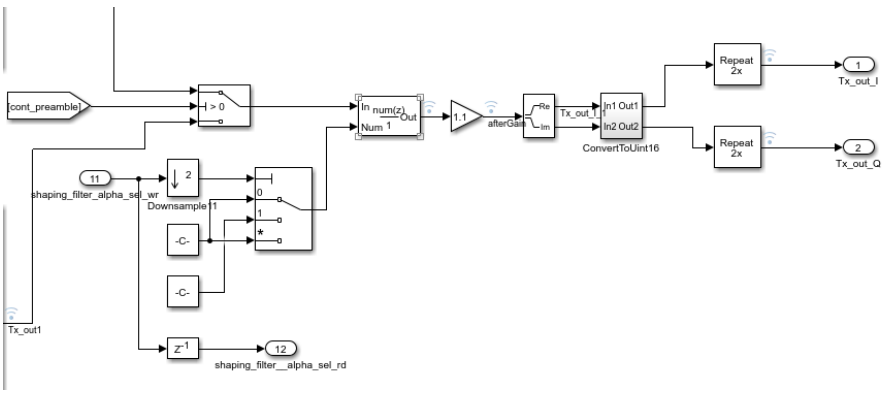


Fig. 4.19 Pulse shaping and output part of the transmitter

4.4. Custom-made IP core

4.5. Introduction

The FPGA-based signal processing routine, created with the aid of MATLAB HDL Coder, is encapsulated into a user-developed Vivado IP core. To incorporate a transmitter IP core into a Vivado block diagram, a Matlab HDL workflow coder is used. The workflow operates according to a given so-called reference design, which specifies the Vivado block diagram, the way in which the custom-made IP core is merged with it, and the board pin assignment (constraint file). The reference designs have a form of TCL scripts.
In the current project, a reference design by Analog Devices, dedicated to PicoZED (adrv9361z7035) board, is used as a baseline and adopted by the project researchers to meet the requirements of the TE0715 SoM, manufactured by Trenz Electronic.
The original reference design by Analog Devices seems to be most accurate in the cases where the IQ samples (for 2 transmit channels) are generated by the PS, and the PL is used, mainly, for interfacing AD9361. Signal processing in PL is an option (the user-specified IP core can be by-passed in some cases). There are four 16-bit input lines, and the data flow control is of a back-pressure type: the AD9361 transceiver orders samples from the PL, and the request is forwarded to the PS.
In the current project, the data transferred from PS to PL via DMA have the meaning of binary vectors instead of complex IQ samples; the data rate on the PS<>PL interface is significantly smaller than the symbol rate on the SoC<>AD9364 interface as there is nothing else but PL responsible for the physical-layer signal processing (scrambling, channel coding, interleaving, modulation, pulse shaping, etc.). As a consequence, it is more accurate to consider only one wide data line on the PS<>PL interface. Since it is not guaranteed that the PS is able to deliver new data vectors on the PL's request, it is necessary to wire a "data valid" line along with the data line.

4.5.1. Reference design customization

With the aim to overcome the disadvantages of the original reference design, the datapath is significantly modified. Four 16-bit data lines have been replaced with one 32-bit data line. Consequently, data streams are not interleaved anymore (interleaving required troublesome synchronization between the streams), so the blocks, responsible for stream interleaving and deinterleaving are removed. The data line is accompanied by a strobe line, missing in the original reference design by Analog Devices. The *utility_buffer* IP, originally placed between DMA interface and FIFO at the clock domains' border, has been removed. It was devoted to alleviate the problem of asynchronous type of data feed through DMA interface, but it has appeared to cause highly undesired random delays.
The reference design by Analog Devices features more IP cores useless from the perspective of the current project. In particular, it refers to the IP cores playing the role of HDMI, SPDIF, and I2S interfaces; their removal brings reasonable FPGA resources savings.
Some minor changes, shown in Fig. 4.20, have been made in the settings of *axi_ad9361* IP core, responsible for transferring FPGA-generated IQ samples of a passband signal to AD9364 transceiver.

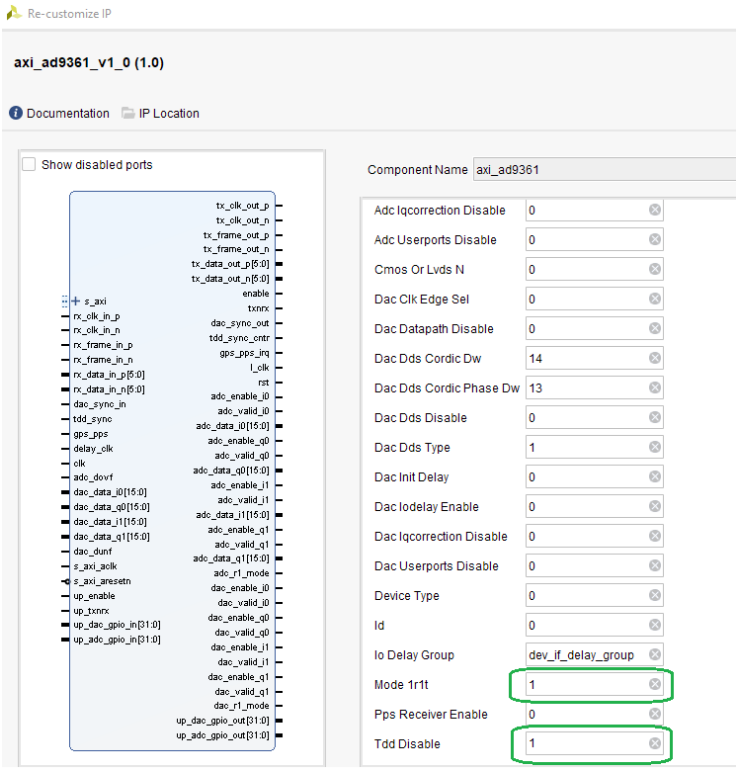


Fig. 4.20 Configuration window of axi_ad9361 IP core

In detail, *1R1T mode* is chosen to eliminate redundant support for two transmit channels (AD9364 features only one transmit channel). Additionally, *TDD disable* option is checked, since the AD9364 is forced to operate permanently in Tx mode by an SPI write to AD9364 registers instead of periodic Tx/Rx toggling, controlled via FPGA pins. Thanks to that, neither 24-bit TDD counter nor a few reference registers of the same size are implemented in FPGA. The rest of configuration fields of axi_ad9361 IP core take the default values. DDS feature is enabled for testing purposes.

Another improvement has been made in the domain of custom-made IP core clocking. In the original reference design, the user's IP core is clocked by the AD9364 clock divided by 2 (or by 4 in the case of 2 transmit streams – not applicable to the current design). It limits the system capability of serial data processing, since half of the clock cycles are not usable. Instead, the custom-made IP core is now clocked with the original AD9364 clock (*rx_clk*), distributed throughout the FPGA device directly from a respective BUFG element, as shown in Fig. 4.21.

The decision to eliminate a separate clock domain for custom-made IP core results with a simpler clock cross-domain management: there is only one clock-domain crossing in the data path, handled safely by means of a FIFO in *axi_ad9361_dac_dma* IP core. To transfer commands and status messages data between the time domains (*fpga_clk0* and *rx_clk*) through AXI4-Lite, a 3-stage synchronizer is placed in the *axi_cpu_interconnect* IP core. Together with AXI protocol handshaking, it guarantees safe transfers. The principles of operation of *axi_cpu_interconnect* is explained according to Fig. 4.22. The input AXI bus (S00_AXI) interfaces Zynq PS – it is clocked by the fabric *fpga_clk0* clock. The interconnect IP core is responsible for dispatching the commands to numerous IP cores, featuring AXI4 interface, via the output AXI buses: M00_AXI … M04_AXI. All such IP cores except for the custom-made data processing IP core are clocked by the same *fpga_clk0* clock. In such cases, the couplers visible in Fig. 4.22 are transparent (as explicitly shown for M03_AXI). For M04_AXI, connected with the IP core clocked by *rx_clk*, the *auto_cc* sub-block is inserted. It contains the abovementioned 3-stage synchronizer.

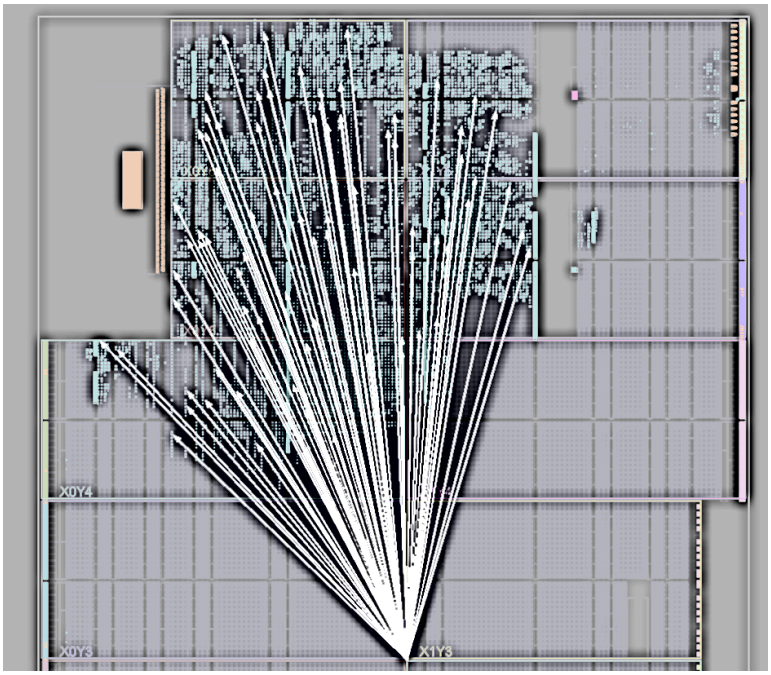


Fig. 4.21 AD9364 clock distribution throughout the FPGA device

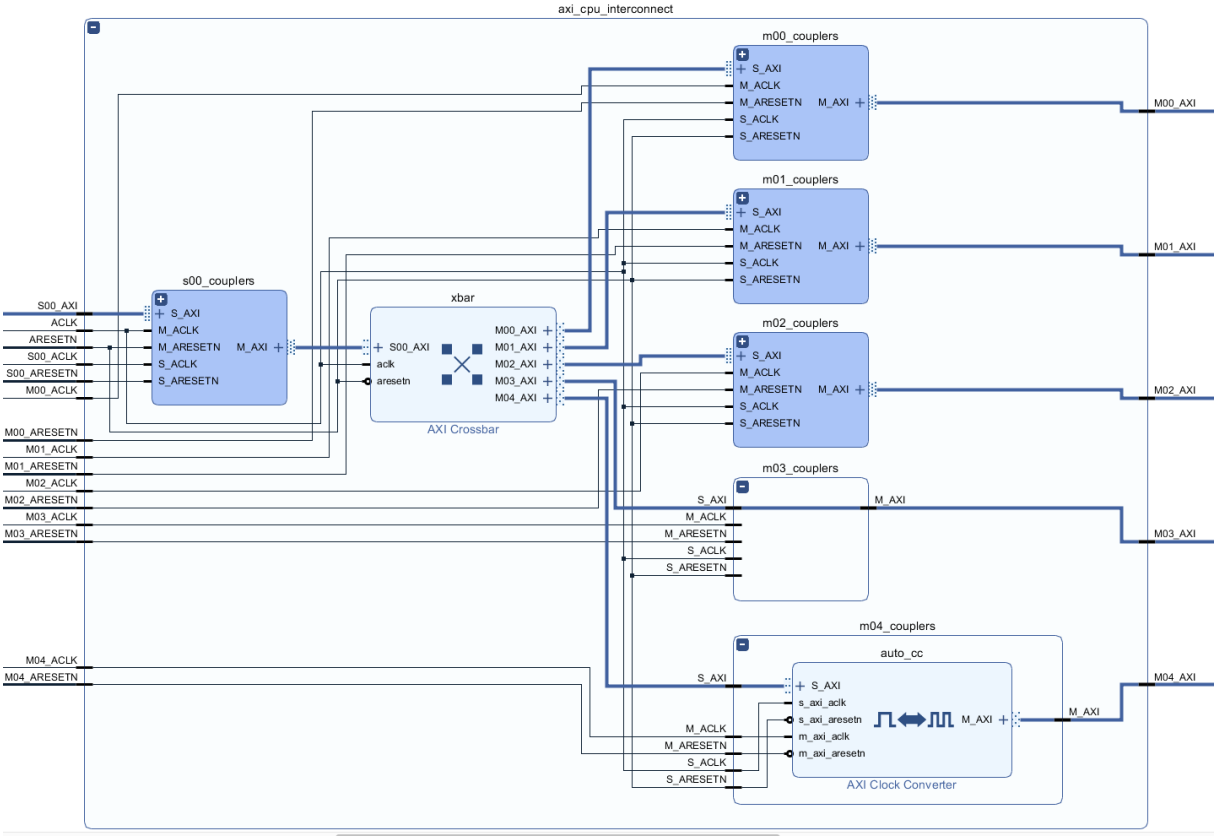
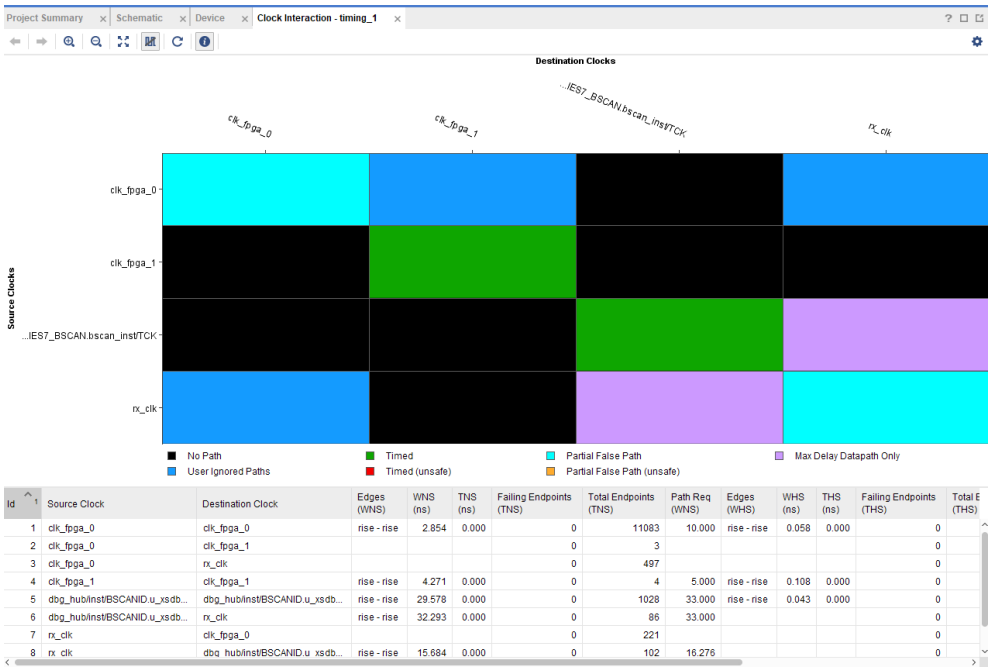


Fig. 4.22 Internal structure of *axi_cpu_interconnect*

Taking into account asynchronous data transfer between the clock domains (for both data path and the control/status AXI channel), it is desired to constrain intra-clock paths on FPGA as false paths, thereby instructing the Vivado placer to ignore them; it helps overcome timing-related issues when

routing. Fig. 4.23 proves that the paths between clock domains: *fpga_clk0* and *rx_clk* are successfully set as false paths in Vivado.



Id	Source Clock	Destination Clock	Edges (WNS)	WNS (ns)	TNS (ns)	Failing Endpoints (TNS)	Total Endpoints (TNS)	Path Req (WNS)	Edges (WHS)	WHS (ns)	THS (ns)	Failing Endpoints (THS)	Total E (THS)
1	clk_fpga_0	clk_fpga_0	rise - rise	2.854	0.000	0	11083	10.000	rise - rise	0.058	0.000	0	
2	clk_fpga_0	clk_fpga_1				0	3					0	
3	clk_fpga_0	rx_clk				0	497					0	
4	clk_fpga_1	clk_fpga_1	rise - rise	4.271	0.000	0	4	5.000	rise - rise	0.108	0.000	0	
5	dbg_hub/inst/BSCANID.u_xsdb...	dbg_hub/inst/BSCANID.u_xsdb...	rise - rise	29.578	0.000	0	1028	33.000	rise - rise	0.043	0.000	0	
6	dbg_hub/inst/BSCANID.u_xsdb...	rx_clk	rise - rise	32.293	0.000	0	86	33.000				0	
7	rx_clk	clk_fpga_0				0	221					0	
8	rx_clk	dbg_hub/inst/BSCANID.u_xsdb...	rise - rise	15.684	0.000	0	102	16.276				0	

Fig. 4.23 Clock interaction report for the implemented design in Vivado

Not only is the MATLAB HDL workflow responsible for generating appropriate interfaces of the custom-made IP core and incorporating it into the reference design, but also for attaching extraordinary constraint files to the project. The constraint files contain the settings related to the hardware pinout, clock frequency, false paths, etc. The pinout for Trenz board differs from PicoZED pinout in terms of voltage standard and exact position of SoC<>AD9364 connectors. For that reason, the constraint files have been updated to meet the project requirements.

4.5.2. The use of Simulink HDL Workflow Advisor

For ease of use the customized reference design targeting Vivado 2018.2 has been stored and integrated with the HDL Workflow of Matlab 2019a under the name of *TE0715byMK* – it should be chosen as the Target platform in Step 1.1 of the HDL Workflow Advisor, as shown in Fig. 4.24. Step 1.4, shown in Fig. 4.25, brings the possibility to connect the inputs and outputs of the developed Simulink block diagram to appropriate reference design wires (aka target platform interfaces). The meaning of specific target platform interfaces is explained in Table 4.7. It does not include the Simulink ports attached to the AXI4-Lite interface, used to send control commands from PS to PL and read diagnostic messages in the opposite direction.

After passing checks in Steps 2.1-2.4, the HDL code for the custom-made IP core is generated in Step 3.2 of HDL Workflow Advisor. The generated IP core is deposited in a folder specified by the user and can be manually placed into any Vivado block diagram. However, the customized reference design *TE0715byMK* features the possibility to automatically integrate the IP core with the block diagram. It can be done in Step 4.1 of HDL Workflow Advisor. If the process ends successfully, a link to a new-created Vivado project appears in a log window of HDL Workflow Advisor, as shown in Fig. 4.26). Clicking the link launches Vivado and the project opens. It is not suggested to run remaining steps of HDL Workflow Advisor, as they are accurate only for the case when FPGA processing is controlled by Simulink (a kind of hardware-software co-simulation).

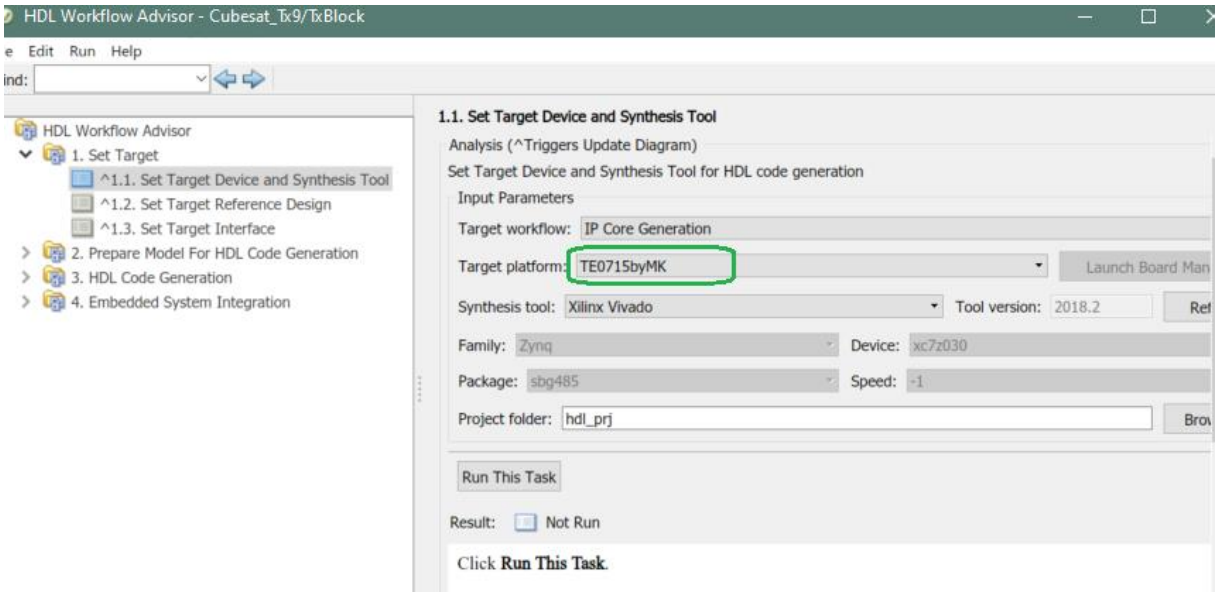


Fig. 4.24 Step 1.1 of HDL Workflow Advisor

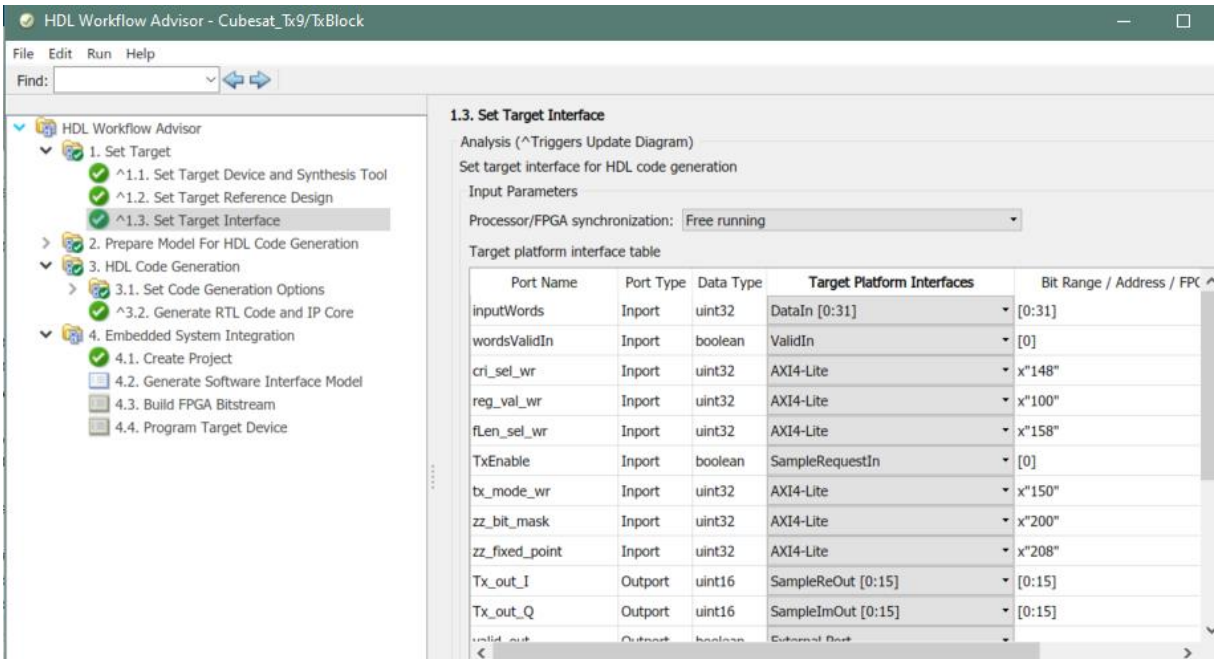


Fig. 4.25 Step 1.4 of HDL Workflow Advisor

Table 4.7 Target platform interfaces in HDL Workflow Advisor

Name	Type	Mating pin on Simulink diagram	Description
DataIn	Input	*inputWords*	32-bit data vectors send from PS via DMA
ValidIn	Input	*wordsValidIn*	Strobe line for input data
SampleRequestIn	Input	(not used)	This line is periodically strobed by *axi_ad9361* IP core to request subsequent IQ samples; for 1R1T mode, a pulse appears every 2nd AD9364 clock cycle
SampleReOut *SampleImOut*	Output	*Tx_out_I* *Tx_out_Q*	16-bit IQ samples of the passband signal, represented in 2's complement format; actually, 4 least significant bits are unimportant since AD9364 is equipped with a 12-bit DAC
DataRequestOut	Output	*TxLoadReq*	Request for a new data vector destined for the FIFO at the clock-domain crossing

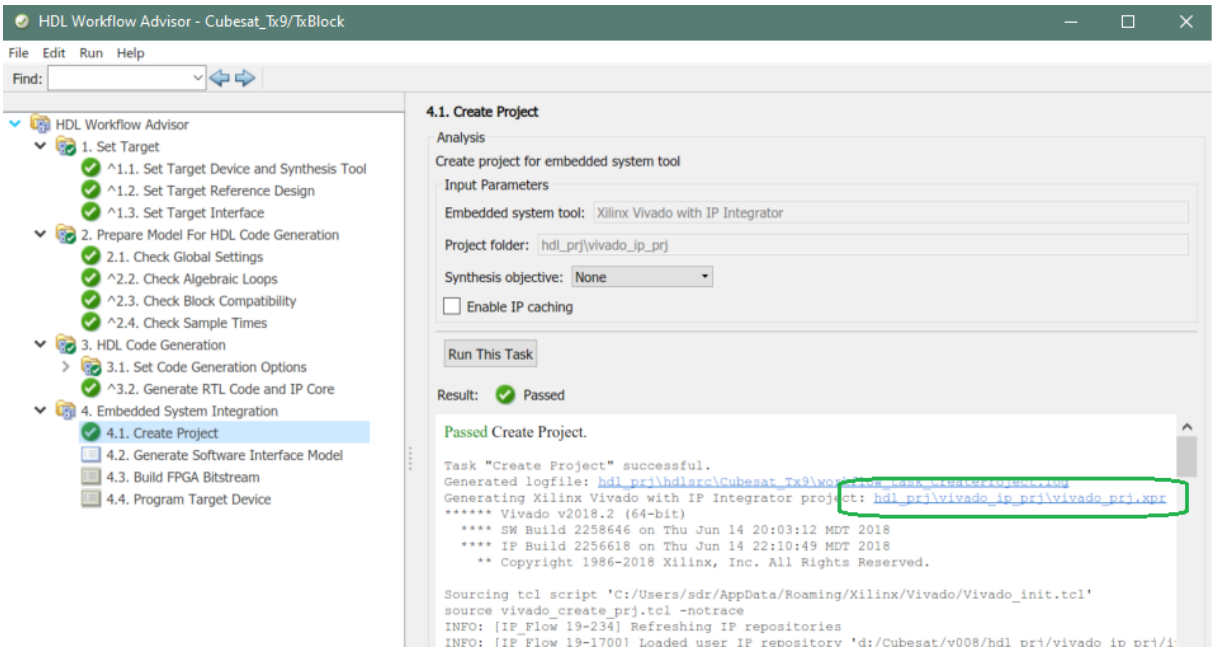


Fig. 4.26 Result of successful execution of the last step of HDL Workflow Advisor

4.5.3. Vivado project details

The complete block design of Vivado project is shown in Fig. 4.27, while a closeup on the custom-made IP core is presented in Fig. 4.28. One can easily recognize the target platform interfaces of the IP core, described in Table 4.7. There are some additional lines: AXI4-Lite bus (to receive control commands from PS and send status messages), as well as reset and clock lines (separate for AXI bus sub-module and the rest of the IP core). Since the clock-domain crossing is located in

axi_cpu_interconnent, the whole custom-made IP core clocking belongs to a single clock domain of *rx_clk*, originated from *l_clk* pin of *axi_ad9361* block. The IP core reset line is conjugated with PS reset by *util_ad9361_divclk_reset block*, responsible for transferring the PS-generated reset to *rx_clk* clock domain. Note that the IP core must be additionally resetted by an AXI write after AD9364 has finished all callibrations. Failure to do so might lead to unpredictable IP core operation and metastability.
The design is synthesized with the clock constraints specified according to the most demanding 20 MHz bandwidth transmission mode. The mode choice is managed by appropriate frequency setting of *rx_clk* on AD9364 and impacts the speed of data passing through the whole data path in the *rx_clk* domain.

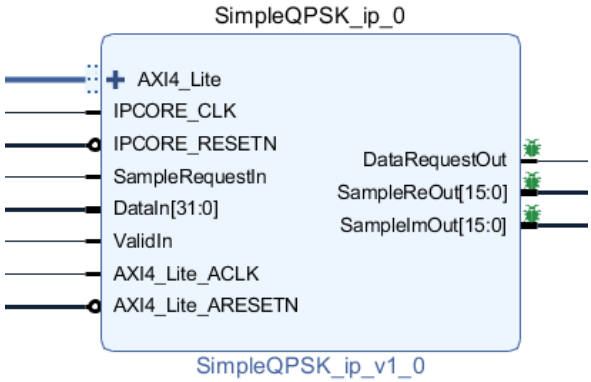


Fig. 4.28 Custom-made IP core

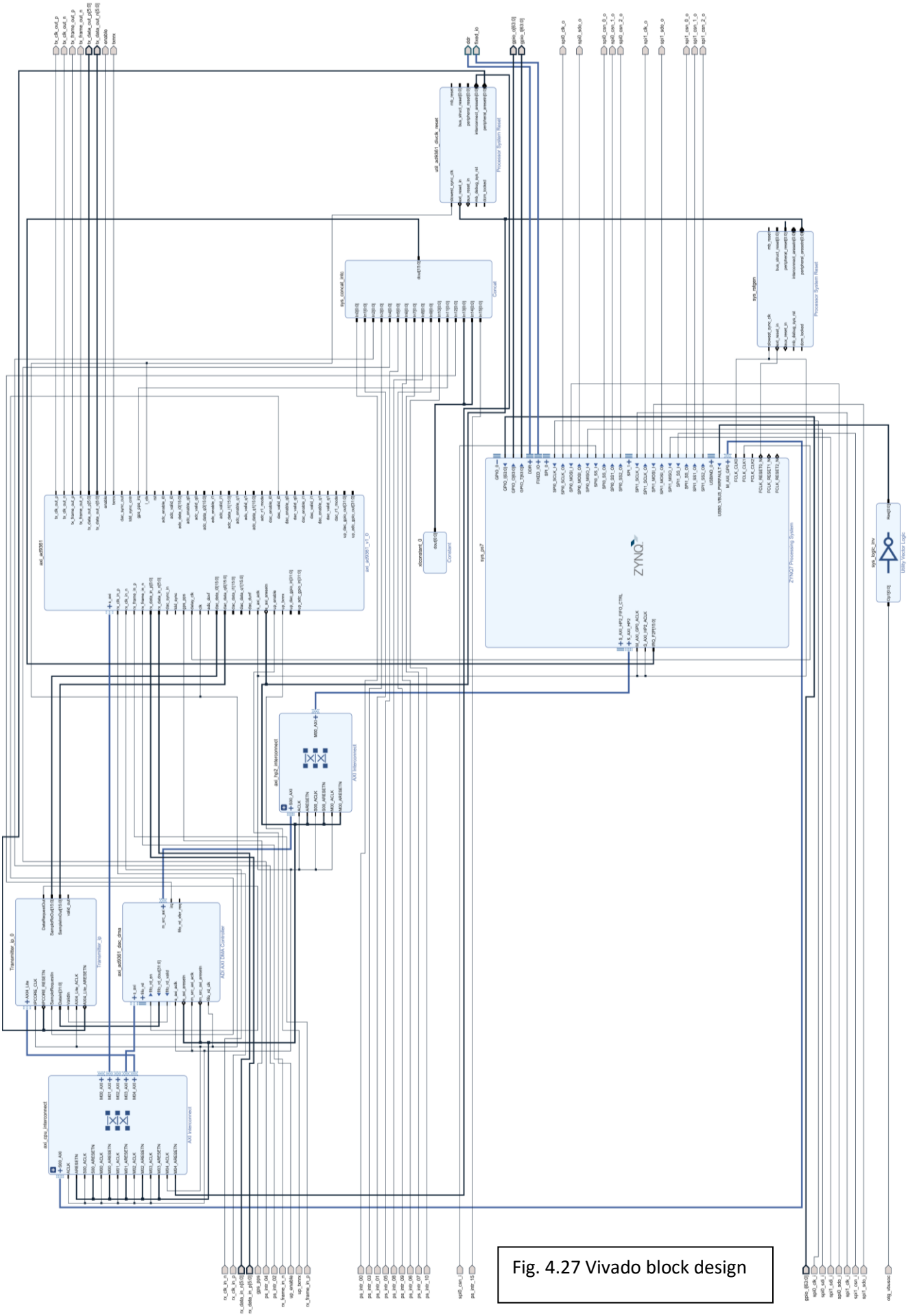


Fig. 4.27 Vivado block design

5. Transmitter Implementation – PS part (software)

5.1. TX vs RX – disambiguation

Processing System part is run on two boards, mainly:

- Trenz board (board version: TE0715-04-30-1I3),
- MITX (aka MiniITX board with Z100 FPGA, board version: Mini-ITX-7Z-ASY-G).

The most recent HDL version Analog Devices' HDL that supports AVNET's MiniITX box is hdl_2017_r1, while HDL version that is compatible with Trenz board is hdl_2018_r2. A chain of dependencies caused by such a seemingly unimportant version difference results in two different Linux O/S versions that run on those boards. It is of an uttermost importance to note that such a choice of O/S version was dictated by technical arguments, not by dogmatic or opinionated ones. Details are shown in the table below.

Side	Board	HDL	O/S	Vivado	Matlab
TX	Trenz TE0715-04-30-1I3	hdl_2018_r2	2018_R2	2018.2	2019a
RX	Mini-ITX-7Z-ASY-G	hdl_2017_r1	2017_R1	2016.4	2017b

5.2. Operating System

Transmitter module is based on Trenz board TE0715-04-30-1I3, a consequence of which is using Analog Devices' Linux version `2018_R2` (kernel 4.14.0). [Note, however, that due to the unavailability of the final board at the time of preparation of this document, the chosen version of O/S was tested only on Trenz's motherboard TE0705-04. As a result, this O/S version is not yet decided to be final.]

5.2.1. Cross-compiling Tools

Linux kernel together with all supporting libraries and tools were built with GCC 11.2.0. [As of the preparation time of this document, the final version of GCC is practically frozen, although it still might be changed to a different one if necessary.]

5.2.2. Shell

O/S is interfaced via `ash` (Almquist shell).

5.2.3. Kernel Configuration

Kernel was configured using a customized Xilinx configuration provided by Analog Devices Inc. in the source tree of Linux kernel under the name [`xilinx_zynq_defconfig`]. The customization involved additional configuration of:

- DMA Engines (with Xilinx DMA Engines),
- AXI DMAC,
- AD9361,
- AD9517, and

- AXI DDS (Digital Direct Synthesizer).

5.2.4. FPGA Driver and Kernel Modules

FPGA Driver is loaded as a kernel module during the system boot-up. Detailed documentation of the driver attributes can be found in the auto generated documentation file [10]. Delivered as an attachment to this document. [Note, however, that eventual further changes to the implementation of the IP Core may require corresponding updates to the driver documentation].

5.2.4.1. Driver Attributes Description

As mentioned previously, the documentation describing driver attributes is automatically generated. A similar approach is used to create the driver code (obviously, only the most redundant parts). This is achieved by describing driver's attributed in a YAML file, which in turn is used to generate driver's and Latex's code for the driver itself and its documentation, respectively. Such an approach was used in order to easily keep in sync changes made to the driver and its documentation. An example entrance that describes the `hardware_version` attribute is shown in the listing below.

```
# Snip...
- name: hardware_version
  generated_driver_code:
[offsets,driver_entrances,driver_attributes_short,help_messages]
  type: __u32
  rd_offset: 0x04■■
  wr_offset: null
  rd_buffer_size: 16
  wr_buffer_size: 10
  rd_function: scnprintf
  wr_function: kstrtou32
  mask_spec: null
  trx_side: [tx,rx]
  help_msg: |-
      None
  description: |-
      Version of bit-stream hardcoded in the hardware.
      It is not possible to write into this register!
  hardware_name: DEV_ID
  available_values_hardware: null
# Snip...
```

5.2.4.2. Kernel Modules Loading

Despite most of the kernel modules being compiled directly into the kernel itself, the FPGA driver is not. Such an approach allows eventual changes to the driver without the need of recompiling the whole kernel. The script used to load / unload kernel modules (which is located in [`/etc/rc.d/init.d/modules_conf`]) is presented below. Note that contrary to most arguments that such scripts accept, this particular one also accept arguments: `load`, `unload` and `reload` (which corresponds to standard: `start`, `stop` and `restart`, respectively). Such an approach allows using semantic that is closer to module loading / unloading. The configuration file that drives the modules loading is located in [`/etc/modules.d/modules.conf`] and it is discussed in the next section.

```
#!/bin/ash
```

```
#
# modules auto loading/unloading
#

. /etc/rc.d/init.d/functions

modules_config_file="/etc/modules.d/modules.conf"

modules_load_unload() {
    local modprobe_param="${1}"
    local msg="loading"
    [ "${modprobe_param}" == "-r" ] && msg="unloading";


    echo -n "${0}: Checking whether ${modules_config_file} exists and is valid: "
    [ -r ${modules_config_file} ] && grep -qv "^($|#)" ${modules_config_file}
    local ERR=$?
    [ 0 == ${ERR} ] && true || false
    check_status
    [ 0 == ${ERR} ] || { echo "${0} Not ${msg} modules! Problems with
${modules_config_file} file!"; exit 0; }

    sleep 0.1

    while read module args; do
        # Skipping blank or commented-out lines"
        case "${module}" in
            ""|"#"*) continue;;
        esac

        modprobe ${modprobe_param} ${module} ${args}
        ERR=$?
        echo -n "${0}: ${msg} ${module} with params: `[ "" != "${args}" ] && echo
${args} || echo -*NONE*-`: "
        [ 0 == ${ERR} ] && true || false
        check_status
        sleep 0.1

    done < ${modules_config_file}

}

case "$1" in
    start|load)
        modules_load_unload ""
    ;;

    unload|stop)
        modules_load_unload "-r"
    ;;
    reload|restart)
        $0 stop

        sleep 1

        $0 start
    ;;
    *)
        echo "Usage: ${0} {start|stop|restart|load|unload|reload}"
        exit 1
    ;;
esac

exit 0
```

5.2.4.2.1. Modules configuration file

The aforementioned modules configuration file (which is located in [`/etc/modules.d/modules.conf`]) allows loading arbitrary modules (not only the FPGA driver). The example script is shown in the listing below. Besides specifying modules to load, it also allows specifying module's parameter, e.g., in the listing below module `fpgatrx` is loaded with parameter `DEBUG` set to `1`. Kernel object files that contain the modules' code are located in [`/lib/modules/$(uname -r)/kernel/drivers/`], where [`$(uname -r)`] is release of the running kernel, in our case it is: `4.14.0-xilinx-ge77ffb40e9a0-dirty`. [But as already mentioned this release might still be subject to an eventual change.]

```
# File: /etc/modules.d/modules.conf
#
# In order to load module at the system boot-up, add:
#
# module_name module_param_1 module_param_2
#

fpgatrx DEBUG=1
```

5.2.5. Libraries and Tools

O/S is delivered with tools and libraries described in the table below. [Please note that the final versions of some of these artifacts may be changed if deemed necessary.]

Tools / Library	Version TX	Version RX	Description
Binutils	2.27		Set of tools and libraries for building binary executable(s), e.g., liker, assembler, etc. All build for ARM Cortex A9 processor, but without dubious optimisation flags.
Busybox	1.24.2		Swiss-army-knife toolbox with standard set of tools for working in a Linux environment. (All tools are delivered as via symbolic links to one executable.)
IANA-ETC	2.30 patched		Data / information package for network protocols and services.
MPC	1.0.3		Arbitrary precision floating-point complex arithmetic library. (GCC dependency.)
MPFR	3.1.4		Arbitrary precision floating-point library. (GCC dependency.)
musl-libc	1.1.19		Standard C library for embedded systems.
zlib	1.2.11		Data compression library.
netplug	1.2.9.2		GNU/Linux daemon for network services.
Dropbear	2018.76		Lightweight implementation of SSH library.
LibXML2	2.9.8		XML parsing library, implemented in C. libiio dependency.
Boost	1.67		Boost – an umbrella of C++ utility libraries. Most of them are header-only libraries. Only three are installed on the final system: `libboost_atomic`, `libboost_chrono`,

			`libboost system`.
tree	1.7.0	1.7.0	Command line recursive directory viewer / explorer.
libiio	0.14		Hardware abstraction layer library via IIO module (Industrial Input / Output) for GNU/Linux. Mainly used to
gtest	1.11.0	1.11.0	Unit test library.
gflags	2.2.1		Command line parsing library.
googlebenchmark	1.6.1	1.6.1	Benchmarking library
{fmt}	8.1.1	8.1.1	Text formatting library
iproute2	ss190197		Network support tools.

5.2.6. Device Tree and Node Configuration

In order to easily distinguish between various systems configurations we add to the device tree file parameter describing specific configuration of the board. An excerpt from a device tree is shown below.

```
/{
    wzldevicemode {
        mode = "trenz";
    };
};
```

On the running system, current [`wzldevicemode`] (WZL here stands for **W**ireless **Z**YNQ **L**ab) can be read form [`/sys/firmware/devicetree/base/wzldevicemode/mode`] file. In the case the node describing the current configuration changes, the file with the fixed name that contains the actual location of the current configuration is located in [`/etc/radio/wzl-dev-mode-file-location`]. Such an approach ensures a single reference point to the actual location of the the file describing the device mode.

Definition of the FPGA implementation of the custom made IP Core is also provided in the device tree (in the `FPGA/amba_pl` section). The entrance in the device tree for `SimpleQPSK` IP Core is shown below.

```
/{
    amba_pl: amba_pl {
        #address-cells = <1>;
        #size-cells = <1>;
        compatible = "simple-bus";
        ranges ;
        SimpleQPSK_ip_0: SimpleQPSK_ip@43c00000 {
            compatible = "xlnx,SimpleQPSK-ip-1.1";
            reg = <0x43c00000 0x10000>;
        };
        // ...
    };
};
```

5.2.7. RF Configuration

RF configuration files reside in the [/etc/radio] directory on the primary/root partition. An example listing of subset of its directories is shown below.

```
/etc/radio/filters
├── cubesat-filter-v0001.ftr
├── cubesat-filter-v0002-1R1T-mode.ftr
├── cubesat-filter-v0003-61dot44.ftr
├── cubesat-filter-v0004-30dot72.ftr
├── cubesat-filter-v0005-7dot68-pll-ad9364.ftr
├── cubesat-filter-v0006-15dot36-pll-ad9364.ftr
└── lte_5MHz.ftr

/etc/radio/current/
├── ad9361-config.gflags
├── config-dispatcher.gflags
├── session-plan.yaml
└── session-scheduler-config.gflags
```

As can be deduced from the listing above, definition of FIR filters is in [/etc/radio/filters] directory. An example FIR filter definition file is show below. Its format is self-explanatory.

```
$ head -12 /etc/radio/filters/lte_5MHz.ftr
# Generated with AD9361 Filter Design Wizard 16.1.3
# MATLAB 9.2.0.538062 (R2017a), 25-May-2018 16:55:22
# Inputs:
# Data Sample Frequency = 7680000 Hz
TX 3 GAIN 0 INT 2
RX 3 GAIN -6 DEC 2
RTX 983040000 122880000 61440000 30720000 15360000 7680000
RRX 983040000 122880000 61440000 30720000 15360000 7680000
BWTX 4372840
BWRX 4694670
-5,-10
0,-21
...
```

File [lte_5MHz.ftr] is used only for demonstration without disclosing actual details of the FIR filters used in the real system (mainly number and values of consecutive filter taps).

5.2.8. Application Configuration Files

Besides FIR filters configuration files [/etc/radio] directory also contains

```
/etc/radio/current/
├── ad9361-config.gflags
├── config-dispatcher.gflags
├── session-plan.yaml
└── session-scheduler-config.gflags
```

These files contain configurations of the custom applications and are described more thoroughly further in the document. The [`/etc/radio/current`] directory is in fact a soft link to the actual directory that contains configuration for a particular transmission side (TX or RX).

5.2.9. Pre-O/S Components and Boot Sequence / Order

Boot sequence on ARM-based hardware is divided into separate stages. Initially the FSBL (First Stage Boot Loader) prepares hardware, initializes CPUs and starts SSBL (Second Stage Boot Loader), which in our case is U-boot. Then SSBL/U-Boot decompresses the Linux kernel image and loads it together with a device tree describing the hardware and peripherals into memory. Next, the control is passed to the kernel, which boots itself, launches [`/sbin/init`] program that finalizes the Linux booting-up and starts services and applications required for ensuring the whole system is in an operational state.

5.2.9.1. FSBL

Beside standard initialization, Xilinx's FSBL allows configuring additional hardware via FSBL hooks. For example, patch provided by Trenz allows configuring SI5338 module.

In the next three sections we show logs from FSBL, U-Boot and loading Linux kernel. These logs can be used as a reference for adjusting and / or fine-tuning different versions of the mentioned software components. They should be treated more as a guidance that a gold-standard when preparing custom solutions.

5.2.9.1.1. FSBL Boot Logs

FSBL loading logs are presented below. The manifest info section describes internals used to create a final [`BOOT.BIN`] file, it is not a necessary part and it is used solely for simplifying identification of the loaded bitstream.

```
MANIFEST INFO:
-----------------------
HDF FILE:                 sr-cubesat-trenz-tx-v0011.hdf
HDF GIT SHA:              8b041aee83724fadcd64867d266cabf8cdbfb005
IP CORE REPORT PATH:
d:/TrenzPrebuild/system/ip_lib/SimpleQPSK_ip_v1_1/doc/doc_arch_axi4_lite.jpg
ZYNQ XTOOLCHAIN GIT SHA:  bad57fe9919f23f83c19aeeb71f0a3bb37e2e70a
FSBL build date:          Tue, 14 Dec 2021 17:12:32 +0100
-----------------------
Xilinx Zynq First Stage Boot Loader (TE + PUT/TGM modified)
Release 2018.2  Dec 14 2021-17:12:50
```

5.2.9.2. SSBL / U-boot

U-boot logs are only for the reference.

```
U-Boot 2018.01 (Oct 11 2021 - 16:38:10 +0200) Xilinx Zynq ZC702

Board: Xilinx Zynq
Silicon: v3.1
I2C:   ready
DRAM:  ECC disabled 1 GiB
MMC:   sdhci@e0100000: 0 (SD)
** No device specified **
Using default environment

In:    serial@e0000000
Out:   serial@e0000000
```

```
Err:   serial@e0000000
Board: Xilinx Zynq
Silicon: v3.1
Net:   ZYNQ GEM: e000b000, phyaddr ffffffff, interface rgmii-id
eth0: ethernet@e000b000

U-BOOT for petalinux

ethernet@e000b000 Waiting for PHY auto negotiation to complete..... TIMEOUT !
Hit any key to stop autoboot:  0
reading uEnv.txt
486 bytes read in 12 ms (39.1 KiB/s)
Loaded environment from uEnv.txt
Importing environment from SD ...
Running uenvcmd ...
Copying Linux from SD to RAM...
reading uImage
4076304 bytes read in 238 ms (16.3 MiB/s)
reading devicetree.dtb
10683 bytes read in 17 ms (613.3 KiB/s)
** No boot file defined **
```

5.2.9.3. Linux Kernel

Linux kernel logs are only the reference.

```
## Booting kernel from Legacy Image at 03000000 ...
   Image Name:   Linux-4.14.0-xilinx-ge77ffb40e9a
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    4076240 Bytes = 3.9 MiB
   Load Address: 00008000
   Entry Point:  00008000
   Verifying Checksum ... OK
## Flattened Device Tree blob at 02a00000
   Booting using the fdt blob at 0x2a00000
   Loading Kernel Image ... OK
   Loading Device Tree to 07ffa000, end 07fff9ba ... OK

Starting kernel ...

Booting Linux on physical CPU 0x0
Linux version 4.14.0-xilinx-ge77ffb40e9a0-dirty (tgm@asus) (gcc version 8.3.0
(GCC)) #1 SMP PREEMPT Mon Oct 25 18:38:15 CEST 2021
CPU: ARMv7 Processor [413fc090] revision 0 (ARMv7), cr=18c5387d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
OF: fdt: Machine model: xlnx,zynq-7000
Memory policy: Data cache writealloc
cma: Reserved 16 MiB at 0x3f000000
random: fast init done
percpu: Embedded 16 pages/cpu @ef7cf000 s35084 r8192 d22260 u65536
Built 1 zonelists, mobility grouping on.  Total pages: 260608
Kernel command line: console=ttyPS0,115200 root=/dev/mmcblk0p2 rw earlyprintk
rootfstype=ext4 rootwait
PID hash table entries: 4096 (order: 2, 16384 bytes)
Dentry cache hash table entries: 131072 (order: 7, 524288 bytes)
Inode-cache hash table entries: 65536 (order: 6, 262144 bytes)
Memory: 1012868K/1048576K available (6144K kernel code, 266K rwdata, 1672K
rodata, 1024K init, 152K bss, 19324K reserved, 16384K cma-reserved, 24576)
Virtual kernel memory layout:
    vector  : 0xffff0000 - 0xffff1000   (   4 kB)
    fixmap  : 0xffc00000 - 0xfff00000   (3072 kB)
    vmalloc : 0xf0800000 - 0xff800000   ( 240 MB)
    lowmem  : 0xc0000000 - 0xf0000000   ( 768 MB)
    pkmap   : 0xbfe00000 - 0xc0000000   (   2 MB)
    modules : 0xbf000000 - 0xbfe00000   (  14 MB)
      .text : 0xc0008000 - 0xc0700000   (7136 kB)
      .init : 0xc0900000 - 0xc0a00000   (1024 kB)
      .data : 0xc0a00000 - 0xc0a42a80   ( 267 kB)
```

```
       .bss : 0xc0a42a80 - 0xc0a68e44   ( 153 kB)
Preemptible hierarchical RCU implementation.
        RCU restricting CPUs from NR_CPUS=4 to nr_cpu_ids=2.
        Tasks RCU enabled.
RCU: Adjusting geometry for rcu_fanout_leaf=16, nr_cpu_ids=2
NR_IRQS: 16, nr_irqs: 16, preallocated irqs: 16
efuse mapped to f0800000
slcr mapped to f0802000
L2C: platform modifies aux control register: 0x72360000 -> 0x72760000
L2C: DT/platform modifies aux control register: 0x72360000 -> 0x72760000
L2C-310 erratum 769419 enabled
L2C-310 enabling early BRESP for Cortex-A9
L2C-310 full line of zeros enabled for Cortex-A9
L2C-310 ID prefetch enabled, offset 1 lines
L2C-310 dynamic clock gating enabled, standby mode enabled
L2C-310 cache controller enabled, 8 ways, 512 kB
L2C-310: CACHE_ID 0x410000c8, AUX_CTRL 0x76760001
zynq_clock_init: clkc starts at f0802100
Zynq clock init
clocksource: ttc_clocksource: mask: 0xffff max_cycles: 0xffff, max_idle_ns:
537538477 ns
sched_clock: 16 bits at 54kHz, resolution 18432ns, wraps every 603975816ns
timer #0 at f080a000, irq=16
sched_clock: 64 bits at 333MHz, resolution 3ns, wraps every 4398046511103ns
clocksource: arm_global_timer: mask: 0xffffffffffffffff max_cycles: 0x4ce07af025,
max_idle_ns: 440795209040 ns
Switching to timer-based delay loop, resolution 3ns
Console: colour dummy device 80x30
Calibrating delay loop (skipped), value calculated using timer frequency.. 666.66
BogoMIPS (lpj=3333333)
pid_max: default: 32768 minimum: 301
Mount-cache hash table entries: 2048 (order: 1, 8192 bytes)
Mountpoint-cache hash table entries: 2048 (order: 1, 8192 bytes)
CPU: Testing write buffer coherency: ok
CPU0: thread -1, cpu 0, socket 0, mpidr 80000000
Setting up static identity map for 0x100000 - 0x100060
Hierarchical SRCU implementation.
smp: Bringing up secondary CPUs ...
CPU1: thread -1, cpu 1, socket 0, mpidr 80000001
smp: Brought up 1 node, 2 CPUs
SMP: Total of 2 processors activated (1333.33 BogoMIPS).
CPU: All CPU(s) started in SVC mode.
devtmpfs: initialized
VFP support v0.3: implementor 41 architecture 3 part 30 variant 9 rev 4
clocksource: jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns:
19112604462750000 ns
futex hash table entries: 512 (order: 3, 32768 bytes)
pinctrl core: initialized pinctrl subsystem
NET: Registered protocol family 16
DMA: preallocated 256 KiB pool for atomic coherent allocations
cpuidle: using governor menu
hw-breakpoint: found 5 (+1 reserved) breakpoint and 1 watchpoint registers.
hw-breakpoint: maximum watchpoint size is 4 bytes.
zynq-ocm f800c000.ocmc: ZYNQ OCM pool: 256 KiB @ 0xf0880000
zynq-pinctrl 700.pinctrl: zynq pinctrl initialized
e0000000.serial: ttyPS0 at MMIO 0xe0000000 (irq = 36, base_baud = 6249999) is a
xuartps
console [ttyPS0] enabled
vgaarb: loaded
SCSI subsystem initialized
usbcore: registered new interface driver usbfs
usbcore: registered new interface driver hub
usbcore: registered new device driver usb
media: Linux media interface: v0.10
Linux video capture interface: v2.00
pps_core: LinuxPPS API ver. 1 registered
pps_core: Software ver. 5.3.6 - Copyright 2005-2007 Rodolfo Giometti
<giometti@linux.it>
PTP clock support registered
EDAC MC: Ver: 3.0.0
```

```
FPGA manager framework
fpga-region fpga-full: FPGA Region probed
Advanced Linux Sound Architecture Driver Initialized.
clocksource: Switched to clocksource arm_global_timer
NET: Registered protocol family 2
TCP established hash table entries: 8192 (order: 3, 32768 bytes)
TCP bind hash table entries: 8192 (order: 4, 65536 bytes)
TCP: Hash tables configured (established 8192 bind 8192)
UDP hash table entries: 512 (order: 2, 16384 bytes)
UDP-Lite hash table entries: 512 (order: 2, 16384 bytes)
NET: Registered protocol family 1
RPC: Registered named UNIX socket transport module.
RPC: Registered udp transport module.
RPC: Registered tcp transport module.
RPC: Registered tcp NFSv4.1 backchannel transport module.
hw perfevents: no interrupt-affinity property for /pmu@f8891000, guessing.
hw perfevents: enabled with armv7_cortex_a9 PMU driver, 7 counters available
workingset: timestamp_bits=30 max_order=18 bucket_order=0
jffs2: version 2.2. (NAND) (SUMMARY)  ©2001-2006 Red Hat, Inc.
bounce: pool size: 64 pages
io scheduler noop registered
io scheduler deadline registered
io scheduler cfq registered (default)
io scheduler mq-deadline registered
io scheduler kyber registered
dma-pl330 f8003000.dmac: Loaded driver for PL330 DMAC-241330
dma-pl330 f8003000.dmac:        DBUFF-128x8bytes Num_Chans-8 Num_Peri-4
Num_Events-16
```

[Trenz board version: TE0715-04-30-1I3.]

[Final O/S version is not yet decide due to the lack of final PUT/Trenz boards.]

5.3. Custom Made Applications

5.3.1. Application: ad9361-config.run [TX + RX]

[`ad9361-config.run`] application is used to configure AD9361 and the internal IP Core. The program is launched automatically at the system startup. To an extent it might be re-launched during the normal operational state of the system, although such an on-the-fly-re-configuration **is strongly discouraged**, as it may result in a non-optimal system state (e.g. not every AD9361 and IP Core internals could be properly configured).

The configuration of [`ad9361-config.run`] application is kept in [`/etc/radio/current/ad9361-config.gflags`] file. Note, however, that on the TX side, the location of the configuration file might be changed due to the availability of the pre-boot/post-boot configuration update mechanism. This mechanism is realized by an appropriate software and it is transparent from the point of view of [`ad9361-config.run`] application.

The example of the configuration file mentioned in this section is shown below.

```
# Default configuration of AD9361

# Note that config lines cannot have comments!
```

```
### Cubesat specific configuration [BEGIN]

# Without this option nothing in this section
# is taken into account during configuration
--conf_cubesat

# Quadrature tracking
--quad_track=ON

# ENSM mode
# Only TX and RX are support.
# Anything else will lead to problems
# Values: TX, RX
--ensm_mode=TX

### Cubesat specific configuration [END]


### FPGA TRX config [BEGIN]

# Turn ON/OFF FPGA TRX (TDD)
--fpgatrx_enable=ON

# Select code rate
# Values: 0, 1, 2, 3, 4, 5, 6
--code_rate=0

# TX Data source
# Values: 0, 1, 2
--fpgatrx_tx_data_src=0

# Length of frequency offset estimation preamble
# Values: 0, 1, 2
--fpgatrx_frequency_offset_estimation_preamble_length=2

# Waiting time (in ms) before configuring / enabling FPGA TRX module
--fpgatrx_enable_wait_time_ms=950

# Wait time (in ms) after resetting the outer FPGA TRX IP CORE
# We wait only if --fpgatrx outer ipcore reset is present, i.e.,
# it is not uncommented.
--fpgatrx_wait_after_outer_ipcore_reset_ms=45

# Whether we do or do not reset the outer FPGA TRX IP CORE.
# Comment if you want to disable resetting.
--fpgatrx_outer_ipcore_reset


### FPGA TRX config [END]

# Direction
--direction=TRX

# FIR filter configuration
--fir_filter_file=/etc/radio/filters/cubesat-filter-v0004-30dot72.ftr
--fir_filter=ON


# Extra register content
# Format used is: reg1 << val1; reg2 << val2
# NO QUOTES AROUND REGISTERS!
# --extra registers content=0x035 << 0x0B


# Carrier frequency in GHz
# TX/RX filters on the small PUT radio boards have range 2120 -- 2170 [MHz]
--c_frq=2.145

# Bandwidth in MHz
#--bandwidth=15
--bandwidth=28

# Sampling rate in MSPS (mega samples per second)
#
#--sampling_rate=7.68
--sampling_rate=40.816326
```

```
# TX power gain in dB
--tx_power_gain=-25


# Logging capability
--log

# Disable ADI digital interface FIR tune
# (tuning must be disabled on picozed/ADRV1CRR-FMC)
--disable_digital_interface_tune_fir

# To simulate the bahaviour without setting any AD9361 config
# Uncomment the following line:
--dry-run
```

5.3.2. Application: config-dispatcher.run [TX only]

[`config-dipatcher.run`] is the application that is responsible for:

- Reading configuration from OBC (via exchanging `nanopb` messages on the UART interface) for dependent applications, which are
  - [`ad9361-config.run`] application, and
  - [`session-scheduler.run`] application.
- Translating this configuration to the format that is accept by the two applications mentioned in previous bullet points.
- Preparing custom configuration files that are then delivered to the dependent applications.

An example configuration file for this application is shown below.

```
# Default configuration for config-dispatcher.run


## UART Configuration [BEGIN]

# TTY device
--tty=/dev/ttyPS1

# TTY parity
--parity=NoParity

# TTY Duplex mode
--duplex=FullDuplex

# TTY speed (in bits per second)
--speed=115200

## UART Configuration [END]


# Time for which we wait to dispach configurations.
# This time is in only counted until we receive first
# configuration-related message. This is because the
# exact delay cannot be precisely computed; assuming
# the worst-case scenario the size of the configuration
# file must be treated as a random variable.
--uart_begin_config_wait_time_ms=1750


# Configuration maps:
# Note: All configuration maps **must** be specified in single line.
#       Semicolon at the end is optional
--config_maps=static_config: /etc/radio/current/ad9361-config.gflags > /var/cache/ad9361-
config.gflags; session_plan: /etc/radio/current/session-plan.yaml > /var/cache/session-
plan.yaml;



# Logging
--log
```

```
# To 'dry' run it uncomment the following line:
# --dry-run
```

5.3.3. Application session-scheduler.run [TX only]

[`session-scheduler.run`] application is responsible for applying on-the-fly changes to the AD9361 (if necessary and possible) and to the custom IP core delivered by PUT. This application consumes the session plan (see an example file below) and applies the configuration at specified epochs / times.

During the communications session, the following transmission parameters can be changed:

- code rate (entrance: [`fpgatrx/code_rate`]),
- TX power (entrance: [`ad9361/out_voltage0_hardwaregain`] – there is possibility to change TX power of only 0-th channel, since CBFR is equipped with only one antenna).

REMARK! Note also that the order of epochs in the session plan need not to be arranged in a time ordered manner. The required sorting would be realized by the software.

```
# Default configuration of satellite's session scheduler

## Session plan contains a list of 'epoch' and 'config' pairs
## that specify the time and exact configuration that is applied
## to the RF communication module (a.k.a. CBTM, C-Band Transmitter Module).
session_plan:
    ## Epoch ('epoch') is the time at which the configuration is applied.
    ## Its format is; YYYY-MM-DD HH:MM:SS
  - epoch: 2021-08-19 16:35:30
    ## Configuration ('config') contains a list of strings of the format:
    ## 'value' > 'configuration entity'
    ## Character '>' is mandatory!
    config:
      - 1 > fpgatrx/code_rate
      - -24.50 > ad9361/out voltage0 hardwaregain
  - epoch: 2021-08-19 13:55:55
    config:
      - 4 > fpgatrx/code rate
      - -25 > ad9361/out_voltage0_hardwaregain
  - epoch: 2021-08-19 13:56:30
    config:
      - 0 > fpgatrx/code rate
      - -26.50 > ad9361/out_voltage0_hardwaregain
```

Bibliography

[1] CS.S1.Gen System specification
[2] Zynq-7000 All Programmable SoC Overview (DS190)
[3] TE0715 Technical Reference Manual
[4] AD9364 RF Agile Transceiver Data Sheet
[5] HMC7357 Data Sheet

Appendix A

Schematics of the CBSR transmitter module.

Appendix B

Mechanical design of the CBSR transmitter module.

Note: full module documentation can be found in external *step* and *gerber* files.