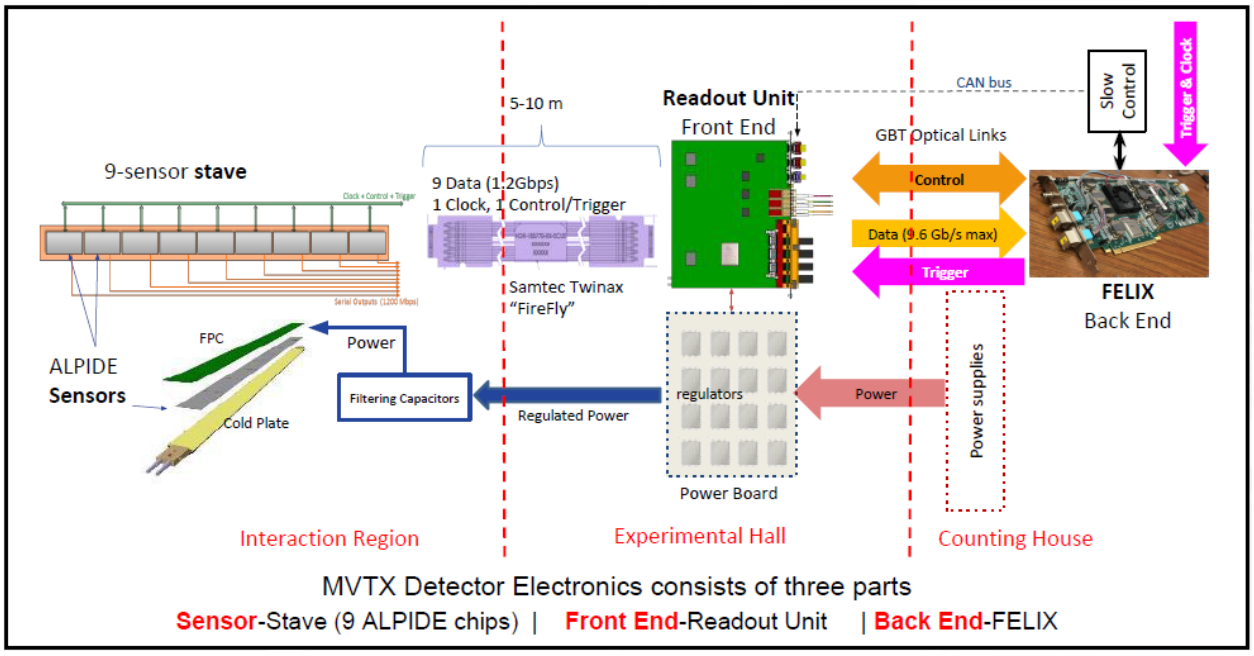
The Monolithic-Active-Pixel-Sensor-based vertex Detector (MVTX) Readout and Controls

the technical note for the MVTX readout and controls
August 26, 2018 (revision 1.0)



Proposing Organization: Los Alamos National Lab

Principal Investigator: Ming X. Liu

Phone: 505-412-7396
Email: mliu@lanl.gov

Collaborators: Mark Prokop, Alex Tkatchev, Sho Uemura, Kun Liu, Cesar da Silva, Pat McGaughey, Andi Klein, Xuan Li, Darren McGlinchey, Sanghoon Lim, Walt Sondheim, Hubert van Hecke

Authors: Alex Tkatchev
Email: alextkatchev17@gmail.com

Sho Uemura
Email: meeg@slac.stanford.edu

Todo list

Table of Contents

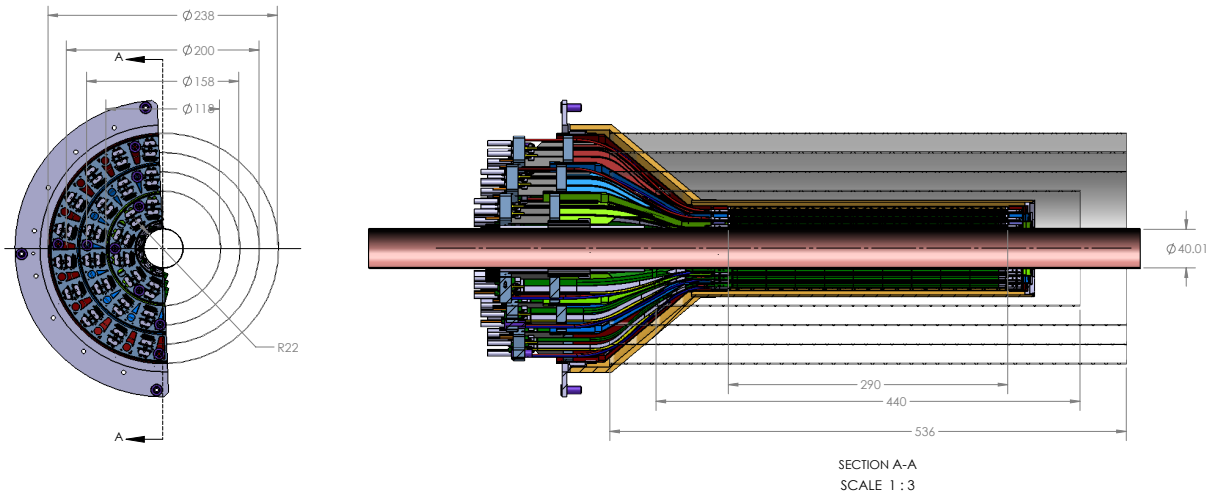


Figure 1: Layout of the Maps-based VerTeX detector (MVTX).

1 System Description

1.1 Description

The MVTX design leveraged the R&D performed for the inner layer ALICE/ITS detector described in the technical report [1]. The layout of the MVTX is shown in Figure 1. It consists of the three layers azimuthally segmented in units named staves, which are mechanically independent.

An overview for the whole electronic system is shown in Figure 2. The most fundamental element of the MVTX is the ALPIDE sensor which consists of the pixel matrix, analog frontend, digitization, data formatting, and data transmission integrated in the same chip (monolithic active pixel sensor). Details of the ALPIDE sensor is described in Sec.2.3. Nine ALPIDE sensors are wire bonded to a single Stave with 9 independent data lines and common control and clock.

The nine serialized data links from each Stave is sent to one Readout Unit (RU) trough a 5 meters copper twinax SAMTEC-FireFly cable described in Section 2.4.1. The RU also provides power configuration and monitoring for the Stave. The RU crate is placed outside the sPHENIX magnet area. Details of the RU are described ion Section 3.3. Each RU provides an optical connection to the back-end Front End Link Exchange (FELIX) unit, in the counting house to which the data streams and slow controls are sent trough GBT optical links.

The back-end unit adopted for MVTX is the FELIX board, it has 48 inputs and outputs capable to support GBT optical links over multimode fiber. The board also contains a 16-lane PCIe Gen 3.0 interface to the host server, and a Xilinx Knitex Ultrascale FPGA. One FELIX is expected to support 8 RUs, which

Table 1: Geometrical parameters of the MVTX.

	layer 0	layer 1	layer 1
Radial position (min) (mm)	22.4	30.1	37.8
Radial position (max) (mm)	26.7	34.6	42.1
Length (sensitive area) (mm)	271	271	271
Pseudo-rapidity coverage	$\pm$2.5	$\pm$2.3	$\pm$2.0
Active area (cm^2)	421	562	702
Pixel chip dimensions (mm^2)		15 $\times$ 30	
Number of chips	108	144	180
Number of Staves	12	16	20
Staves overlap in $r\phi$ (mm)	2.23	2.22	2.30
Gap between chips (μm)		100	
Chip dead area in $r\phi$ (mm)		2	
Pixel size (μm)		(20 - 30) $\times$ (20 - 30)	

will utilize half of the optical links. Each FELIX is installed in a host server. More details on the FELIX board can be found in Section 4.

The software running on the FELIX server reads data from FELIX over the PCIe bus, packages it in the sPHENIX data format Phenix Raw Data Format (PRDF), and transfers it to the sPHENIX DAQ. RCDAQ which is the sPHENIX DAQ system is described in Section 5.

The power distribution for the system is performed by the power board, described in Section 2.5. Each Stave is powered by one power board. The Power Boards are supplied by one CAEN bulk power supply. A detail description of the control system is found in Section 7.1.

Details on the signal timing and trigger is described in Section 6.

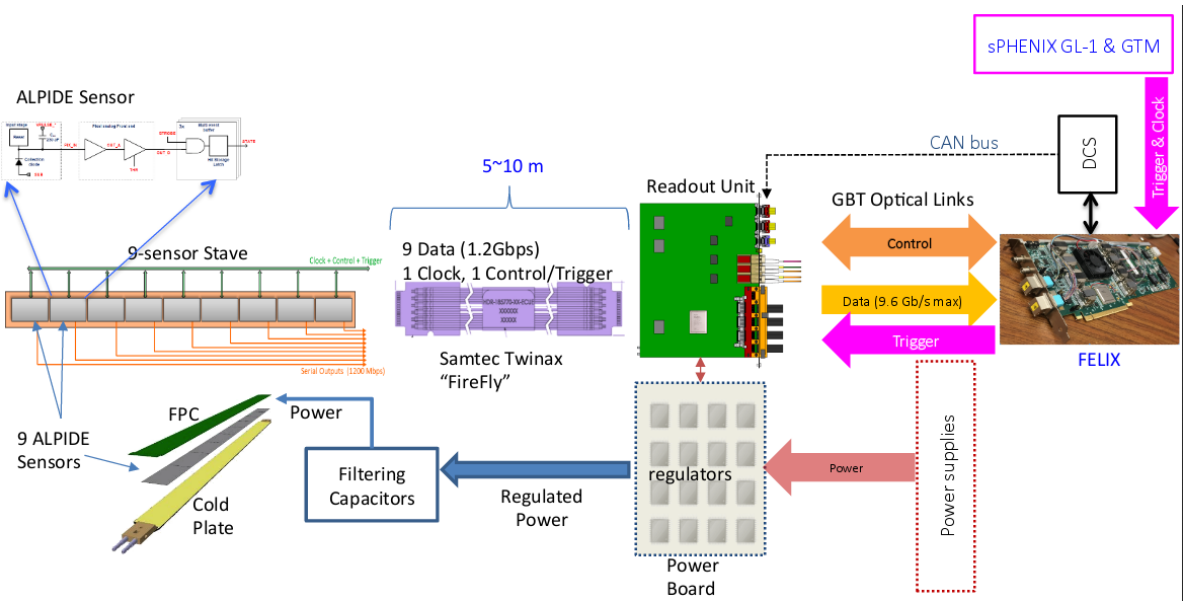


Figure 2: Block diagram of the MVTX readout electronics.

2 Sensor

2.1 Detector Description

The Stave consists of 9 ALPIDE sensors which takes in a common clock and control line and outputs 9 gigabit data streams. Each ALPIDE sensor consists of a matrix of 512 pixel rows by 1024 pixel columns. Each pixel contains an amplifier discriminator and 3 hit storage multi event buffer. The data output rate is configurable. Zero suppression is done on chip. Readout modes can be configured to triggered or continues mode. The device can be configured for internal pulse injection mode to pulse specific pixels, typically used for testing. PRBS data generation for cable verification. As well as pixel masking functionality in the event a pixel is damaged or remains high. MVTX will use triggered mode at 1.2 Gigs.

2.2 Stave Description

2.3 ALPIDE Description

The ALPIDE chip is based on the Monolithic Active Pixel technology implemented in a 180 nm CMOS process fabricated by Tower Jazz. The active pixel region for the ALPIDE sensor is 30mm (X direction) by 15mm (Y direction) which contains 1024×512 (X×Y) sensitive pixels, see Figures 4 and 3. The dimension of each pixel is 29.24μm × 26.88 μm (X×Y). The pixel columns are counted from 0 to 1023 from left to right along the X axis, and the pixel rows are counted from 0 to 511 from the top to the bottom. The periphery circuit is located in the bottom of the chip within the 1.2×30 mm^2 region.

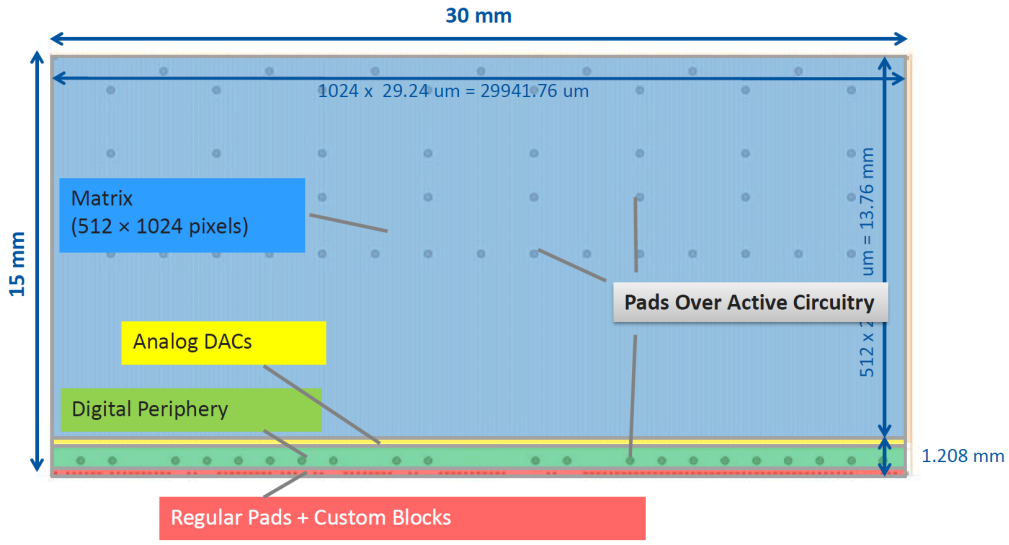


Figure 3: The ALPIDE sensor Floor Plan.

The ALPIDE design is based on a charge sensitive amplifier, amplifier with capacitive feedback. The front-end and discriminator act as an analogue delay line. The readout block diagram is shown in Figure 5 and the schematics is shown in Figure12. The collection n-well has octagonal shape with 2 μm diameter and n-well to p-well spacing of 3 μm. In Figure12, diode D1 is the sensor p-n junction. The input node is continuously reset by diode D0. VRESETD establishes the reset voltage of the charge collecting node (pix_in). A particle hit will lower the potential at the pixel input pix_in by a few tens of mV. This will cause the source follower formed by M1 and the current source M0 to force the source node to follow this voltage excursion and dump charge associated with the voltage change and the capacitance of the source node onto

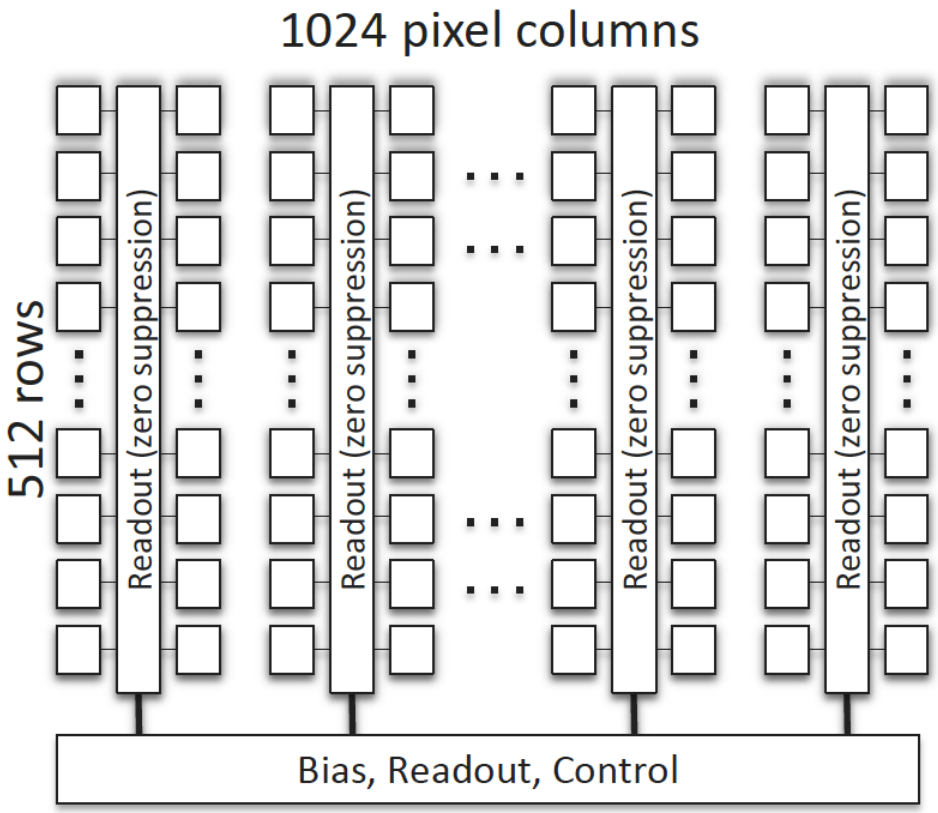


Figure 4: The ALPIDE chip architecture.

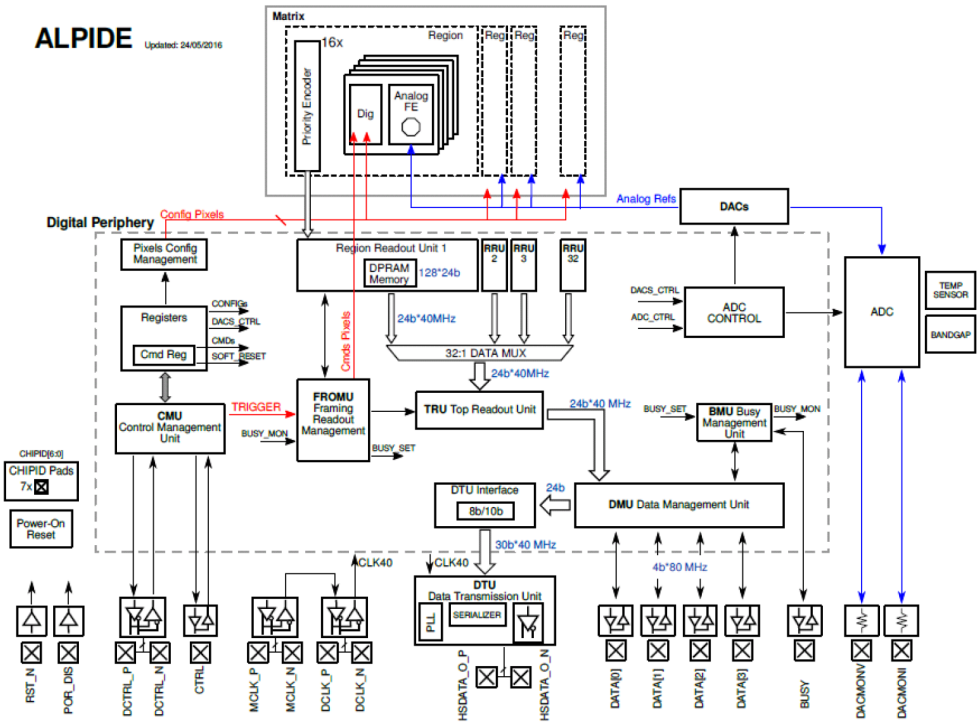


Figure 5: The ALPIDE sensor block diagram.

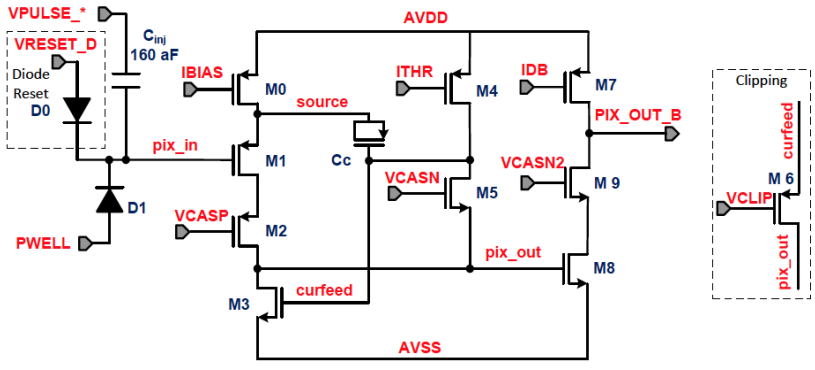


Figure 6: The ALPIDE front end schematics.

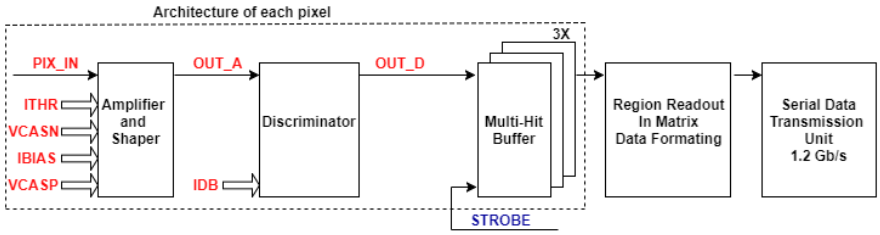


Figure 7: The ALPIDE block diagram 2.

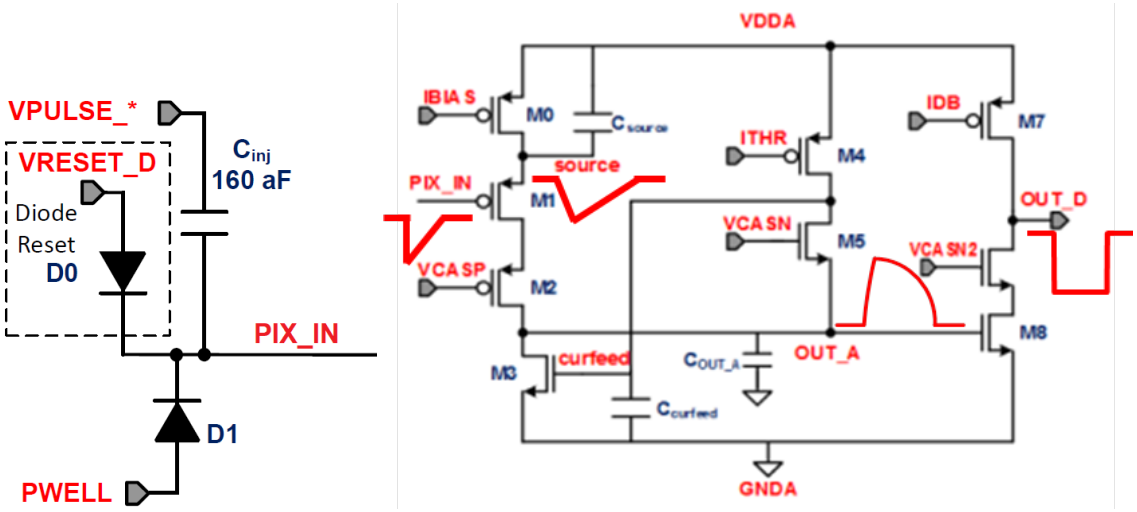


Figure 8: The ALPIDE front end schematics.

the analog output node pix_out[2]. The charge threshold of the pixel is defined by ITHR, VCASN and IDB. The effective charge threshold is increased by increasing ITHR or IDB. It is decreased by augmenting VCASN. Voltage bias VCLIP controls the gate of the clipping transistor M6. The lower VCLIP, the sooner the clipping will set in.

The output of the ALPIDE front-end has a peaking time of around 2μs within the typical 5μs duration. This allows operating the chip in triggered mode when the latency of the incoming trigger is comparable with the front-end peaking time. A common threshold is applied to all pixels. The latching of the discriminated hits is controlled by global STROBE pulse. The STROBE pulse can be generated from either external trigger or internal trigger, the duration of the STROBE is programmable. Each pixel contains a pulse injection capacitor for the test charge injection in the input of the front-end. A digital-only pulsing mode can be selected to write the logical signal in the pixel memory cells. The pulsing pattern is programmable. The readout of the pixel matrix is based on a circuit named Priority Encoder. The transfer of the matrix data to the periphery circuit is zero-suppressed. The operation voltages are summarized in Table 2.

Each pixel in the ALPIDE matrix has 3 in-pixel data storage elements (buffer). The Multi Event Buffer (MEB) enables the storage of 3 complete frames without the completion of a matrix readout or any data loss.

Table 2: The operation voltage for the MVTX chip/stave.

	Minimum value	Default value	Maximum value
AVSS (Analog ground)	0 (V)	0 (V)	0 (V)
AVDD (Analog supply)	1.62 (V)	1.8 (V)	1.98 (V)
DVSS (Analog ground)	0 (V)	0 (V)	0 (V)
DVDD (Analog supply)	1.62 (V)	1.8 (V)	1.98 (V)
PWELL (Bias Voltage)	-6 (V)	TBD	0 (V)
SUB (Bias Voltage)	-6 (V)	TBD	0 (V)

The data from the 32 region readout blocks are assembled and formatted by a Top Readout Unit module. There are two readout modes: one is the "Triggered" mode, the other is the "Continuous" mode. The "Triggered" mode defines as the strobe generation and readout are triggered externally. The "Continuous" mode refers to the frames are continuously integrated and readout with programmable strobe duration.

2.3.1 Pulse Injection and Masking

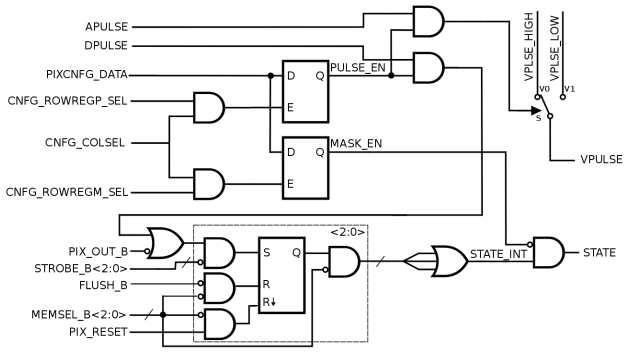


Figure 9: The ALPIDE pixel logic.

The pixels of the ALPIDE chip contain built-in testing features. They can be used to force the generation of a hit using a test charge injection capacitor (analog pulsing), directly setting the pixel state register (digital pulsing), or disabling readout of the pixel (masking). Details on the pixel digital section including pulsing are shown in Figure 9. Each pixel contains two read-only registers: MASK_EN and PULSE_EN. The Mask Enable (MASK_EN) register disables readout of the pixel so the pixel will not report a hit under any circumstance (real particle, analog pulse, or digital pulse). The Pulse Enable (PULSE_EN) register enables the pulsing functionality of the pixel; this register must be enabled for the pixel to respond to APULSE or DPULSE signals. The MASK_EN and PULSE_EN registers for all pixels are controlled through special register writes; individual pixels, or arbitrary patterns of pixels, can be masked or pulsed.

Once the pulsing has been enabled in the desired pixels, two pulsing signals (APULSE and DPULSE) are used to trigger the injection of the charge in the front-end or to set the state register. The edges of the APULSE signal cause the injection of current pulses in the front-end input node pix_in. The polarity of the pulse depends on the direction of the edge (rising or falling). The rising edge simulates the effect of release of charge in the collection diode by a particle hitting the pixel. The total charge of the current pulse is controllable by dedicated DACs. The DPULSE signal can be used to set the pixel state latches that constitute the Multi-Event buffer. Both pulsing operations require that the PULSE_EN latches of the target pixels are set and that one of the three global STROBE signals is asserted, in order to store the hit in the corresponding MEB location. The masking of the pixels has priority over the pulsing, therefore a masked pixel does not produce a hit when it is pulsed.

2.3.2 Matrix Readout and Priority Encoder

The readout of pixel hit data from the matrix is based on a circuit named Priority Encoder, depicted in Figure 10. There are 512 instances of this circuit, one every two pixel columns. The Priority Encoder provides to the periphery the address of the first pixel with a hit in its double column, selecting it according to a hardwired topological priority. During one hit transfer cycle a pixel with a hit is selected, its address is generated and transmitted to the periphery and finally the in-pixel memory element is reset. The address of the next pixel with a hit in the double column is then calculated. This cycle is repeated until the addresses of all pixels initially presenting a valid hit at the inputs of a Priority Encoder have been transmitted to the periphery and all the pixel state registers have been reset. The transfer of the frame data from the matrix to the periphery is therefore zero-suppressed.

Each Priority Encoder is a fully combinatorial circuit and it is steered by sequential logic in the periphery during the readout of a matrix frame. It is implemented in a very narrow region between the pixels, extending vertically over the full height of the columns: see Figure 11. There is no free running clock distributed in the matrix and there is no signaling activity if there are no hits to read out. The average energy needed to encode the address of a hit pixel is of the order of 100 pJ. Power is consumed proportionally to the readout rate and to the frame occupancy. The Priority Encoders also implement the buffering and distribution of readout and configuration signals to the pixels.

2.3.3 Triggering and Timing

For the MVTX operation at the sPHENIX experiment, we foresee to use the trigger mode as sPHENIX data taking will be mainly in triggered mode(maximum rate = 15kHz). The collection of pixel states (reffed as frame) will be generated at a particular time and transmitted off chip following the reception of a trigger. The distribution of the MVTX timing and trigger is shown in Figure 13. The sPHENIX level-1 (LV1) trigger latency is around 4 μs. This leads to the study of the trigger delay time and ALPIDE pulse shaping time tuning to check any impacts on the efficiency. This trigger latency also raises the question of the pile up effects on the MVTX track efficiency at sPHENIX especially in the several MHz p+p collisions.

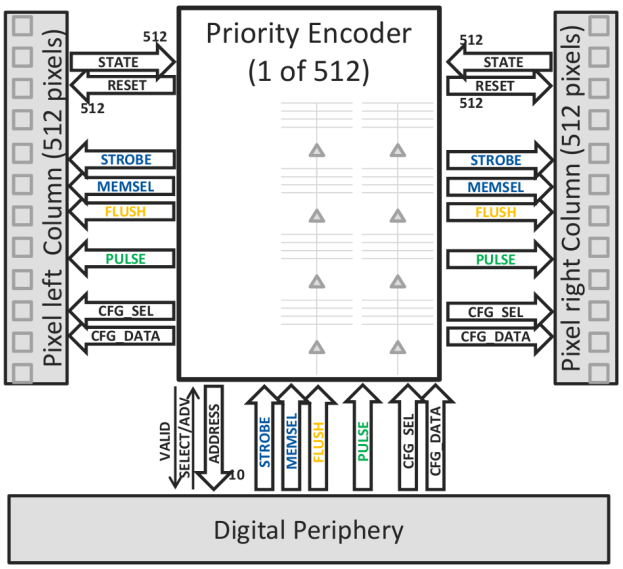


Figure 10: The ALPIDE priority encoder.

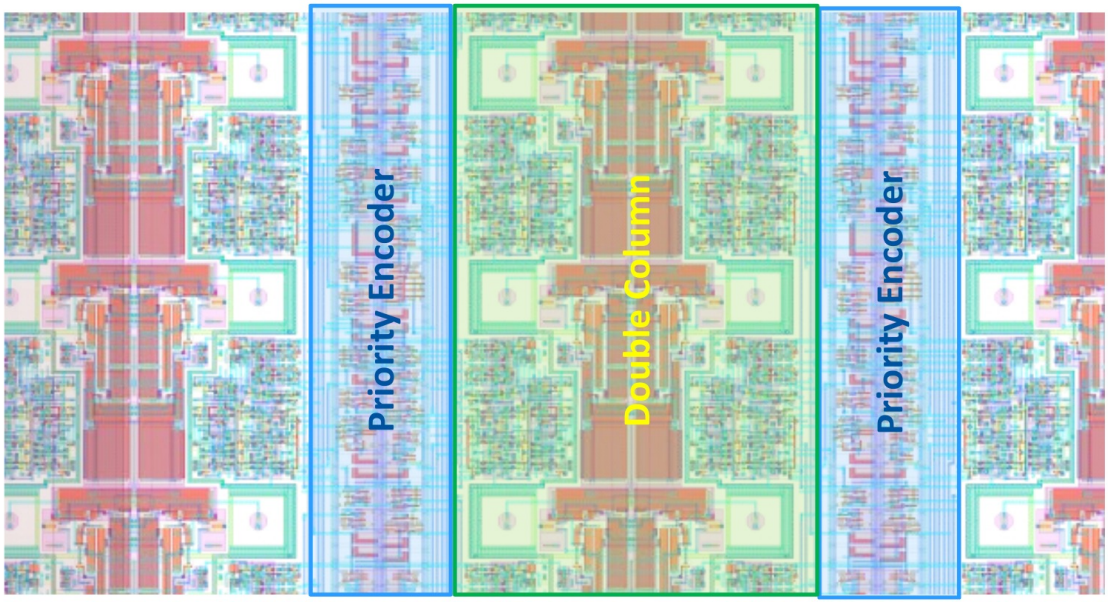


Figure 11: The layout of the pixel matrix, showing the double columns and priority encoders.

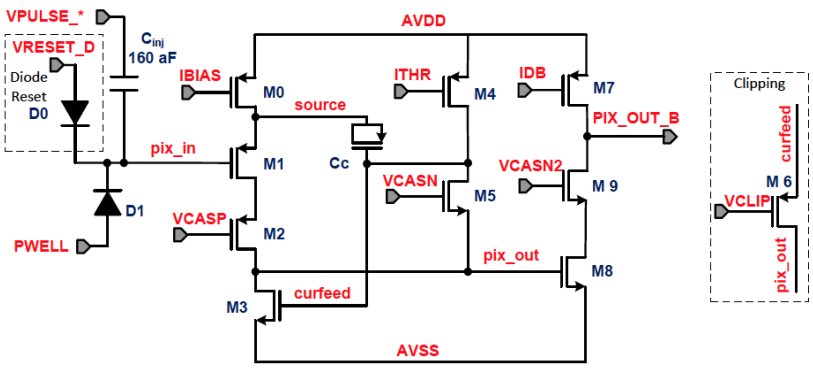


Figure 12: The ALPIDE waveforms.

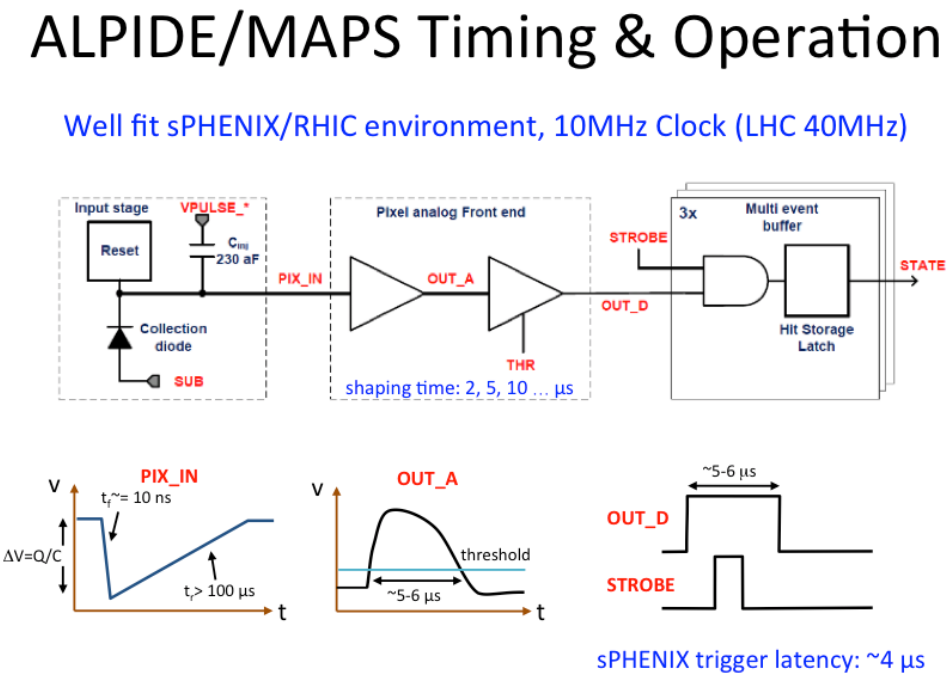


Figure 13: Block diagram of the ALPIDE pixel timing and trigger.

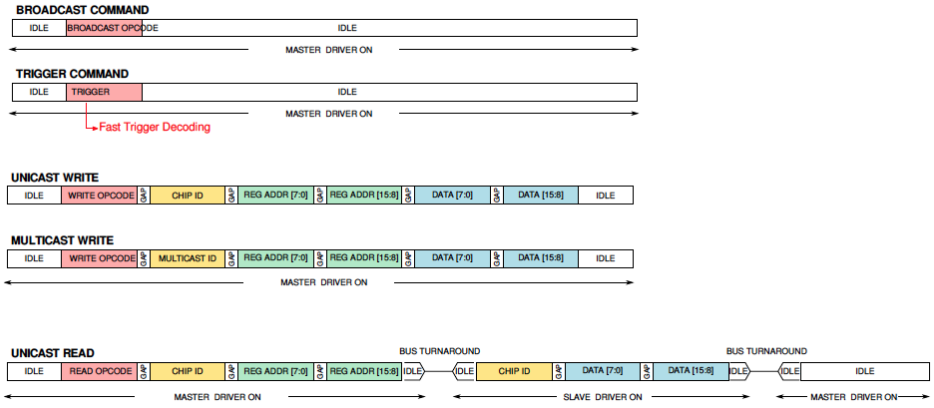


Figure 14: The format of the ALPIDE chip transaction on the control bus.

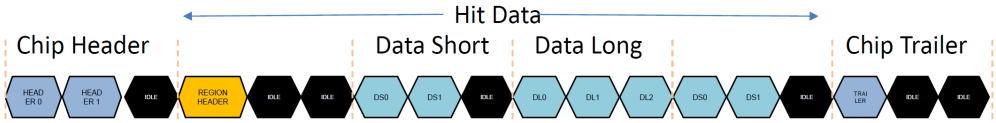


Figure 15: The format of the ALPIDE chip transaction on the control bus.

2.3.4 Data Transmission Unit (DTU)

The Data Transmission Unit (DTU) provides a fast serial link for the transmission of the data from the ALPIDE sensor. For MVTX, each sensor's DTU transmits data over a differential serial line with a line rate of 1.2 Gb/s to the off-detector electronics (Readout Unit). The data stream is transmitted over an aluminum over polyimide FPC (Flexible Printed Circuit) and then to a micro-twinaxial cable.

The DTU consists of two modules: a DTU Logic module, which interfaces the ALPIDE's Data Management Unit to the DTU, and the Data Transmission Unit itself, which contains the serializer, the transmission clock PLL, and the LVDS drivers. The DTU Logic module has the following functions:

- 8b/10b encoding of the data produced by the Data Management Unit and to be transmitted by the DTU
- Programming of the serial port line rate (1200 Mb/s, 600 Mb/s, 400 Mb/s) according to operating mode and configuration
- Monitoring of the PLL lock status and re-synchronization of the serializer
- Test features for the Data Transmission Unit: bypass of 8b10b encoding, PRBS-7 pseudorandom pattern, or static test pattern

The Data Transmission Unit and its Serializer are agnostic of the programmed output line rate. They always operate in the same fashion, shifting out a 30 bits data word loaded into the serializer at every cycle of the main digital clock (40 MHz). It is the DTU LOGIC that generates the 30 bits of the DTU DATA bus to obtain the desired output rate, effectively generating slower bit serial stream at the DTU output. For the

operation at 1200 Mb/s, all bits of the DTU DATA bus can be different. When transmitting at 600 Mb/s, every bit is replicated and transmitted twice, or equivalently for two consecutive bit Unit Intervals. Finally, every bit is replicated three times when operating at 400 Mb/s.

The DTU interface is designed to have an output signal compatible with the LVDS standard (at least for the voltage swing; the common mode voltage is reduced to 0.9 V). Resistive termination is required. A 600 MHz transmission clock is provided by an on-chip PLL.

The DTU architecture, shown in Figure 16, is based on a Double Data Rate (DDR) transmission scheme. The DTU Logic module generates a 30-bit word every 40 MHz clock cycle, which is loaded into two 15-bit shift registers. The 1.2 Gb/s line rate, combined with the 8b10b gives a data rate of 960 Mb/s for the MVTX configuration, or 3 bytes per 40 MHz clock cycle. The two shift registers are synchronized on the two 600 MHz clock edges and drive the line driver after a single-ended to differential conversion. A secondary path, equal to the main one but with two extra delay latches drives a second driver in order to provide pre-emphasis.

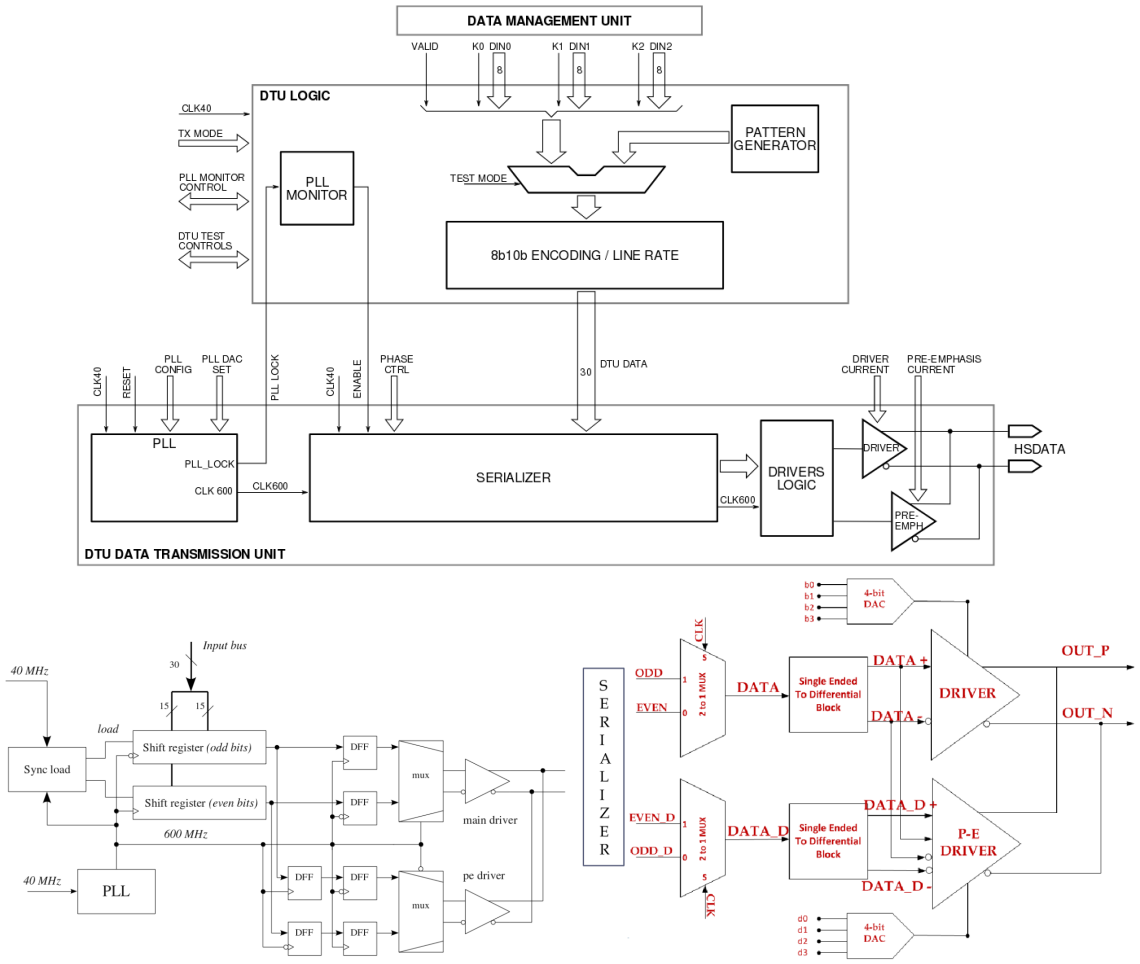


Figure 16: ALPIDE Data Transmission Unit (DTU).

The LVDS driver provides a current between 0 and 5 mA with a 0.312 mA resolution over a 100 ω differential cable. The driver is compatible with both commercial LVDS receivers and the GBTX SLVS

receivers. A pre-emphasis current of up to 50% (0 to 2.5 mA) has been implemented in order to be able to compensate for excessive RC on the cable. The pre-emphasis time width is one bit period, i.e. the current bit is emphasized if different from the previous one. The driver and pre-emphasis currents are controlled with 4-bit DACs. The output common mode has been set to 0.9 V (i.e. lower than the 1.2 V from the LVDS standard) in order to reduce the power consumption and have a better match with the 1.8 V supply voltage.

The ALPIDE line rate can be switched between 400 Mb/s, 800 Mb/s, and 1200 Mb/s. This is implemented by repeating the

2.3.5 Charge Collection

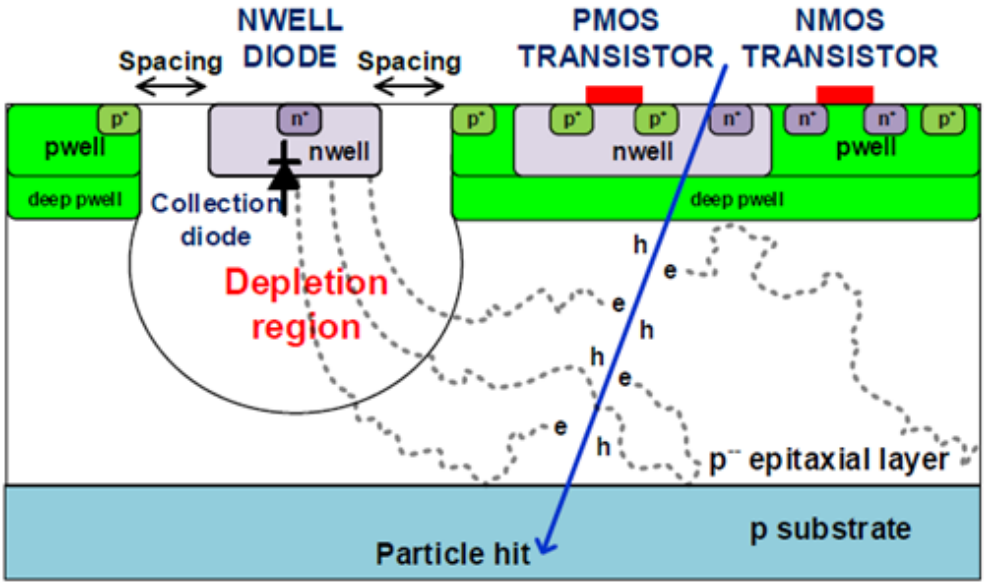


Figure 17: Depletion Region

When a charged particle passes through the silicon of the ALPIDE sensor, the energy lost by the particle is converted to electron-hole pairs. The signal detected by the ALPIDE consists of free electrons created in the epitaxial layer of the sensor and collected in a collection diode. The epitaxial layer is a layer of high-resistivity, weakly p-type doped silicon, and is bounded below by a p-type substrate and above by a deep p-well. Since both the substrate and p-well are more heavily doped than the epitaxial layer, there is a potential difference that prevents electrons from traveling from the epitaxial region to these others. Except for the collection diode, all of the in-pixel circuitry is contained inside the deep p-well so it cannot interfere with the collection of electrons from the epitaxial layer.

The collection diode is an octagonal n-well implanted in the epitaxial silicon through an opening in the deep p-well: see Figure 20. The $p-n$ junction between the n-well and the epitaxial silicon results in a depletion region devoid of holes in the epitaxial layer, where the concentration gradient of holes (which tends to push holes towards the epitaxial layer) overcomes the electric field created by the charge imbalance in the depletion region (which tends to force holes towards the n-well). The high electric field in the depletion region drifts any electrons inside the depletion region rapidly to the collection diode n-well.

Electrons created in the epitaxial layer cannot enter the substrate or p-well regions, so they diffuse in the low electric field until they hit a depletion region. This diffusion process is slow and random, so electrons can end up in collection diodes far from where the electron-hole pair was created. Applying a reverse bias to

the collection diode by applying a negative voltage to the substrate enlarges the depletion region, increasing the probability that an electron is collected quickly.

The size of the signal created in the ALPIDE is based on the number of electron-hole pairs created in the epitaxial layer. The epitaxial layer is 25 μm thick; as shown in Figure 18 the rate of energy loss by a minimum ionizing particle varies but has a mean value of 388 eV/μm and (for this thickness) a most probable value around 210 eV/μm. The number of electron-hole pairs created is related to the energy deposited by the ionization energy, which for silicon is 3.67 eV. Therefore the mean number of electron-hole pairs is 25 μm $\times$ 388 eV/μm/3.67 eV = 2600 electrons, and the most probable number is 25 μm $\times$ 388 eV/μm/3.67 eV = 1400 electrons.

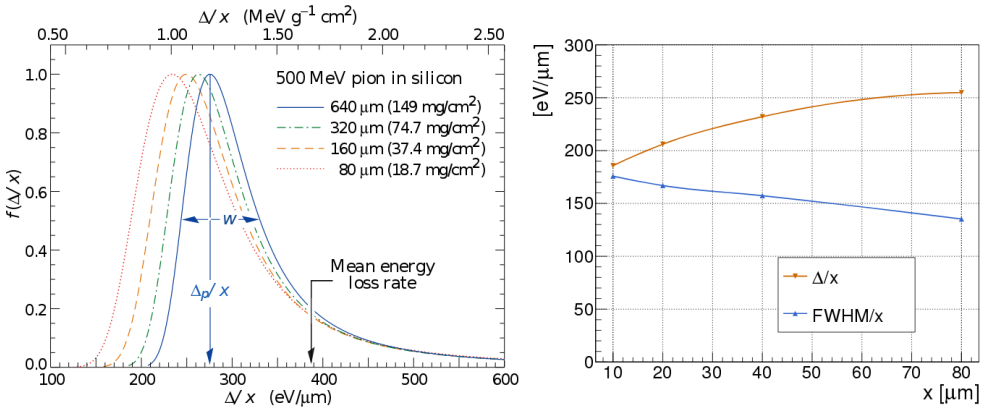


Figure 18: Energy deposition by minimum-ionizing particles in thin silicon.

2.3.6 Simulation and threshold characterization

The purpose of this section is to convey a circuit description of the ALPIDE front end, as well as simulate and characterize its bias and threshold parameters IBIAS, ITHR, IDB, VCASP, VCASN, and VCASN2. The effort began by reaching out to Tower Jazz, special thank to Vivian Gin and obtaining the respective PDK (Process Design Kit). A PDK is a model of the devices within a certain technology variation for their processes, typically provided by the foundry to an ASIC design team. After simulating the schematic is Figure8 and presenting it to Thanushan Kugathasan of the ALPIDE design team at CERN, additional suggestions and guidance was provided for a more accurate simulation. The below section could not have been have been completed without help from Vivian Gin and Thanushan Kugathasan our team is forever grateful for their time. This section is divided into three parts Part 1: ALPIDE front end schematic description. Part 2: ALPIDE simulation results and conclusion Part 3: ALPIDE laser test stand results.

The architecture of each ALPIDE pixel contains a sensing diode, amplifier and shaper stage, discriminator stage, and a digital section consisting of three hit storage registers (Multi Event Buffer). The latching of the discriminated signal above a threshold by the multi-hit buffer is controlled by three global STROBE signal. Once the signal is latched the Address- Encoder Reset-Decoder (AERD) provides the ADDRESS of the hit pixel and whether the current address is VALID to the chip periphery containing the DTU (Data Transmission Unit). The data is then sent off chip at 1.2Gbps (Inner Barrel). The description below will reference the three stages and the input signals of each stage PIX_IN, OUT_A, and OUT_D.

The voltage at the input node PIX_IN drops within the charge-collection time below 10 ns. The reset mechanism lets PIX_IN return to the baseline voltage VBASELINE in about 100 us. The signal amplified

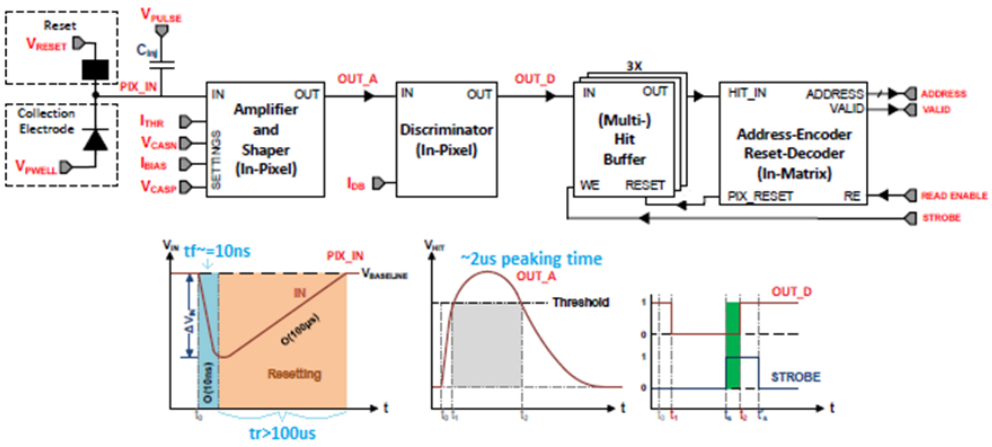


Figure 19: Simulation ALPIDE Reference.

by the front-end that acts as a delay line with a peaking time of around 2 us, the discriminated pulse pulse has an approximate duration of less than 10 us.

The reset is provided either by a diode or a PMOS. The PMOS-reset is a constant-current reset mechanism which provides the current IRESET when VIN deviates from the baseline value. The reset current IRESET can be adjusted to achieve the desired reset time and to adapt to the leakage current of the pixel. The baseline of the pixel can be regulated using VRESET. The diode reset is based on a single forward-biased diode.

The amplifier and shaper consist of two current branches, IBIAS (transistor M0) and ITHR (transistor M4). It receives the input signal PIX_IN and manipulates it in accordance with the IBIAS, ITHR, VCASP, VCASN, and bias and threshold parameters then outputs the OUT_A signal to the discriminator stage. The discriminator stage discriminates and inverts the analog input OUT_A to a digital active low output pulse OUT_D in accordance with the IDB threshold parameter.

Without external stimulation the input transistor M1 conducts the current IBIAS provided by the current source M0. The transistor M5 conducts ITHR current provided by the current source transistor M4, leading to a fixed voltage difference on OUT_A and VCASN. VCASN is used to define the baseline voltage of OUT_A after a particle hit. The cascode transistor M2 transistor circumvents the Miller effect for the input transistor.

Transistors M4 and M5 are used to generate a low frequency feedback to adjust conductivity since all transistors operate in the weak inversions region. The curfeed net loaded with Ccurfeed capacitance is connected to gate of transistor M3. The transistor M3 absorbs IBIAS+ITHR current. Csource and Ccurfeed are combined in one capacitance Cs as shown in figure Figure 21 C.

The source of M1 is coupled to the current-feedback node Ccurfeed . The capacitance of M1 input is much larger than the capacitance of OUT_A. Therefore, the concept is based on charge transfer from a large capacitance to a small capacitance to generate a voltage gain. When PIX_IN receives a negative input pulse, the voltage drop at the gate of M1 increases its current which flows between M1 and OUT_A. With some delay Vsource node and Vcurfeed follow PIX_IN, leading to reduced conductivity of M3 and OUT_A increases. The increase in OUT_A allows M5 to close. Therefore, ITHR charges Cs leading to an increase in Vcurfeed and increases conductivity of M3.

The transistor M6 serves as a clipping mechanism once OUT_A exceeds Vcurfeed and as a pulse dura-

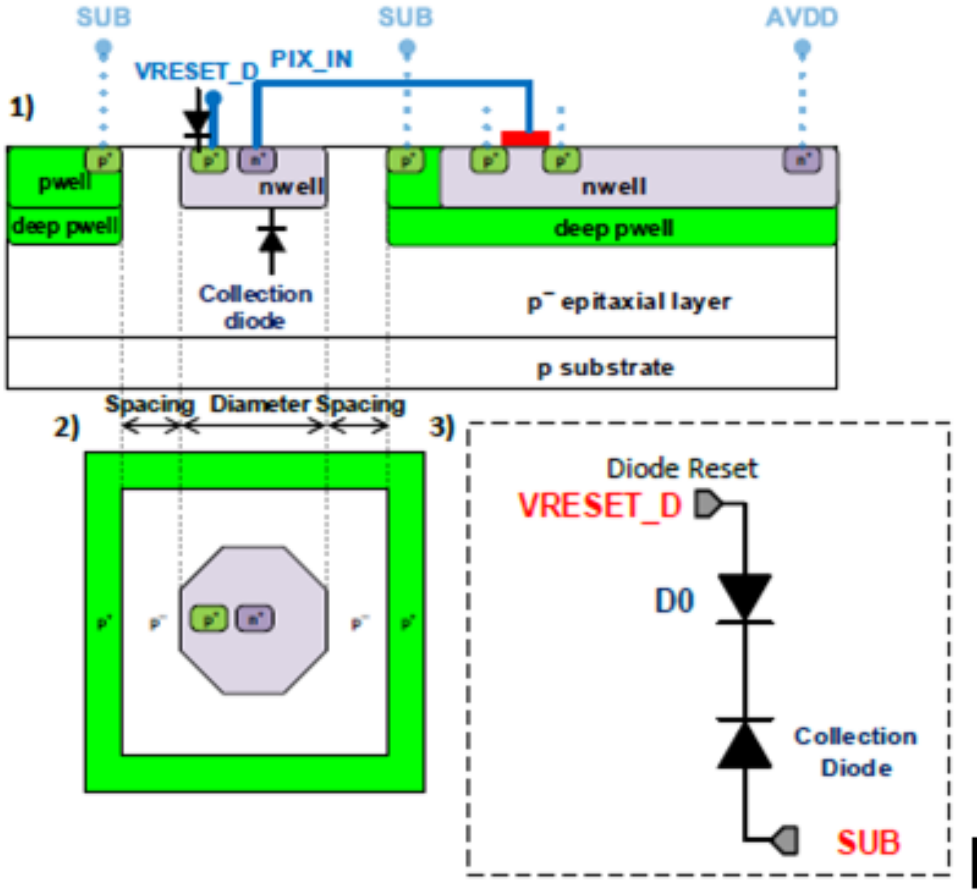


Figure 20: Diode Reset.

tion compressor. Transistor M6 in a diode connection: source and gate are connected to the curfeed node and the drain is connected to the OUT_A node. Normally M6 is reverse biased, but when the OUT_A signal is high enough to forward bias it, M6 provides additional discharge current to compress the pulse duration. The clipping effect is noticeable for input charges larger than 1.4 times the charge threshold both for OUT_A and OUT_D. The rise time of the pulse is defined by the time ITHR and IBIAS need to charge the OUT_A node. Additional charge stored at the source node speeds up process. The pulse duration depends on the clipping point and how quickly CURFEED is charged up to increase the conductivity of M3.

Voltage bias VCLIP controls the gate of the clipping transistor M6. The lower VCLIP, the sooner the clipping will set in.

Transistors M7 and M8 form the second stage which discriminates and inverts the OUT_A signal. In static operations the output current is almost zero however once OUT_A is present it increases the transconductance of M8 and the node OUT_D is discharged. The cascode transistor M9 shown in figure 21 was added to reduce the equivalent Miller capacitance on pix out.

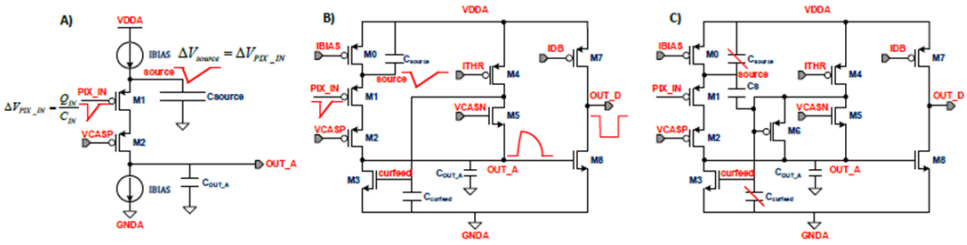


Figure 21: ALPIDE Front End Schematic Principle Practical Implementation Presented Circuit

Figure 25 shows C_in=1.6 fF. 5e3 elementary charge / 1.6e-15 farad = 0.5 V.

Transistors M0, M4, and M7 have been replaced with variable current sources as shown in figure 26 which are controlled in maestro tab using ADE Explorer (virtuoso simulation environment). The voltage parameters VCASP, VCASN and VCASN2 are also variable and controlled in the maestro tab. figure 27 shows the initial simulation schematic following the reference.

The initial simulation schematic was later modified to obtain a more accurate simulation result. The current sources were replaced with current mirrors. The reset circuit was added and the cc capacitor was more accurately modeled with the transistor array shown in 28.

Figure 29 shows the default threshold parameters as shown in the ALPIDE Manual and 30 shows OUT_A and OUT_D using the defaults. Please observe the bias and threshold parameters appear to the left of every waveform displayed.

The waveforms in 30 show OUT_A and OUT_D with 750 electrons, 1550 electrons, and 5000 electrons at the input. The figures in 31 and 32 show variation of the threshold parameters showing a change in OUT_A and OUT_D. Please observe the bias and threshold parameters appear to the left of every waveform displayed.

2.4 Interfaces

The ALPIDE chip has custom control interfaces. The differential control port (DCTRL) supports bi-directional serial signaling at 40 Mb/s. A second single ended control line (CNTRL) is also available but is not used for MVTX. The nine sensors on the MVTX stave are directly connected to a shared control differential line using the DCTRL port. The Control Management Unit block implements the control layer

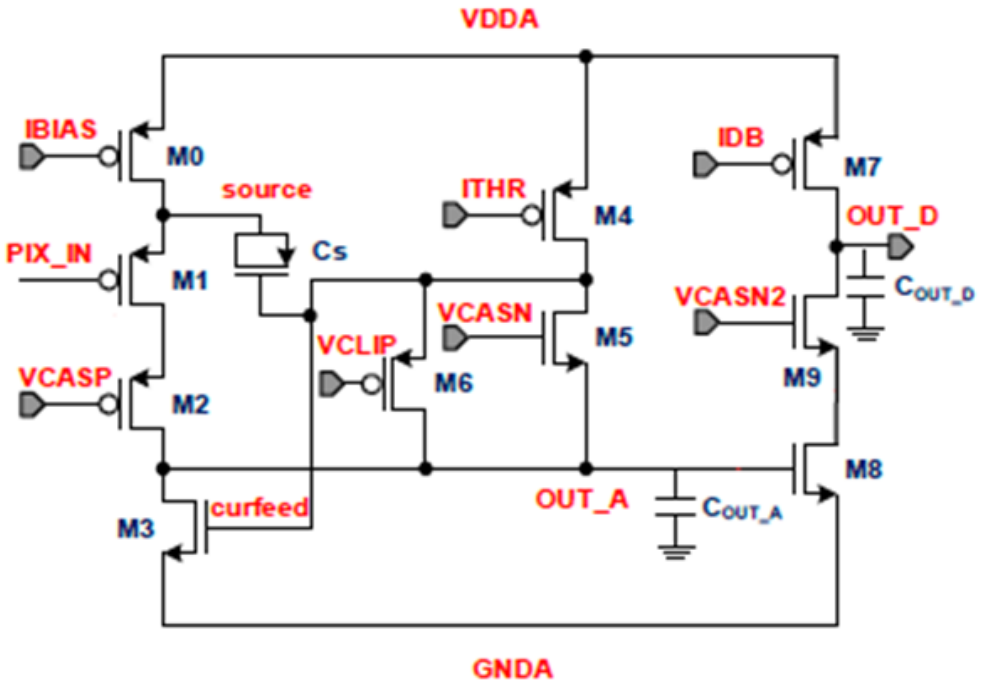


Figure 22: ALPIDE Optimized Schematic

	M0	M1	M2	M3	M4	M5	M6	M7	M8	M9	Cs
W/L [um/um]	1.8/8.5	0.92/0.18	0.22/0.18	0.5/5	2/8.4	0.5/10	0.5/3	0.42/7	0.22/4	0.42/0.2	344 fF

Figure 23: Transistor Sizes Table

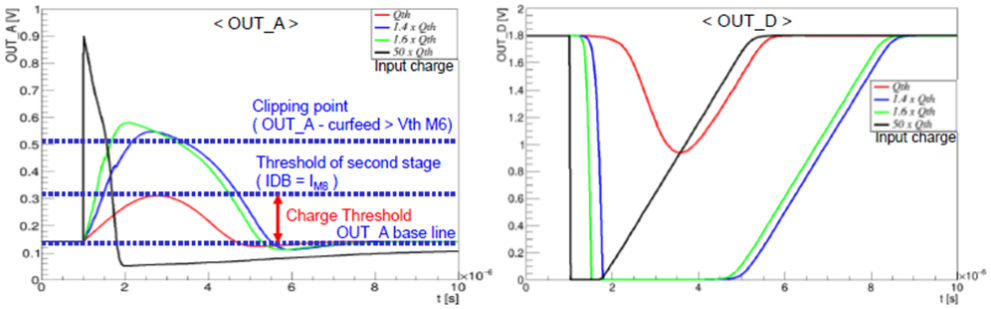


Figure 24: ALPIDE Transient plots OUT_A OUT_D from Presentations and papers

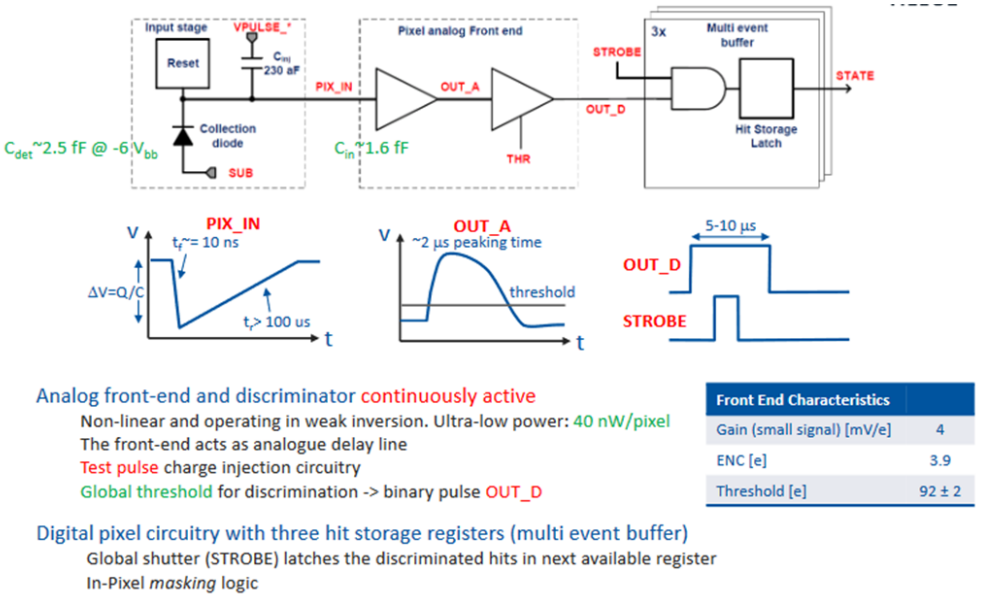


Figure 25: ALPIDE Capacitance

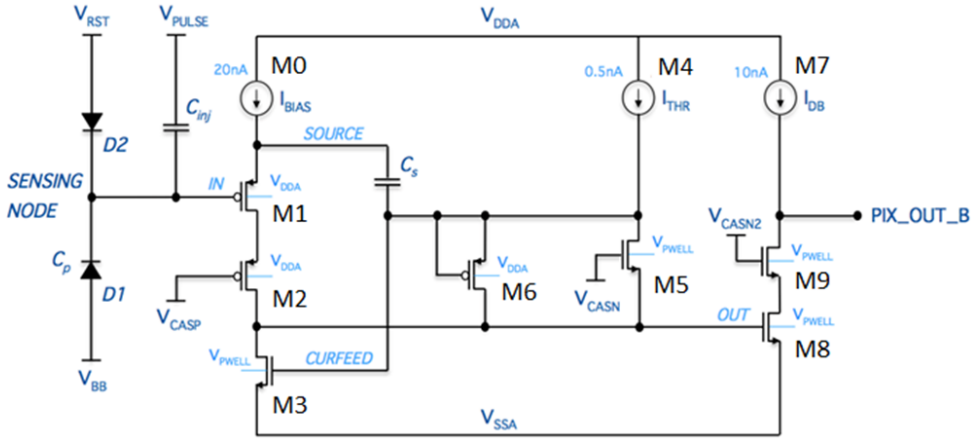


Figure 26: ALPIDE Simulation Schematic reference.PNG

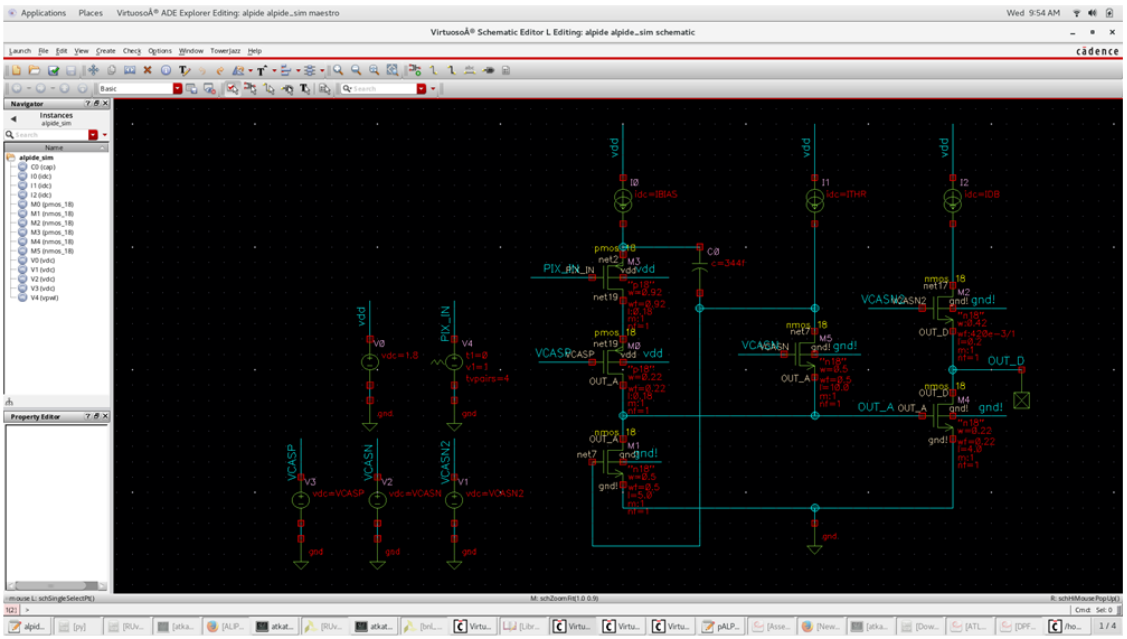


Figure 27: ALPIDE Simulation initial

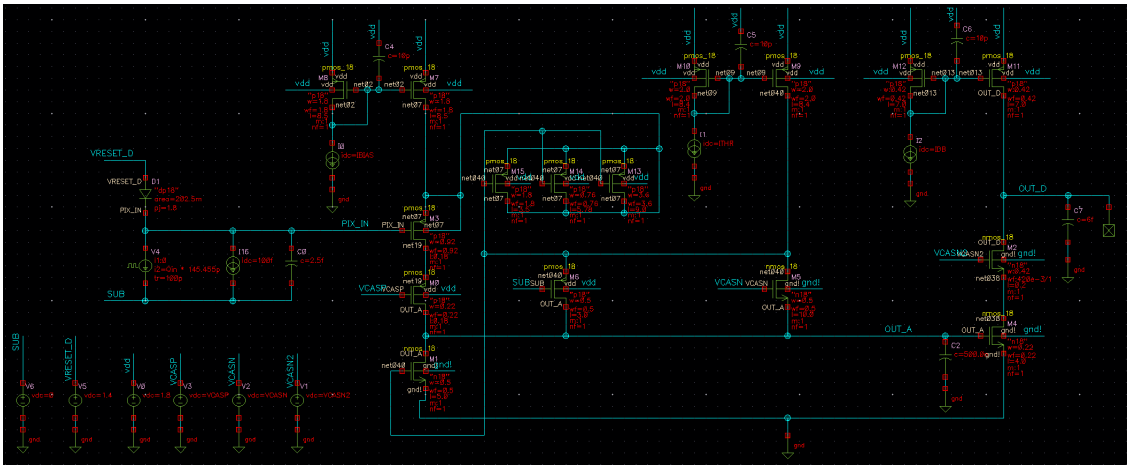


Figure 28: ALPIDE Simulation Schematic.PNG

	Minimum	Maximum	Nominal setting	Nominal value
IBIAS	0 nA	80 nA	64	20 nA
ITHR	0 nA	80 nA	51	0.5 nA
IDB	0 nA	80 nA	64	10 nA
IRESET	0.7 nA	26 pA	50	5 pA
IAUX2	-	-	-	-
VCASP	0 V	1.8 V	86	0.6 V
VCASN	0 V	1.8 V	57	0.4 V
VCASN2	0 V	1.8 V	62	0.44 V
VCLIP	0 V	1.8 V	0	0 V
VRESET_P	0.37 V	1.8 V	117	1.2 V
VRESET_D	0.37 V	1.8 V	147	1.4 V
VPLSE_LOW	0.37 V	1.8 V	0	0.37 V
VPLSE_HIGH	0.37 V	1.8 V	255	1.8 V

Table 4.2: DACs specifications overview.

Figure 29: Default threshold parameters from ALPIDE manual

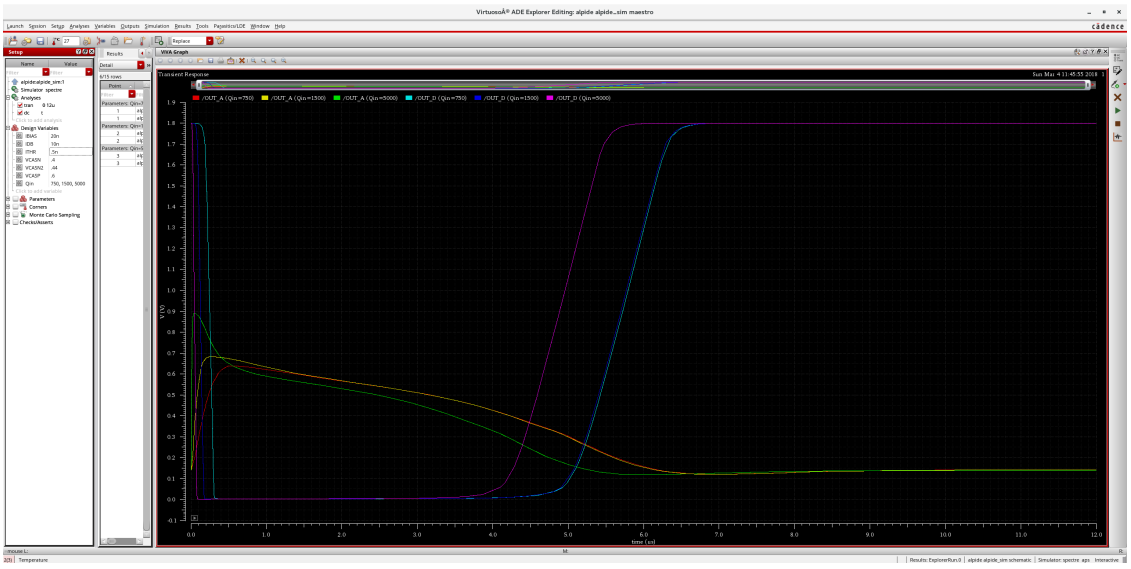


Figure 30: ALPIDE Simulation threshold parameters at defaults OUT_A and OUT_D with 750 1550 5000

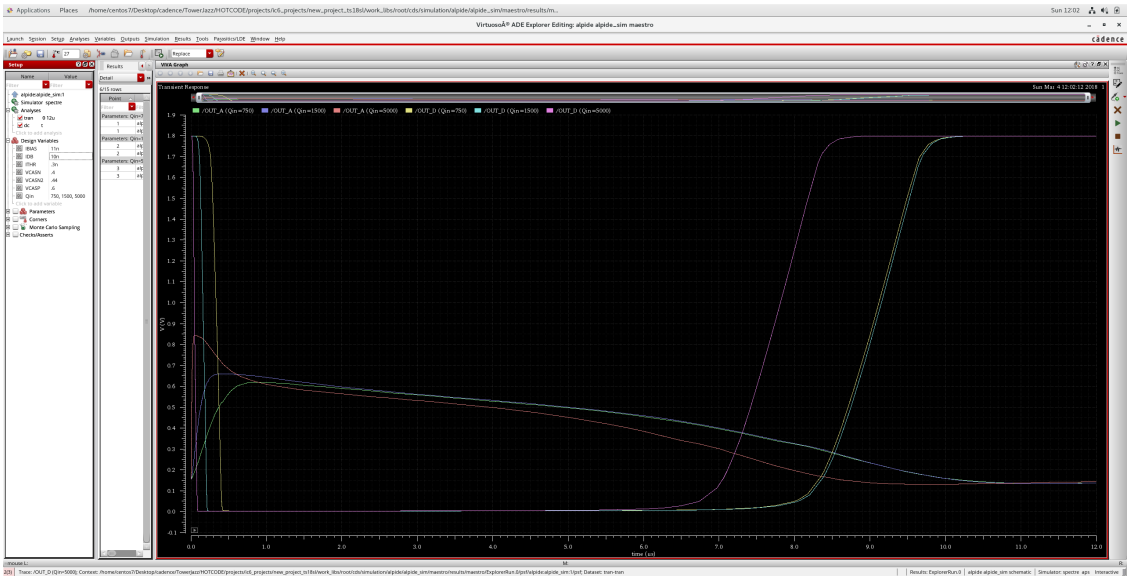


Figure 31: ALPIDE Simulation threshold parameters to get to 8us OUT_A and OUT_D with 750 1550 5000

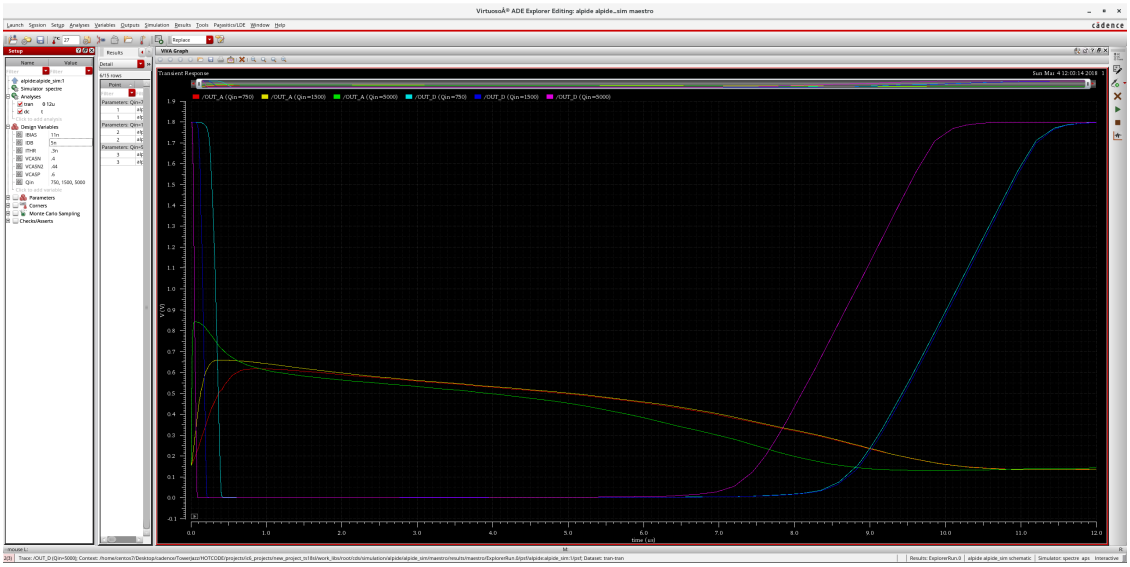


Figure 32: ALPIDE Simulation threshold parameters to get to 8us 2 OUT_A and OUT_D with 750 1550 5000

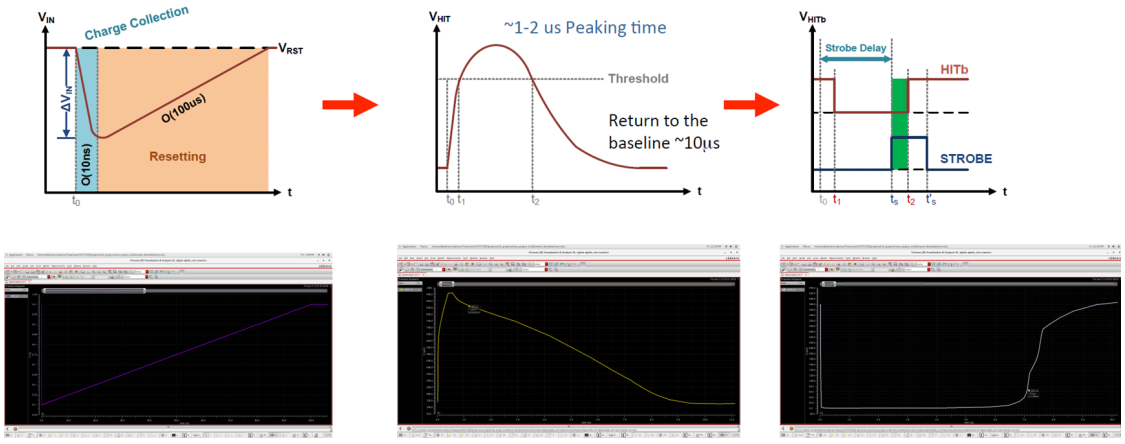


Figure 33: Simulation Validation

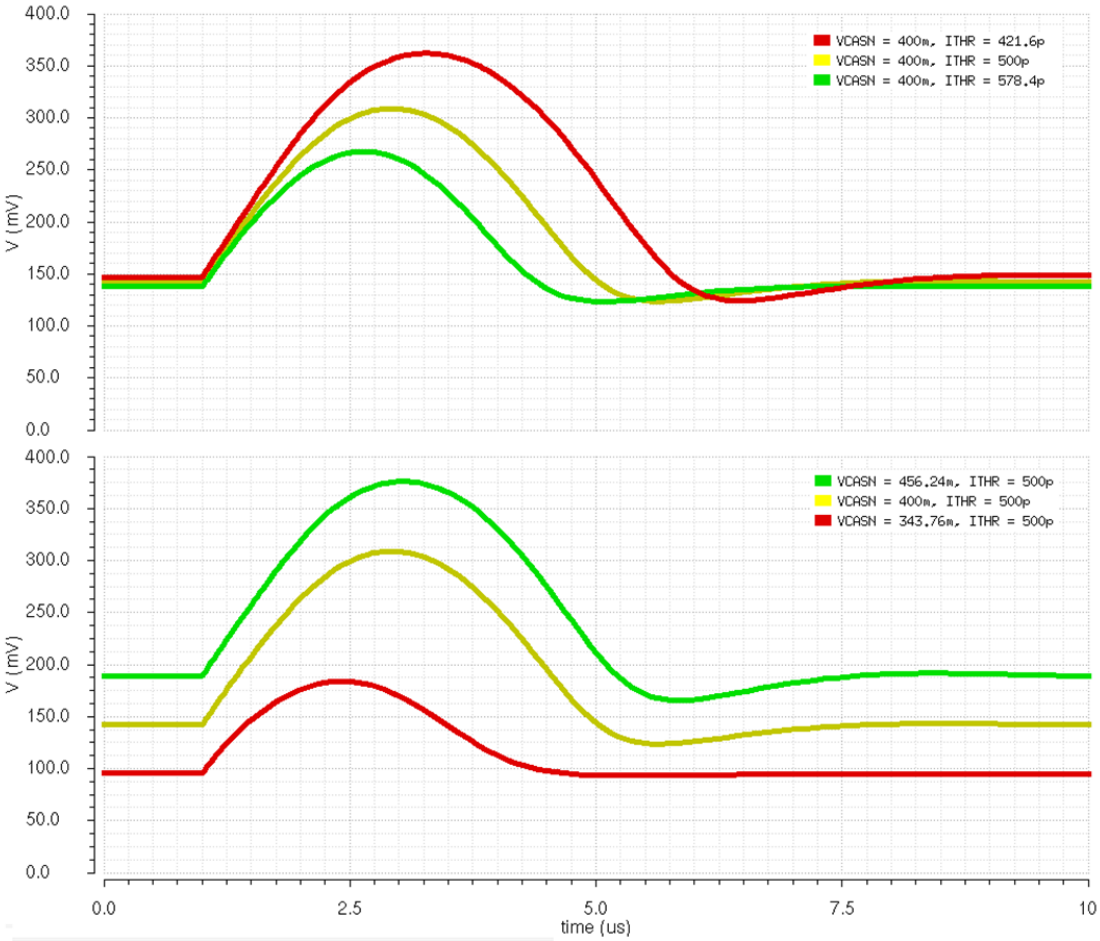
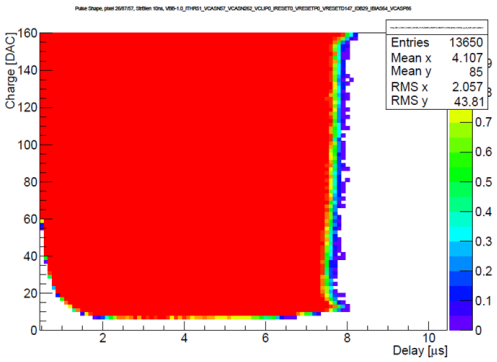


Figure 34: Threshold Parameters

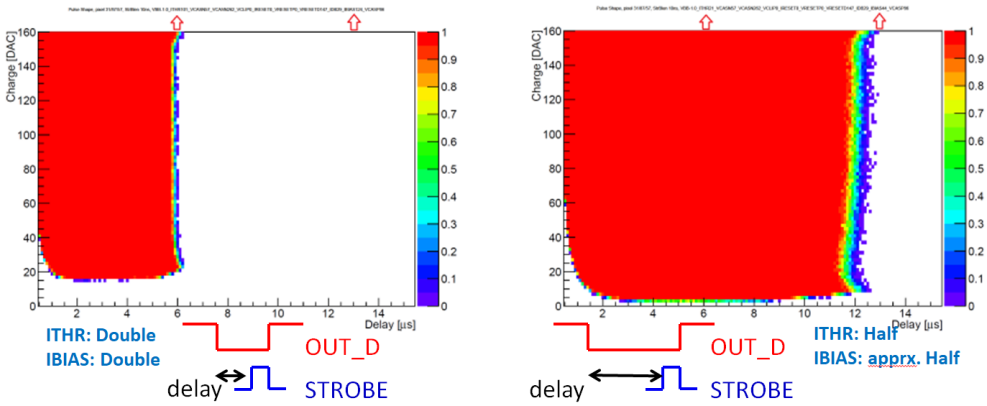
Default settings pulse injections test



	Minimum	Maximum	Nominal setting	Nominal value
IBIAS	0 nA	80 nA	64	20 nA
ITHR	0 nA	80 nA	51	0.5 nA
IDB	0 nA	80 nA	64	10 nA
IRESET	0.7 nA	26 pA	50	5 pA
IAUX2	-	-	-	-
VCASP	0 V	1.8 V	86	0.6 V
VCASN	0 V	1.8 V	57	0.4 V
VCASN2	0 V	1.8 V	62	0.44 V
VCLIP	0 V	1.8 V	0	0 V
VRESET_P	0.37 V	1.8 V	117	1.2 V
VRESET_D	0.37 V	1.8 V	147	1.4 V
VPLSE_LOW	0.37 V	1.8 V	0	0.37 V
VPLSE_HIGH	0.37 V	1.8 V	255	1.8 V

- Pulse Shape, pixel 26/87/57, StrBlen 10ns,
- VBB1.0
- ITHR51
- VCASN57
- VCASN262
- VCLIP0
- IRESET0
- VRESETP0
- VRESETD147
- IDB29
- IBIAS64
- VCASP86

Figure 35: Mosaic Pulse Injection Test



Left
Pulse Shape, pixel 17/87/57, StrBlen 10ns,
VBB1.0
ITHR101
VCASN57
VCASN262
VCLIP0
IRESET0
VRESETP0
VRESETD147
IDB29
IBIAS124
VCASP86

Right
Pulse Shape, pixel 31/87/57, StrBlen 10ns,
VBB1.0
ITHR21
VCASN57
VCASN262
VCLIP0
IRESET0
VRESETP0
VRESETD147
IDB29
IBIAS44
VCASP86

Figure 36: Mosaic Pulse Injection Test OUTD Manipulated

and provides full access to the control and status registers of the chip. The DCTRL and DCLK differential ports are implemented with a custom designed differential transceiver cell. This has been designed with reference to the standard TIA/EIA-899 Electrical Characteristics of Multi-point-Low-Voltage-Differential-Signaling (M-LVDS). The control bus can used to distribute commands to the chip, mostly on the trigger messages. The format of the transaction on the control bus has been shown in Figure 14.

All the analog signals are generated by a set of on-chip 8 bit DACs. The analog section of the periphery also contains an ADC with 10-bit dynamic range, a bandgap voltage reference and a temperature sensing circuit. The ADC can used to monitor the outputs of the DACS, the analog and digital supply voltages, the bandgap voltage and the temperature.

The Data Transmission Unit (DTU) provides a fast serial link for the transmission of the data from the ALPIDE. The MVTX chip transmits its data over a differential serial line with a line rate of 1.2 Gb/s to the off-detector electronics (such as the Readout Unit or the MOSAIC readout module). The data stream is transmitted over an aluminum over kapton FPC (Flexible Printed Circuit) with a maximum length of 300 mm and then to a micro-coaxial cable over a length of 5 m (can be tuned to 7m or 10m based on the sPHENIX integration study).

The power dissipated by the pixel chip shall not exceed 300 mWcm^{-2}. The power consumption is less than 40 mW/cm2 measured in operating condition.

2.4.1 Cables

SAMTEC DESC OF NEW CABLES

2.5 Power System

2.5.1 Specifications

Figure 37: Power supply system for MVTX.

The power for the entire MVTX system (Fig.37) is supplied by:

- one CAEN EASY3000 crate

- one A1676A EASY Branch Controller
- two A2518 LV 8V 10A (50W) modules

The system can supply 64 channels, even though MVTX will need only 48 power channels, one channel per power board/RU/stave. The CAEN crate uses one CAEN Power supply (TBD, SY8800?, A3484/5/6). The power board (Fig.38) has 32 controllable channels for each RU/stave unity. (Needs more specs for the power boards. Operation, interfacing, controls, what are the numbers in the figure? etc...Cesar) The power system will be part of the slow control system.

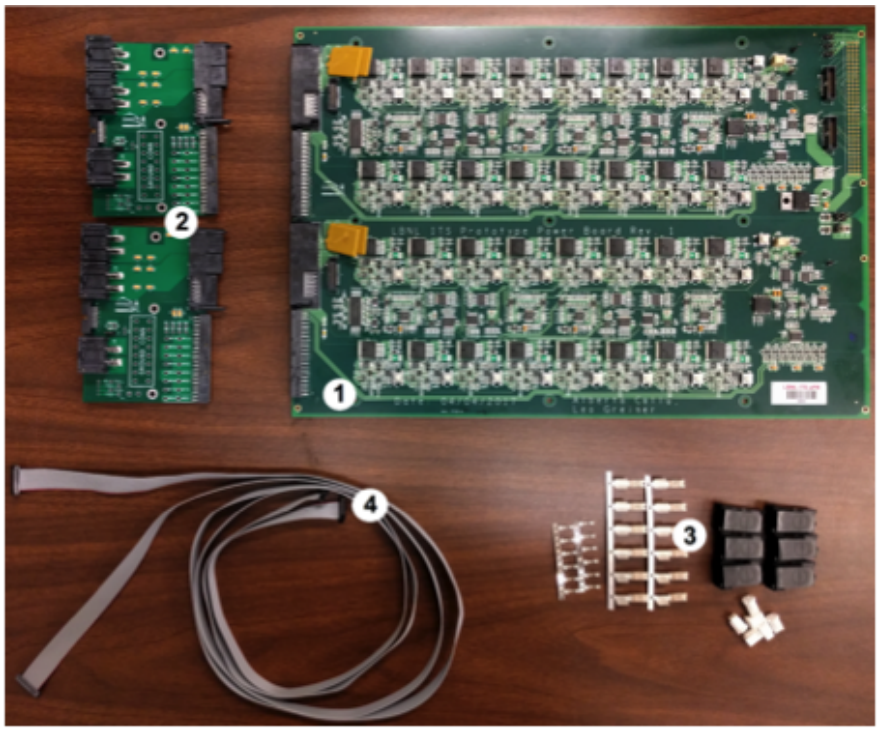


Figure 38: Power card for one Readout Unity/Stave.

1mega hit per mm squared kilahit per cm squared
negative bias gives a better depletion ithr vcasn

3 Front End Electronics

3.1 Description

The RU is the frontend board for MVTX.

3.2 Functionality

3.2.1 GBTx ASIC

The optical GBT link between the RU and FELIX is the main interface between the RU and the counting house. On the RU, the link is implemented with two components: the VTRx/VTTx optical module and the GBTx ASIC. The VTRx and VTTx are SFP-like modules designed by CERN for high radiation tolerance. The GBTx is the hardware implementation of the GBT protocol; Figure 39 shows the functionality of the ASIC.

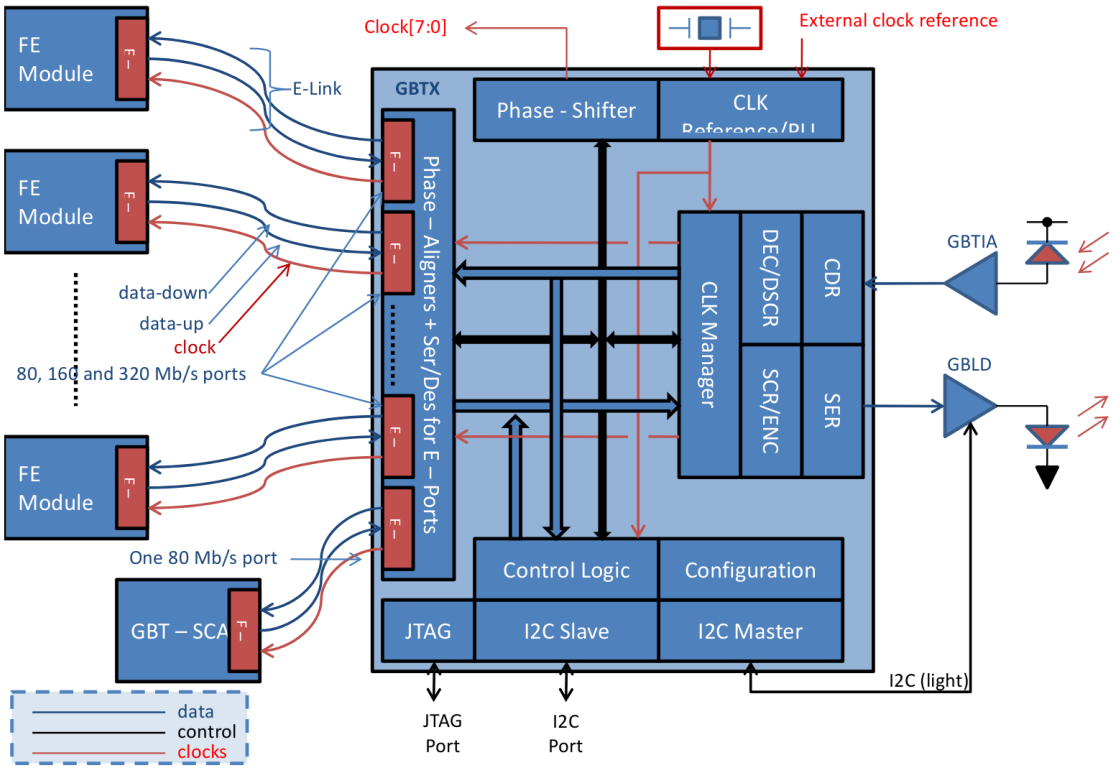


Figure 39: GBTx architecture and interfaces.

The GBTx configuration registers can be set in several ways. First, they can be accessed over the GBTx I2C slave interface; on the RU this is connected to an I2C master firmware block in the Kintex Ultrascale FPGA, so the GBTx can be configured through the FPGA's Wishbone bus (further described in Section 3.2.4. Second, the registers can be accessed through the GBT link, as described in Section 3.3.1. Third, the GBTx contains "e-fuses" that can be irreversibly programmed (using a special CERN I2C dongle), so the programmed register values are automatically loaded on power-on.

The GBTx data input and output "e-links" are connected to the Kintex Ultrascale FPGA. The data frame is 80 bits wide and clocked at 40 MHz; on the RU the e-links are configured so the frame is carried on 10 e-links. Each e-link is a differential pair carrying 8 bits of the data frame, serialized at 320 Mbps.

In the FPGA, the serialization and deserialization of the e-links are performed in the gbtx_controller firmware block. The data to be transmitted is serialized using the Xilinx OSERDESE3 primitive. The data received from the GBTx is deserialized using the Xilinx IDELAYE3 primitive (for fine adjustment of the data alignment with clock) and the ISERDESE3 primitive (deserializer), then passes through a BitSlipInLogic_8b block (based on Xilinx XAPP1208, to replace similar functionality in the 7-Series ISERDES) that corrects the word alignment of the deserialized data.

3.2.2 Readout Data Flow

Figure 40 shows the data path inside the FPGA from the ALPIDE to the GBTx ASICs. The differential signals from the ALPIDE data port are received by the GTH transceiver on the FPGA. The complete readout flow, from the transceiver to the packaging of the event for transmission over GBT, is contained in the datapath_ib firmware block, the components of which are described below. All data signals are clocked at the same 160 MHz clock.

alpide_frontend_gth : This is a wrapper containing the Xilinx transceiver IP core, which handles comma alignment and 8b/10b decoding, and a FIFO that synchronizes the decoded data to the 160 MHz readout clock. The output from this block to the next is the raw (commas and IDLEs not suppressed) ALPIDE data stream (8 bits wide), plus flags for valid (not comma and FIFO not empty) and 8b10b errors.

alpide_datapath : This block performs two functions. First, a protocol_checker block monitors the incoming data stream and checks for all possible violations of the ALPIDE protocol; the protocol error signals from this block are counted by a alpide_datapath_monitor block in datapath_ib. Second, a protocol_tracker block identifies BUSY words and only writes valid, non-BUSY words to a FIFO. The interface from this block to the next is the read interface to the FIFO (8 bits wide, clocked at 160 MHz).

data_packager : This block takes the idle-suppressed bytes from alpide_datapath and packs them into 72-bit (9 bytes) words for GBT transmission. The interface from this block to the next is a FIFO-like interface (data and valid outputs, read input) with behavior similar to a first-word fall-through FIFO. In addition, three output signals (start, stop, timeout) indicate the data stream state: whether the protocol tracker has detected a start, stop, or timeout in the current event. These signals stay high once asserted and are reset at the end of the event.

gbt_packer : This block is triplicated (TMR). In addition to the ALPIDE data interface from data_packager, this block reads the trigger information FIFO in trigger_handler. When trigger information is received, gbt_packer waits for data from the data_packager blocks. When data appears, gbt_packer transmits the event headers defined by the RU event format, then the data. Since data appears on all active data_packager blocks, the gbt_packer arbitrates between data_packager blocks: on each clock, it reads and transmits a data word from one data_packager, strobes the data_packager read line, then selects the next data_packager that is unmasked, has data (valid asserted), and has not yet sent the end of its event (stop not asserted). Once a stop has been received from every data_packager, gbt_packer transmits the event trailer.

GBT output FIFO : The output of gbt_packer is written to a FIFO that is instantiated in datapath_ib. The read interface of this FIFO is the interface from datapath_ib to the GBTx controller's TX interface.

3.2.3 Trigger

The RU can operate in two trigger modes: triggered or continuous mode. In triggered mode, the RU sends triggers to the ALPIDE sensors in response to triggers received on the GBT link. In continuous mode, the RU generates triggers on a fixed period. MVTX will use triggered mode.

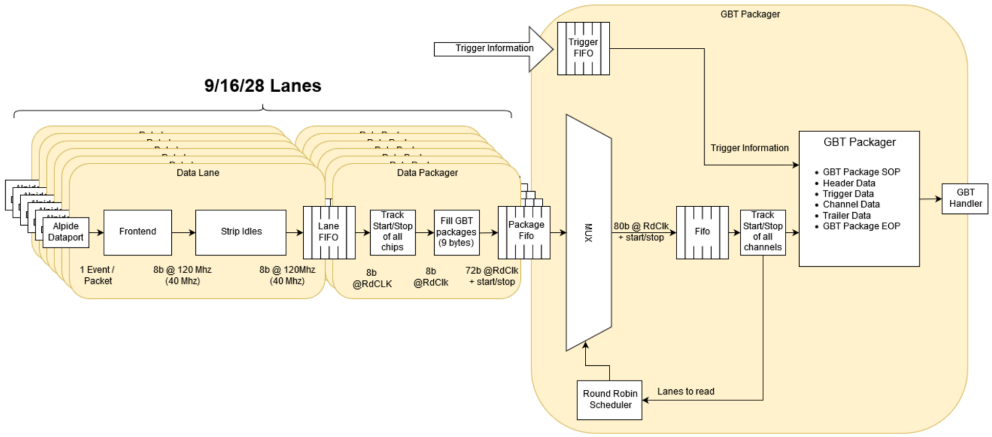


Figure 40: Block diagram of the data flow inside the RU. For MVTX, there are 9 lanes per stave, and the data lanes run at 120 MHz. This diagram corresponds to the datapath_ib block in the RU firmware. The mapping from diagram blocks to firmware is as follows: Inside "Data Lane," "Frontend" is alpide_frontend_gth and "Strip Idles" and the "Lane FIFO" are contained within alpide_datapath. "Data Packager" is data_packager. "GBT Packager" is gbt_packer.

The trigger processing functions are contained in the `trigger_handler` block in the Kintex Ultrascale FPGA firmware. This block takes the GBT RX signals as input. The important output interfaces are pulse and trigger signals, which command the ALPIDE control interface to pulse or trigger the ALPIDE sensors, and a trigger FIFO, which commands the readout path to prepare for data and communicates the trigger information that needs to be included in the event data.

The functionality of `trigger_handler` is divided among the following components:

- `trigger_handler_mode_tracker`: set the trigger mode by watching GBT RX for special control words (SOT, SOC, EOT, EOC — start/end of triggered/continuous).
- `trigger_handler_period_counter`: for continuous mode, counts time since last trigger
- `trigger_handler_distance_counter`: watches ALPIDE trigger line and reports whether the time since last trigger is less than a minimum value (for suppressing close-together triggers)
- `trigger_handler_generate_trigger`: generates a trigger pulse: for triggered mode, send a trigger if GBT RX receives a physics trigger, previous trigger was not within minimum distance, and trigger is not gated
- `trigger_handler_output_data`: stuffs the trigger FIFO with information about the triggers issued
- `pulse_transfer`: widen the output pulse of `trigger_handler_generate_trigger` from a single 160 MHz clock to 4 clocks

3.2.4 Wishbone Configuration Bus

The configuration of the Kintex Ultrascale FPGA is managed using Wishbone, an open-source bus interface. As implemented on the Kintex Ultrascale, the bus is 32 bits wide to match the width of the interface to the Cypress FX3 USB ASIC. Various firmware blocks on the FPGA connect to the shared Wishbone bus. Each such block has a Wishbone module address, and a number of 16-bit Wishbone registers. Figure 42 shows how the addresses and data are packed into a bus transaction.

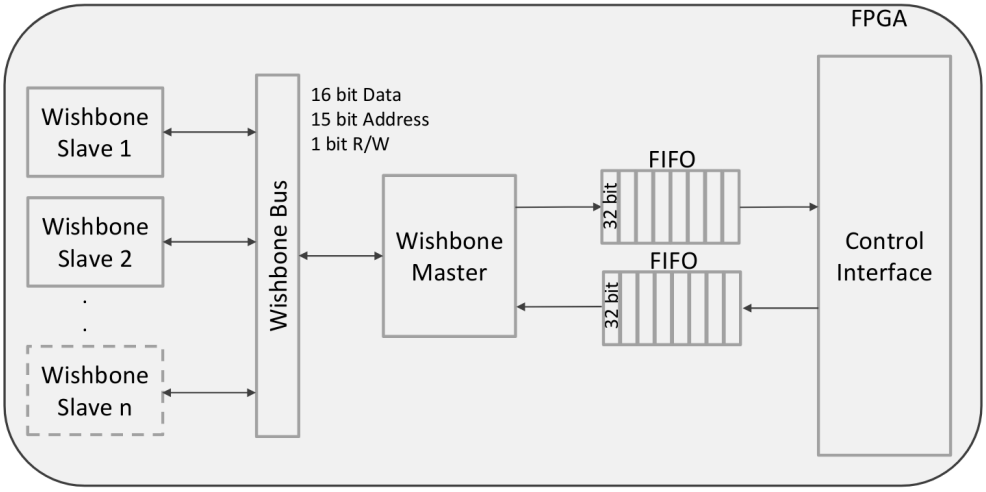


Figure 41: Block diagram of the Wishbone bus on the RU. The control interface can be either the USB or GBT interface of the RU.

- **DCS -> FPGA**
  - *Write Request:*

31	24	16	0
W/R=1	Module Addr[6:0]	Reg Addr[7:0]	Data[15:0]

  - *Read Request:*

31	24	16	0
W/R=0	Module Addr[6:0]	Reg Addr[7:0]	Don't Care[15:0]

- **FPGA -> DCS**
  - *Write Response:*

    None (no acknowledgement, but errors are kept in a wishbone slave register – associated with the wishbone master - for possible later interrogation)

  - *Read Response:*

    Errors are kept in a wishbone slave register – associated with the wishbone master - for possible later interrogation

31	24	16	0
Error	Module Addr[6:0]	Reg Addr[7:0]	Data Response[15:0]

Figure 42: Transactions on the Wishbone bus. Here DCS (Detector Control System) represents the slow control system, and can refer to either the USB or GBT Wishbone masters. The 32-bit bus accommodates a read/write flag, the address of the Wishbone module, the address of the register inside that module, and the 16-bit value of the register.

The Wishbone bus has two "bus masters" that can send read and write commands. The first is accessed through the USB interface on the RU. The second is accessed through the optical GBT interface: special Single Word Transactions (SWT)" are recognized by the GBT controller block in the firmware and interpreted as Wishbone commands.

The ProASIC3 FPGA also uses a Wishbone bus for configuration, but the bus is 15 bits wide (7-bit address, 8-bit data) to match the I2C protocol. The ProASIC3 Wishbone bus has two bus masters: one is controlled by the GBT-SCA ASIC through I2C, and the other is controlled through a UART interface.

3.2.5 Software Configuration Interface

The RU configuration and monitoring software is implemented in Python. The software interface is used primarily in a regression script (for exercising the system in an automated way) and a testbench (which provides command-line interfaces for setup and test of specific parts of the system). The main building blocks of the software interface are the RU software modules and the communication servers.

Each RU software module corresponds to a Wishbone module inside the RU. The source files for software modules are in the directory modules/board_support_software/software/py. Each software module is a subclass of WishboneModule, which is a class providing generic read and write methods for accessing Wishbone registers for the module. The software modules add module-specific functions that abstract away the register addresses; for example, to select the mask that determines which ALPIDE clock lines are driven by the ALPIDE controller module (register address 0x15), one can call Dctrl.set_dclk_mask(value) instead of WishboneModule.write(0x15,value). Modules can also add more sophisticated methods: for example Dctrl.write_chip_reg(address, data, chipid) performs the full series of Wishbone writes necessary to command the ALPIDE controller to write a specified register on an ALPIDE sensor.

The RU is represented in software by the class RUv1 (a subclass of ReadoutBoard), which collects the full set of modules that exist in the RU firmware. This class also holds the mapping between ALPIDE sensors and control ports.

The communication servers are Python classes that represent different physical interfaces by which the software can communicate with the RU. Communication servers exist for both the USB interface and the SystemVerilog simulation. For the USB interface, two implementations exist: PyUsbComm, using the PyUSB Python library, and NetUsbComm, which communicates over the network to a C++ implementation of the USB communication (UsbCommServer) using the libusb library. For the simulation, there are two communication servers: UsbCommSim, which simulates the USB interface, and Wb2GbtxComm, which simulates the GBT optical interface. Both communicate with the simulation through buffer files, each of which represents a one-way communications link: the communication server writes (or reads) the file, and the simulation reads (or writes).

The testbench script (software/py/testbench.py) uses Python Fire to generate a command-line interface. Commands in common use include `initialize_boards`, `setup_sensors`, and `setup_readout`.

3.2.6 Triple Modular Redundancy (TMR) and Scrubbing

The RU will operate in a radiation environment that can induce single event upsets (SEUs) in configuration memory, flip-flops, and combinatorial logic. A three-part strategy is used to protect the RU logic from these effects.

First, the Kintex Ultrascale firmware implements triple modular redundancy (TMR) in many of the logic blocks. TMR in the Kintex Ultrascale firmware is implemented using distributed TMR as shown in Figure 43. Sections of logic to be TMR-ed are divided into blocks consisting of combinatorial logic followed by registered outputs (no internal feedback paths). These blocks are triplicated, and the outputs are processed by triplicated majority voter circuits. As shwon in Figure 44, a majority voter is a combinatorial logic circuit

whose output follows the majority of the inputs: if one input signal is incorrect due to SEU, the output will follow the unaffected inputs and therefore will still be correct. An SEU affecting one of the triplicated blocks is ignored by the voters (since the other two blocks are operating correctly); similarly an SEU affecting one of the three voters is ignored by the voters of the next logic block. An SEU affecting a flip-flop or logic in one triplicated block will clear itself when the affected data is dropped by the voters; since there are no internal feedback paths, bad data cannot survive in the system longer than the propagation delay from one set of voters to the next. SEUs in the configuration only affect the system function if two of the triplicated blocks are affected; to avoid system failure, the first SEU must be corrected before the second SEU occurs.

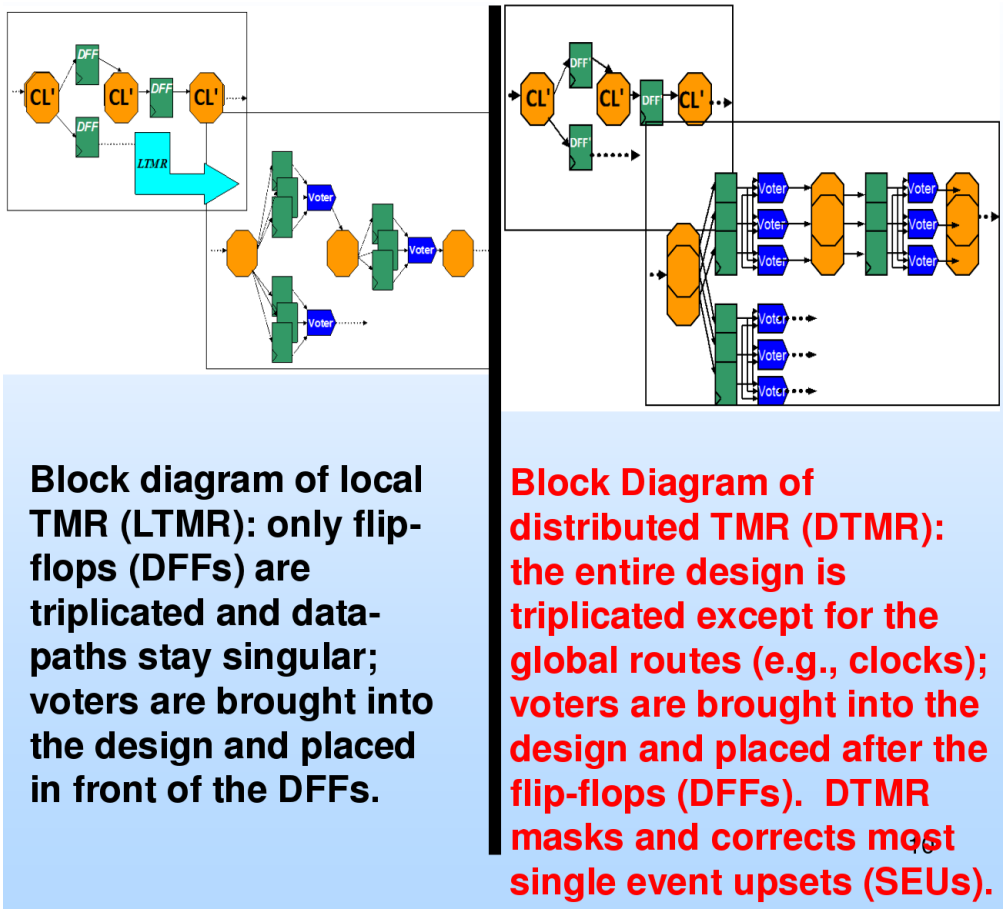


Figure 43: The two types of triple modular redundancy implemented on the RU.

The majority voter circuit has two identical "mismatch" outputs that indicate when any of the voter inputs disagree with the voter output. The mismatch outputs from all majority voters are read by a `radiation_monitor` module, which counts the mismatches and can be read out over the Wishbone bus. This allows real-time monitoring of the SEUs seen on the Kintex Ultrascale FPGA.

Not all of the Kintex Ultrascale firmware is protected by TMR; Figure 45 shows the status of TMR protection of different firmware blocks. Certain firmware blocks, which have relatively low cross-section for SEUs (therefore low SEU rate) and do not have internal state (therefore will resume normal operation if the configuration is corrected), are not TMRed.

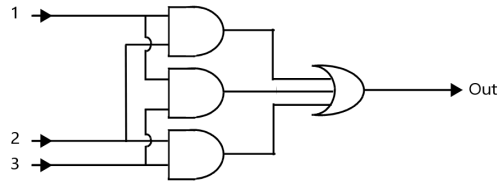


Figure 44: Majority voter circuit. On the Kintex Ultrascale firmware, the voter output is compared with every input to generate a "mismatch" output for monitoring purposes.

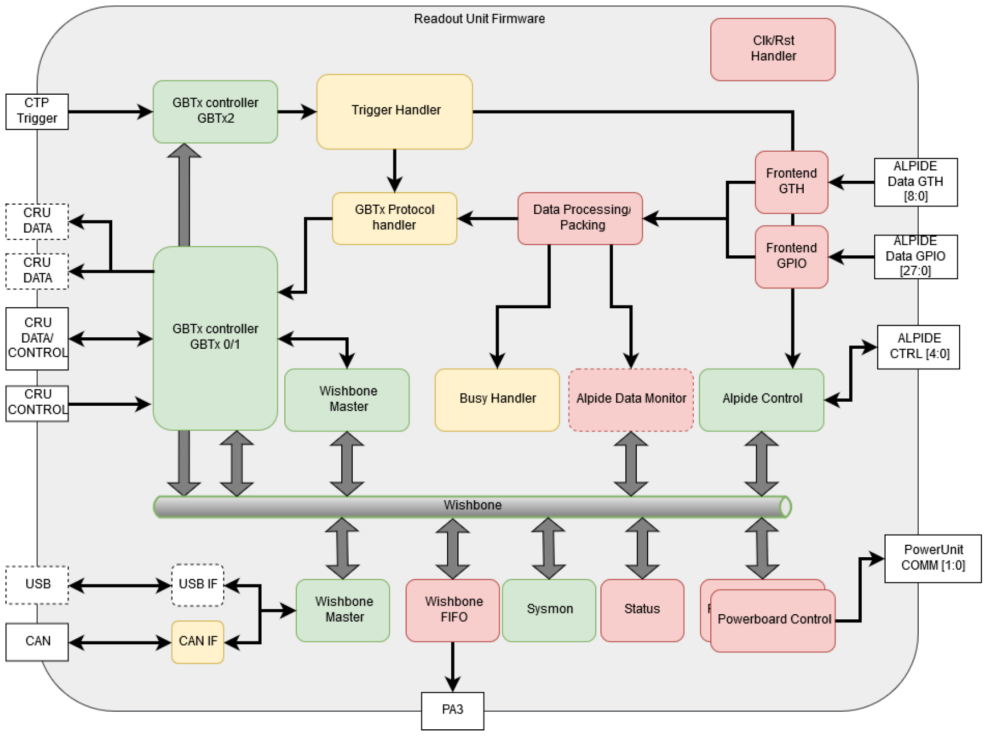


Figure 45: Block diagram of the Kintex Ultrascale firmware, with colors indicating the status of TMR protection. Green blocks are protected with TMR, yellow blocks are planned to be protected, and red blocks will not be protected.

The FPGA output signals can also be protected from radiation effects. The data-valid signals for the GBT protocol are output to the GBTx ASICs through triplicated pins, following the scheme shown in Figure 46.

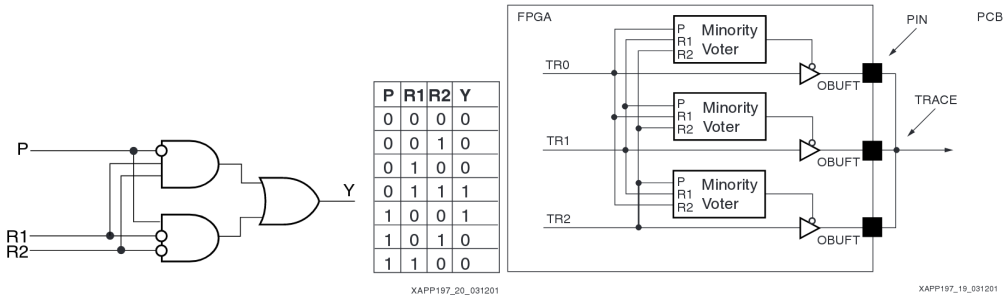


Figure 46: Minority-voted outputs implement TMR on FPGA output pins. (From Xilinx XAPP197)

Second, the Kintex Ultrascale FPGA is continuously reprogrammed during operation: this "scrubbing" corrects configuration upsets before they can cause problems. As shown in Figure 47, this is done using two additional components: a radiation-hardened ProASIC3 FPGA that reprograms the Kintex Ultrascale FPGA, and a flash memory that holds the original "golden" firmware image. The Kintex Ultrascale FPGA supports programming and reprogramming through its SelectMAP interface. The ProASIC3 FPGA operates in two modes: initial programming, where the Kintex Ultrascale is reset, programmed from the golden image, and initialized; and scrubbing, where the Kintex Ultrascale is continuously reprogrammed without interrupting operation.

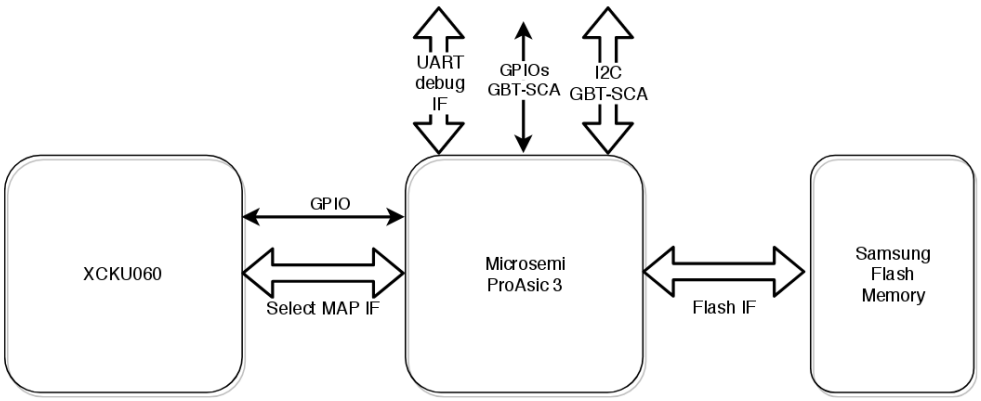


Figure 47: Block diagram of the RU scrubbing scheme, showing the XCKU060 Kintex Ultrascale FPGA to be scrubbed, the ProASIC3 FPGA that performs the scrubbing, and the flash memory that holds the golden firmware image.

Third, the ProASIC3 FPGA is protected from radiation effects. The configuration memory of the ProASIC3 FPGA is flash (as opposed to SRAM, as used in the Kintex Ultrascale) and is therefore immune to radiation effects. The combinatorial logic is also considered relatively insensitive. However, the flip-flops are still susceptible to SEUs, so TMR is still necessary. As shown in Figure 48, only firmware modules involved in the scrubbing functionality are protected with TMR. A simpler version of TMR, local TMR (see Figure 43), is used for the ProASIC3 FPGA: since the majority voter is combinatorial logic, it does not need to be triplicated. Also, since the configuration memory is SEU-immune, the ProASIC3 does not need to be

scrubbed.

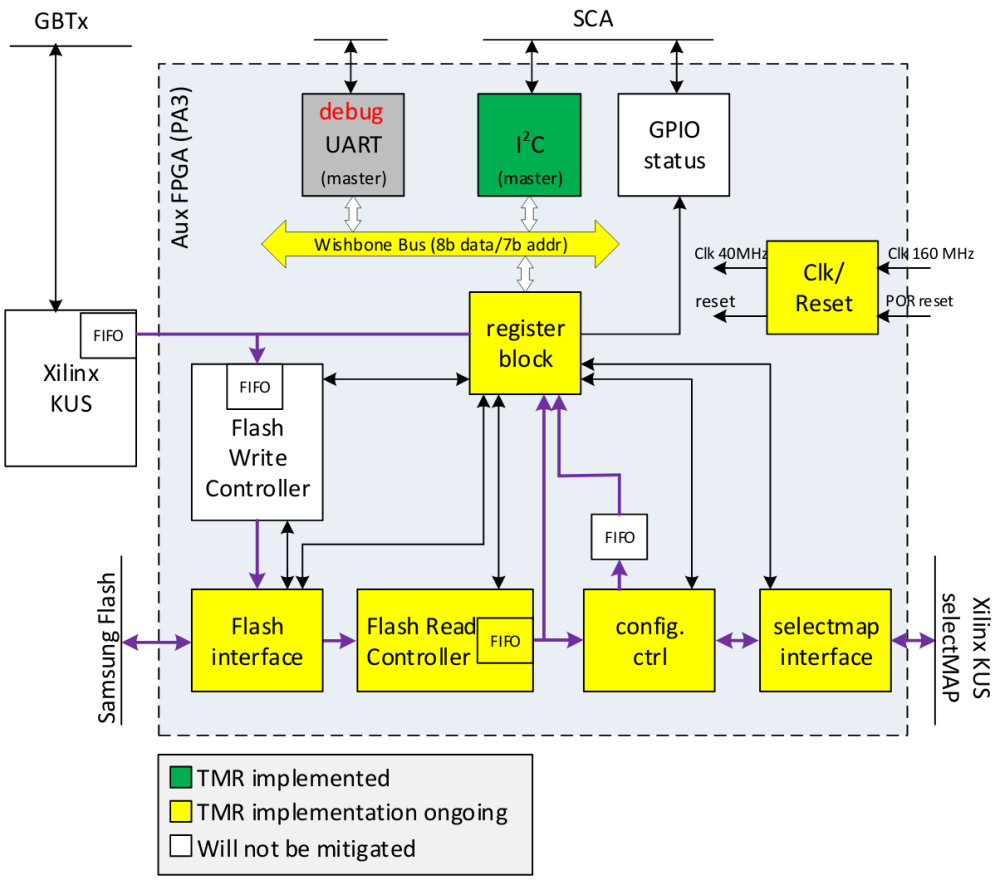


Figure 48: Block diagram of the ProAsic3 firmware, with colors indicating the status of TMR protection.

3.2.7 Clocking

Figure 49 gives an overview of the clocking scheme for the RU. The RU has two clock sources: a fixed-frequency crystal and the GBT clock from FELIX that is recovered by the GBTx ASIC. The GBT recovered clock can optionally be cleaned using a Si5338 jitter cleaner. A clock buffer IC on the RU selects between crystal and GBT clock inputs; the switch is controlled by the ProASIC3 FPGA, unless the component placement on the RU is modified to hard-wire the switch to select the GBT clock input (this was done for the 2018 beam test). The Si5338 is configured using toggle switches on the RU. The Kintex Ultrascale firmware can be compiled to choose between the clock buffer output and the raw GBT recovered clock for the user logic and for the GTH transceivers that receive ALPIDE data: the current MVTX firmware uses the clock buffer output for both.

3.2.8 Specifications

3.3 Interfaces

The RU connects to staves via Samtec FireFly cables. One RU connects to one stave through a transition board.

The stave has one clock line (unidirectional, RU→stave) and one control line (bidirectional) that are

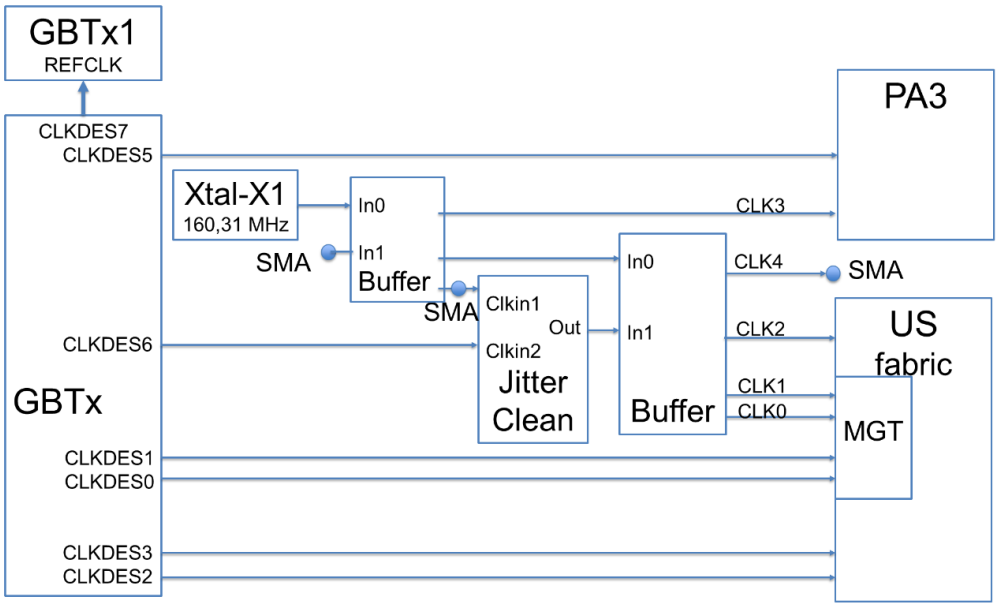


Figure 49: Block diagram of the clock distribution on the RU.

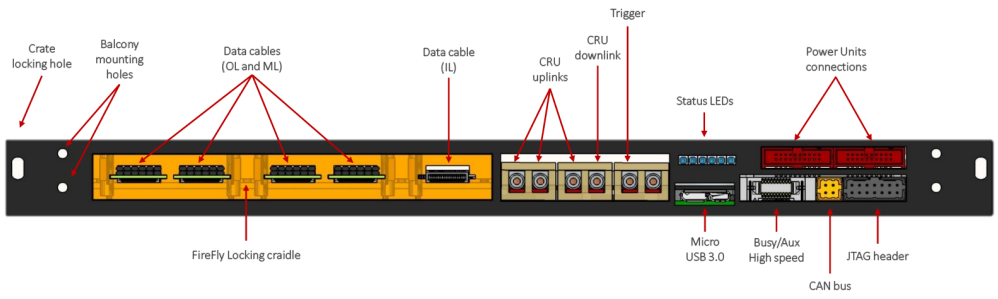


Figure 50: Front panel of the Readout Unit.

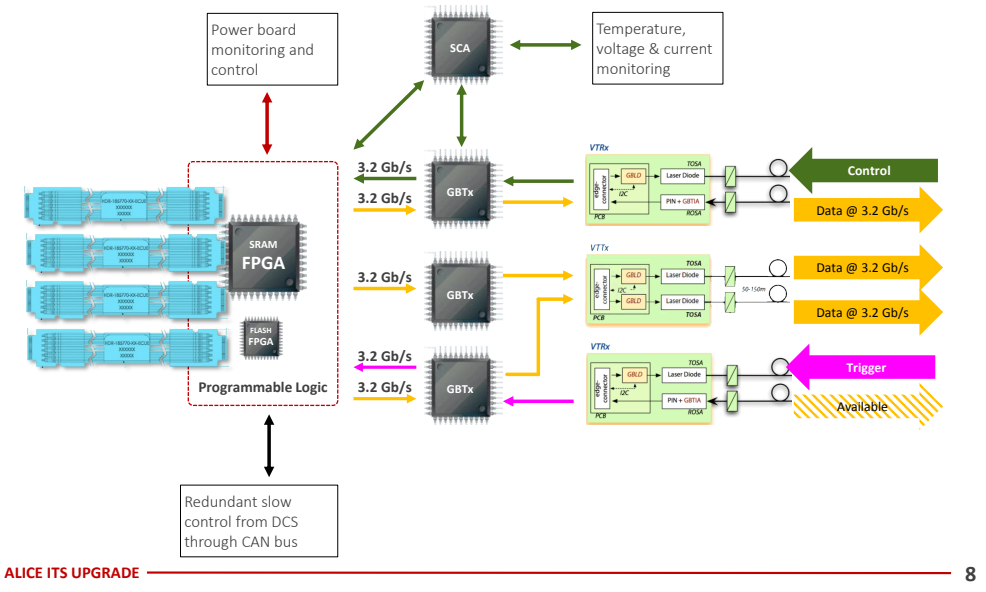


Figure 51: Block diagram of the Readout Unit's fiber links, including the mapping between GBTx chips and VTRx/VTTx fiber transceivers.

common to all ALPIDE chips, and nine data lines (unidirectional, stave→RU) that are connected to individual ALPIDE chips. Section 2.3 describes the ALPIDE interfaces.

Normally, all communication between the RU and the outside world — DAQ, slow controls, timing and trigger — takes place through the radiation-tolerant GBT (GigaBit Transceiver) fiber links. There is a backup path for slow controls through a CANbus interface. In addition, there is a port for a "busy" signal; this can be used to communicate the RU's busy status to a "Busy Unit" but could be used for another purpose.

As shown in Figure 51, the RU has three GBTx chips to support GBT fiber links. These support a total of two fiber inputs and three fiber outputs, which are driven/received by a combination of VTRx (transmitter+receiver pair) and VTTx (transmitter pair) modules. All optical interfaces use multimode fiber.

The GBTx, VTRx, VTTx, and GBT-SCA (Slow Controls Adapter) are all produced and supported by CERN as part of the Versatile Link project.

The RU receives trigger information from FELIX over a GBT link.

3.3.1 GBT Interface

The current RU firmware only uses one GBT TX/RX pair. This pair uses a single VTRx module and the first GBTx on the RU ("GBTx 0").

The GBT link carries the following functions:

- Clock: the clock recovered from the RX link is used as the RU clock for FPGA logic and the GTH transceivers on the FPGA.
- GBTx control: the IC field of the RX GBT frame is interpreted by the GBTx ASIC and used for configuration and monitoring. This communication is based on the HDLC protocol. Responses are returned on the IC field of the TX GBT frame. The protocol is described in Figure 53.
- GBT-SCA control: the EC field of the RX GBT frame is transmitted to the GBT-SCA ASIC, which is used for slow controls on the RU (ADC, DAC, GPIO, JTAG). Responses are returned on the EC field of the TX GBT frame. The protocol is described in Figure 54.
- Trigger: the Kintex Ultrascale FPGA firmware recognizes special trigger words on the data-valid and data fields of the RX GBT frame. The format is described in Figure 52.
- Data: the Kintex Ultrascale FPGA transmits ALPIDE data using the data-valid and data fields of the TX GBT frame. The format is described in Figure 52.

3.3.2 I2C Interface to Power Boards

The Power Boards power the staves: they provide analog and digital supplies and bias voltage. Each Power Board is split into two Power Units. Each Power Unit can supply 8 analog, 8 digital, and 8 bias channels, and is controlled through two I2C buses (a "main" and a "aux" bus). One RU can control the four I2C buses on one Power Board: short ribbon cables connect the RU to the Power Board, and I2C blocks in the Kintex Ultrascale FPGA firmware interface the Wishbone bus to the Power Board.

3.3.3 RU interfaces to slow controls

The primary slow control interface is through the GBT links. The sPHENIX slow controls system will communicate with the RU through FELIX; the RU then controls the stave(s) and power board(s). A backup interface exists through the CANbus. Up to 20 RUs can share a single CANbus branch.

Remote reprogramming of the RU FPGAs will be supported: the GBT-SCA can act as a JTAG master that can be remotely controlled over the GBT link.

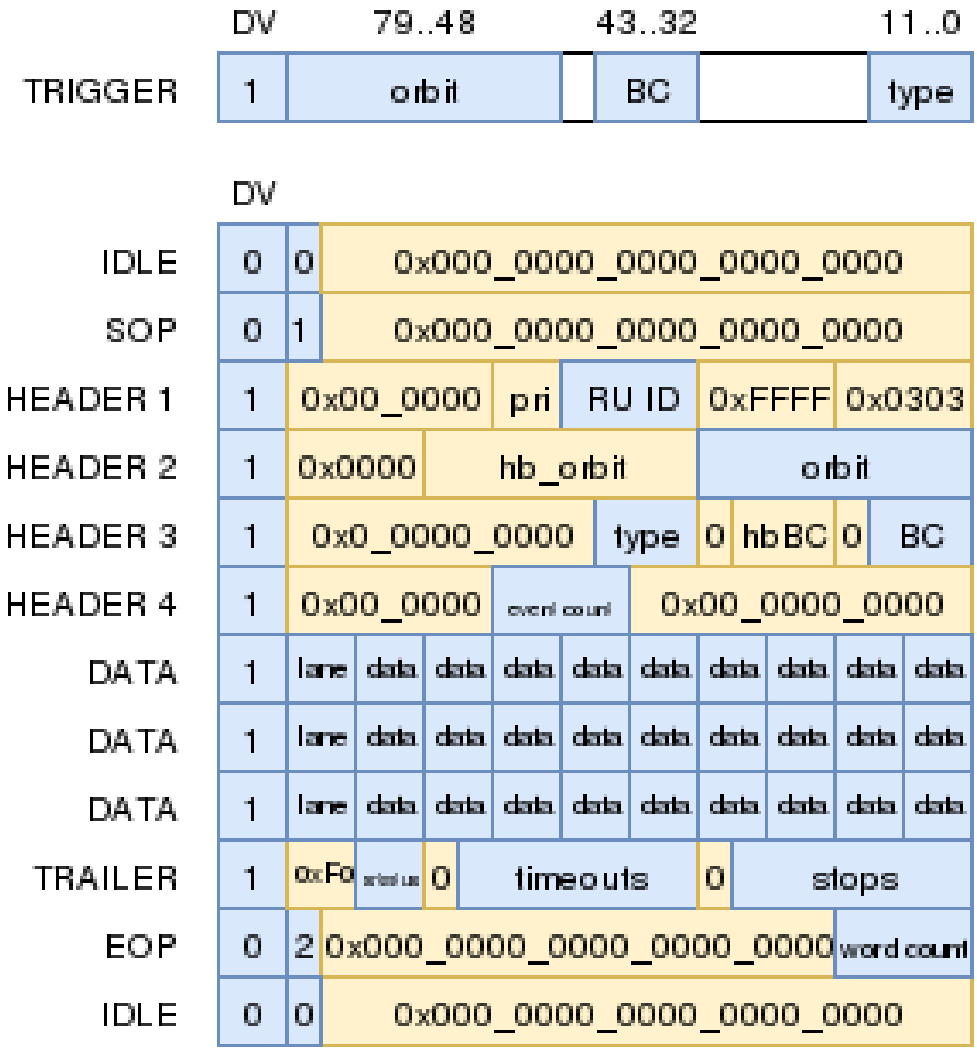


Figure 52: Trigger and data protocols for the data valid (DV) and data fields of the GBT frame. This corresponds to the RU firmware used at the 2018 test beam. Each DATA word contains a lane number and 9 ALPIDE data bytes from that lane. IDLE words may be inserted between DATA words.

Table 48 IC channel frame structure sent to GBTX for a write-read sequence

A	Frame delimiter 8'b 01111110	Not in parity check
B	GBTX i2c address (7 bits) + R/W bit = 0	Not in parity check
C	Command (8 bits)	In parity check
D	Number of data words n[7:0]	In parity check
	Number of data words n[15:8]	In parity check
E	Memory address [7:0]	In parity check
	Memory address [15:8]	In parity check
F	1st data (8 bits)	In parity check
	……	In parity check
	nth data (8 bits)	In parity check
G	Parity word (8 bits)	In parity check
A	Frame delimiter 8'b 01111110	Not in parity check

Table 49 IC channel frame structure sent to GBTX in a read-only sequence

A	Frame delimiter 8'b 01111110	Not in parity check
B	GBTX i2c address (7 bits) + R/W bit = 1	Not in parity check
C	Command (8 bits)	In parity check
D	Number of data words n[7:0]	In parity check
	Number of data words n[15:8]	In parity check
E	Memory address [7:0]	In parity check
	Memory address [15:8]	In parity check
G	Parity word (8 bits)	In parity check
A	Frame delimiter 8'b 01111110	Not in parity check

Figure 53: The HDLC packet definitions for GBTx control over the IC link.

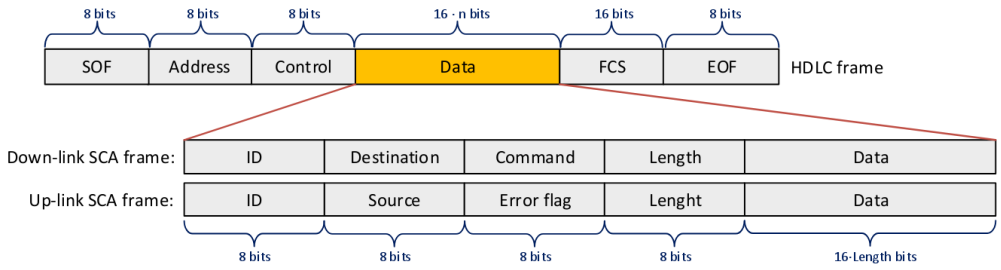


Figure 54: The HDLC packet definitions for GBT-SCA control over the EC link.

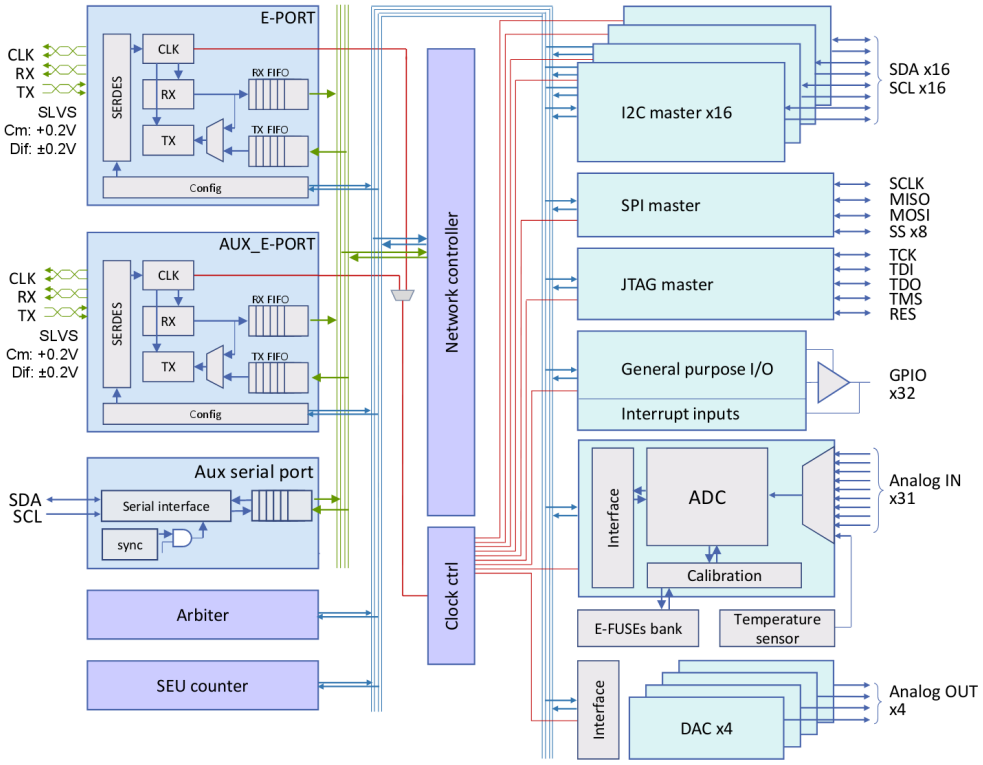


Figure 55: Block diagram of the functions of the GBT-SCA ASIC.

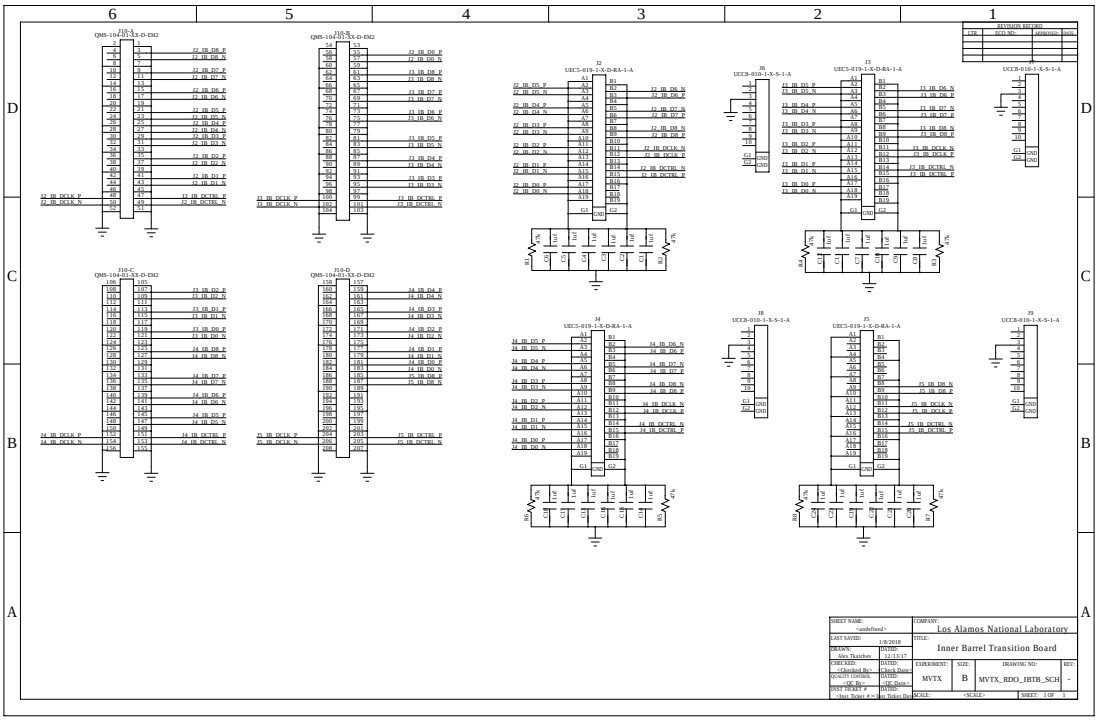


Figure 56: MVTX Readout Unit Inner Barrel Transition Board Schematic

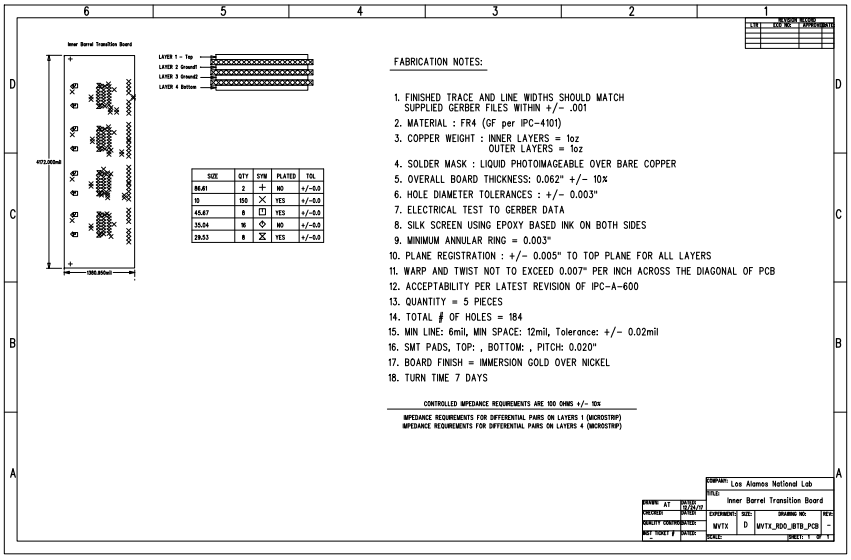


Figure 57: MVTX Readout Unit Inner Barrel Transition Board Fabrication Drawing

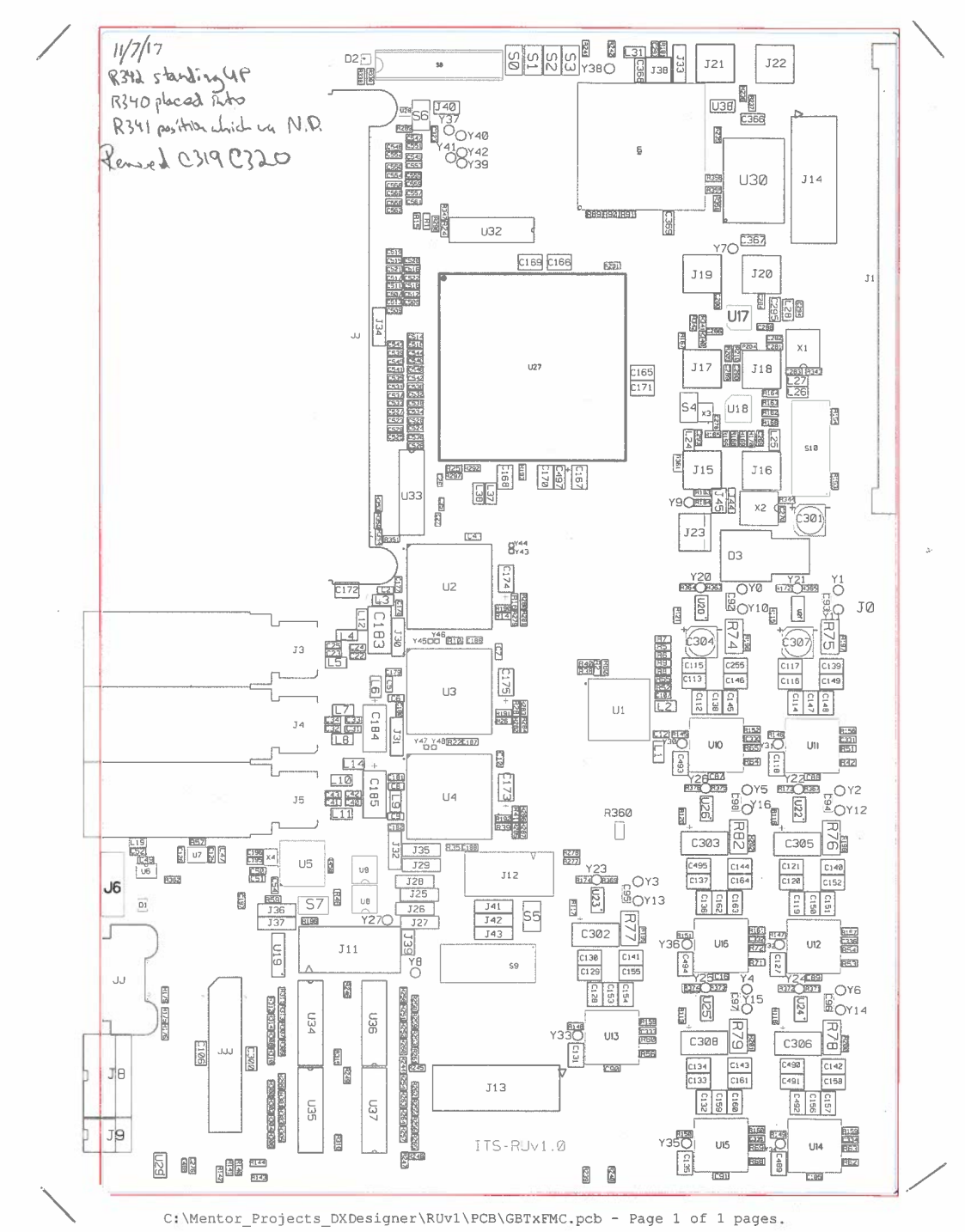


Figure 58: RDOv1 Modification Records

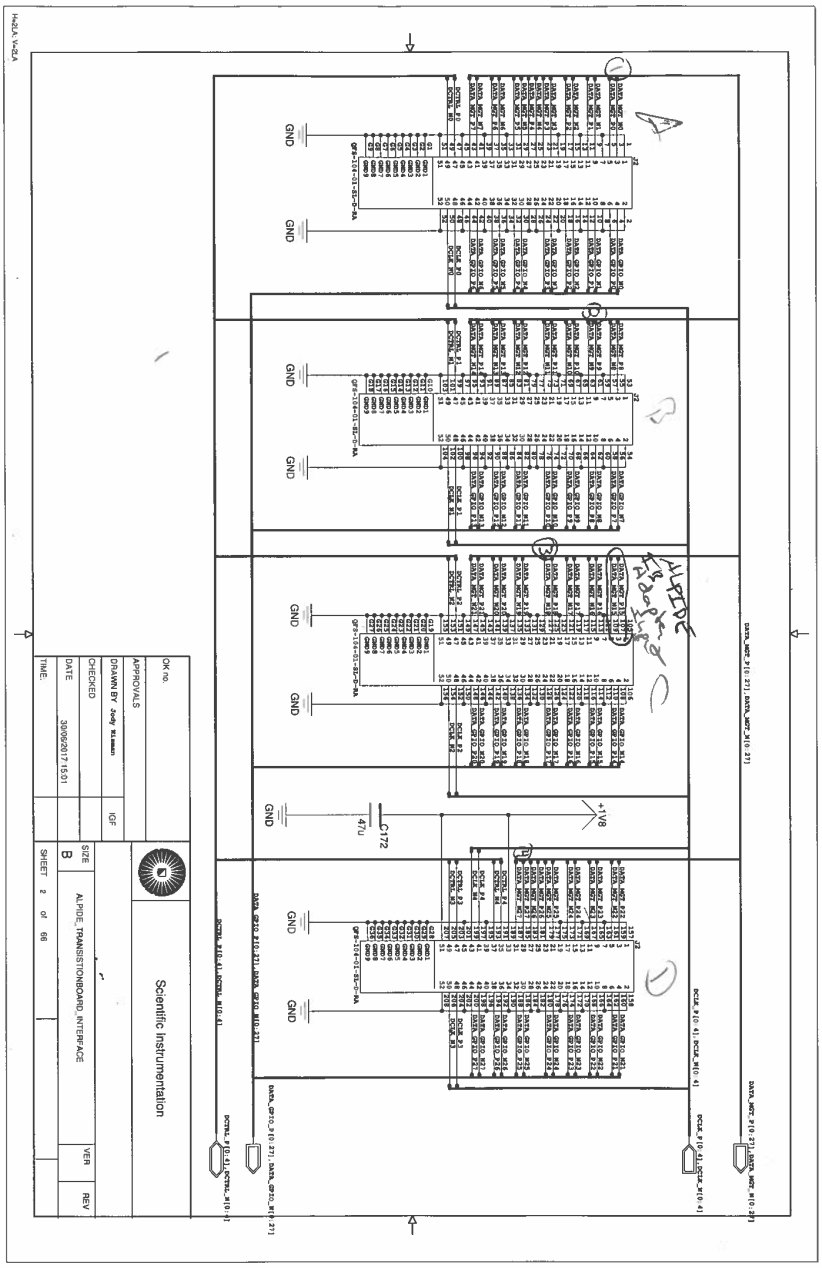


Figure 59: RDOv1 GT Mapping notes 1

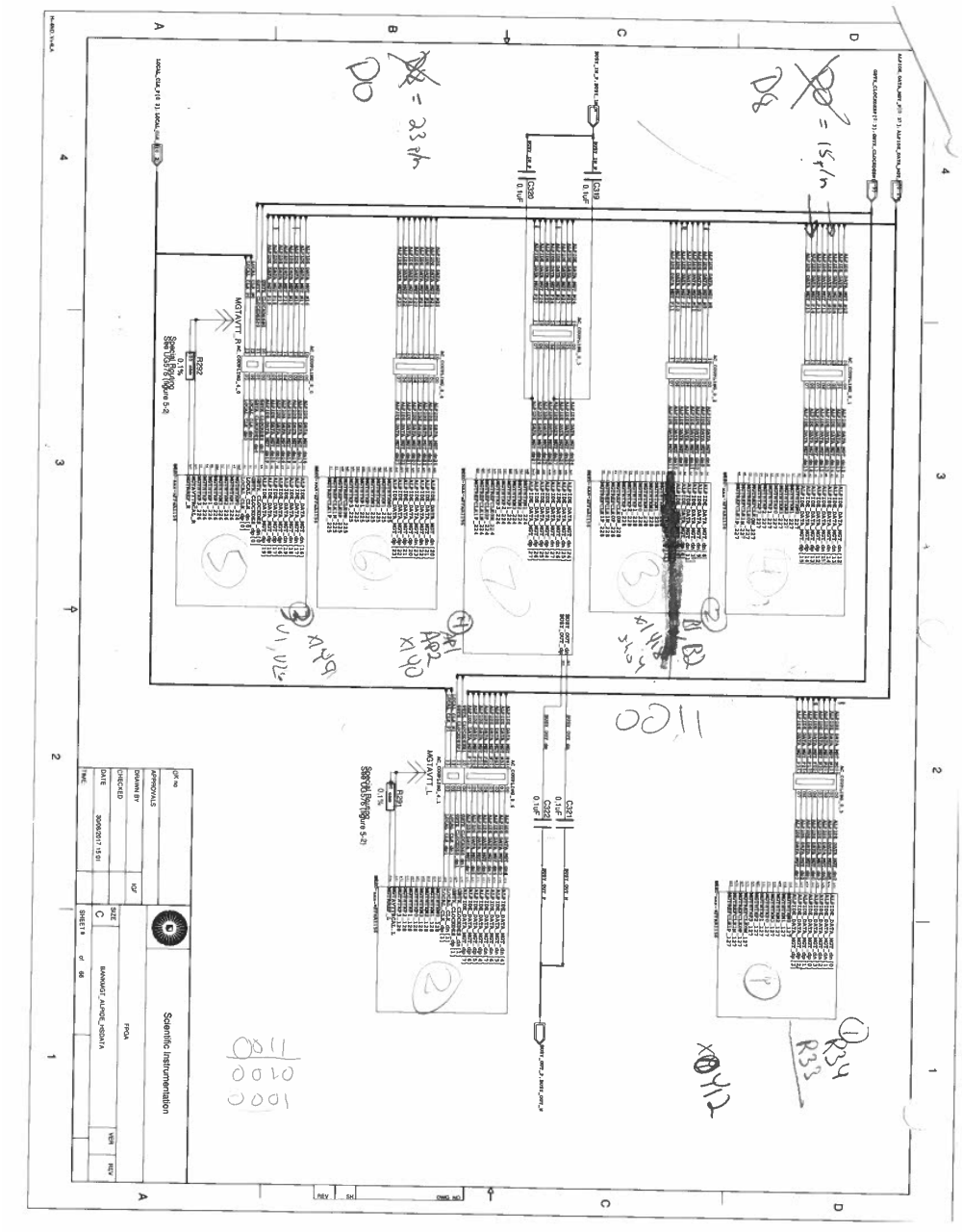


Figure 60: RDOv1 GT Mapping notes 2

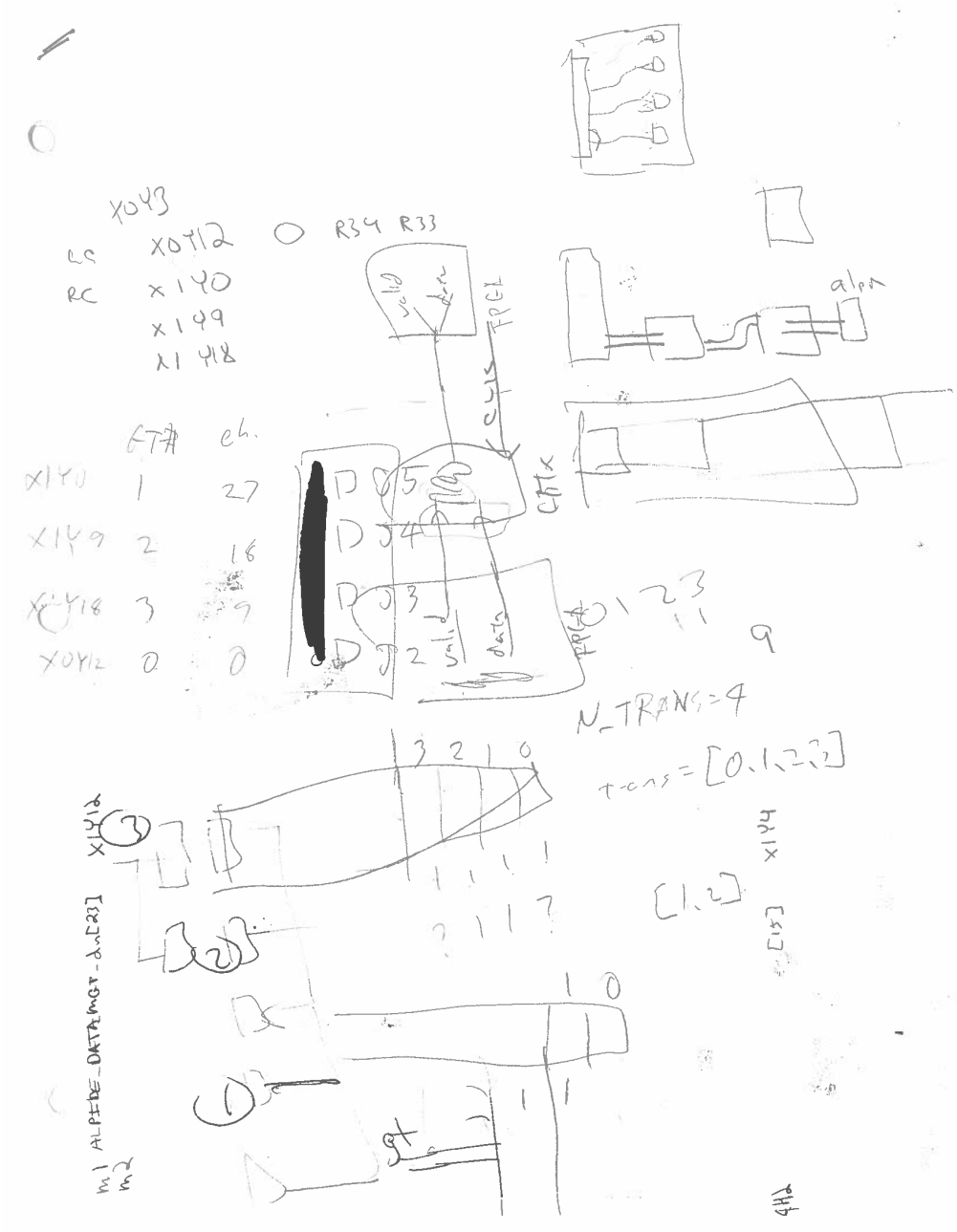


Figure 61: RDOv1 GT Mapping scratch pad

4 Back End Electronics

4.1 Description

The sPhenix MVTX detector will use the Front-End Link eXchange (FELIX) system (developed for ATLAS upgrade) as a Data Aggregation, Data Formatting, and Data Acquisition infrastructure. The FELIX system will distribute Clock, Slow Control, and Trigger to the RU and ALPIDE.

FELIX will serve as a PC-based gateway interfacing custom radiation tolerant optical links to PCIe Gen3. Figure 62

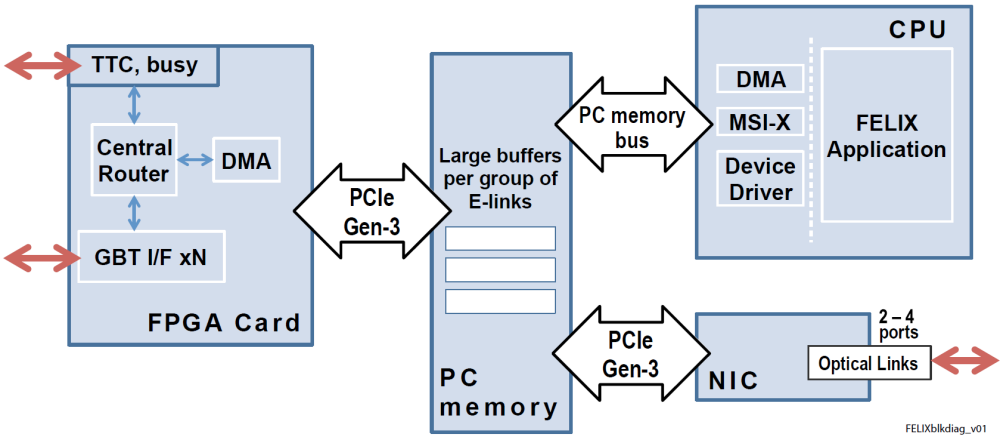


Figure 62: FELIX Block Diagram

Figure 63 shows FELIX firmware architecture which primarily consists of 1) GBT-FPGA core, also referred to as "GBT wrapper," 2) the Central Router for internal data multiplexing, 3) the Timing and Trigger TTC decoder and 4) the DMA engine and PCIe engine referred to as "Wupper." The full list of top-level firmware modules is given in table 3.

The number of GBT links in the FELIX firmware, and other implementation-time options (listed in Table 4), are set in the implementation script `do_implementation_BNL711.tcl`. These options are set as generic parameters in the firmware. This allows the same firmware project to be used to generate bitstreams with different features, or for different hardware.

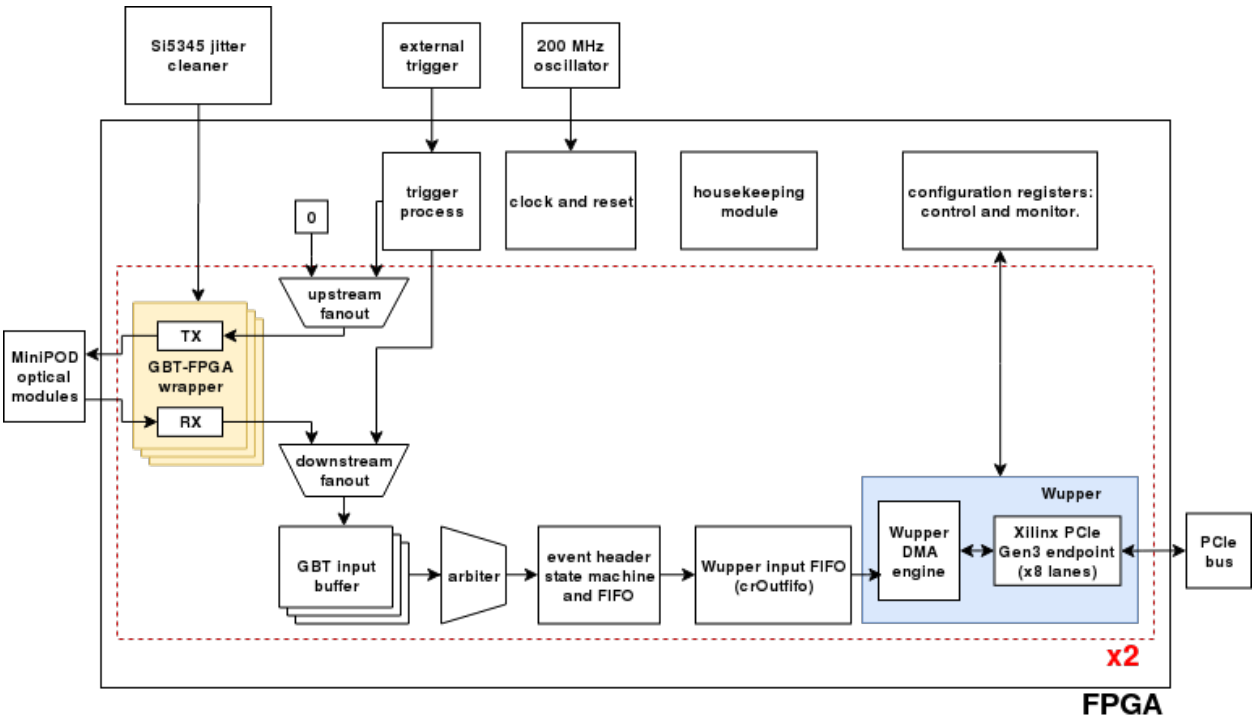


Figure 63: FELIX Firmware Diagram

4.2 GBT wrapper

The GBT wrapper is the bidirectional interface connecting the RU to FELIX via an optical link. The IP core was developed by CERN as part of the GBT-FPGA project and later enhanced by Kai Chen. The clocking methodologies will be described at a later time QPLL vs. CPLL. The difference between the GBT mode and full mode will also be explained at a later time. The MVTX detector will use GBT protocol.

Table 3: Top-level modules in the FELIX firmware.

name	number of instances	description
upstream_fanout_selector	1	mux used to enable/disable sending triggers to FELIX_gbt_wrapper_KCU inputs
FELIX_gbt_wrapper_KCU	1	FPGA implementation of GBT protocol
downstream_fanout_selector	1	mux connecting the FELIX_gbt_wrapper_KCU outputs and gbt_inbuf inputs
gbt_inbuf	GBT_NUM	buffers and packs data words from GBT inputs
rrarbiter	1	arbitrates data transfer from gbt_inbuf buffers to Wupper input FIFO
wupper	2	PCIe/DMA engine
register_map_sync	2	clock domain crossing between PCIe clock and 40 MHz clock
clock_and_reset	1	instantiates MMCM clock managers, generates reset when clock is lost
housekeeping_module	1	interface between register map and onboard devices (I2C, enable pins)
pex_init	1	initialize PEX8732 PCIe switch (I2C, hard-coded)
not used for MVTX		
HDLC_TXRX_WRAPPER	1	interfaces GBT slow control links (IC/EC) to the FELIX register map
debug_port_module	1	output certain signals on test points
ttc_wrapper	1	interface to ADN2814 CDR, decodes ATLAS TTC protocol
TTCdataSwitch	1	switch between received TTC data and an emulator
ttc_busy	1	drives busy output
LMK03200_wrapper	1	initialize LMK03200 jitter cleaner (SPI, hard-coded)

Table 4: Options in the do_implementation_BNL711.tcl script.

name	typical value	description
CARD_TYPE	711	code indicating the board type (VC-709, HTG-710, FLX-711), selects FPGA-specific (Virtex-7 vs. Kintex Ultrascale) behavior
GENERATE_GBT	true	generate the firmware blocks in the GBT data path (rrarbiter, gbt_inbuf, FELIX_gbt_wrapper_KCU, upstream_fanout_selector)
DEBUG_MODE	false	in debug_port_module.vhd, enable debug output to test points
GBT_NUM	6	number of GBT links
PLL_SEL	CPLL	PLL (CPLL or QPLL) type for the GTH transceivers used for GBT
USE_BACKUP_CLK	true	use onboard fixed oscillator (vs. TTC clock)
AUTOMATIC_CLOCK_SWITCH	false	automatic switching between onboard and TTC clocks
NUMBER_OF_INTERRUPTS	8	number of interrupts per Wupper PCIe/DMA engine
NUMBER_OF_DESCRIPTORS	2	number of descriptors per Wupper PCIe/DMA engine
PCIE_PLACEMENT	SLR0	which side of the FPGA to place the second PCIe endpoint (differs between FELIX v1.5 and v2.0)

The wrapper integrates several GBT links. Figure 64 shows a block diagram of a single GBT link or the actual channel. It is composed of a GBT and GTH IP. The GBT IP consists of GBT Tx (that scrambles and encodes the transmitted parallel data), and GBT Rx (that aligns, decodes and descrambles the incoming data stream) which implement the GBT protocol. The GTH IP consists of the Multi-Gigabit Transceiver (MGT) (that serializes, transmits, receives and de-serializes the data). The MGT is the dedicated Serializer/Deserializer (SerDes) in the FPGA . CPLL and QPLL are inside the MGT. Figure 65 shows a detailed block diagram of the GBT and GTH IP. The following section describes each block and the GBT frame shown in figure 97

DC-balance of the data being transmitted over the optical fiber is ensured by scrambling the data. For forward error correction the scrambled data and the header are Reed-Solomon encoded before serialization. The 84-bits (80 bits data, 2 bits IC and 2 bits EC) are first processed by the scrambler, the header is then added (4 bits), the Reed- Solomon (RS) encoding and interleaving takes place and finally the data is serialized. While the scrambler maintains the word size, the RS encoder adds the 32-bit Forward Error Correction (FEC) field adding up to a total frame length of 120 bits. This leads to an overall line code efficiency of 84/120 = 70 percent. At the receiver end the inverse operations are repeated in the reverse order. The fact that RS encoding and decoding are the first and the last operations to be done respectively at the transmitter and receiver (before transmission and after reception) ensures that transmission errors do not get multiplied by the scrambler operation.

4.3 Data Processing

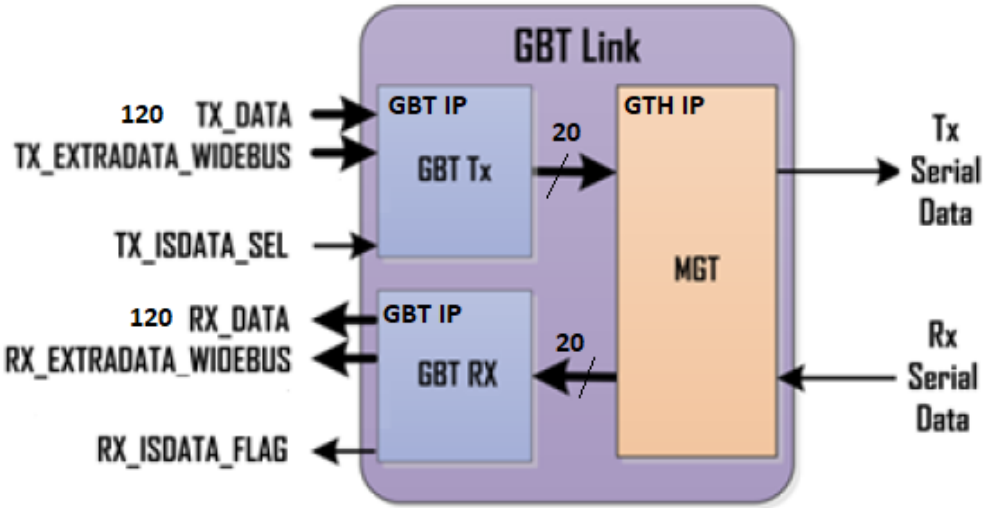


Figure 64: GBT Wrapper

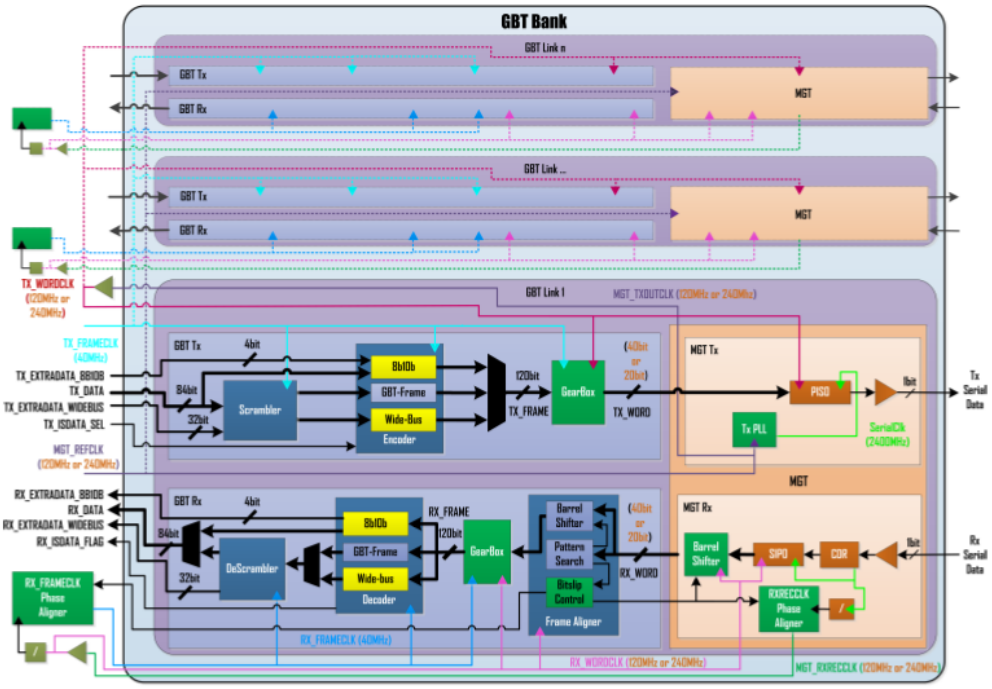


Figure 65: GBT TX RX Block Diagram

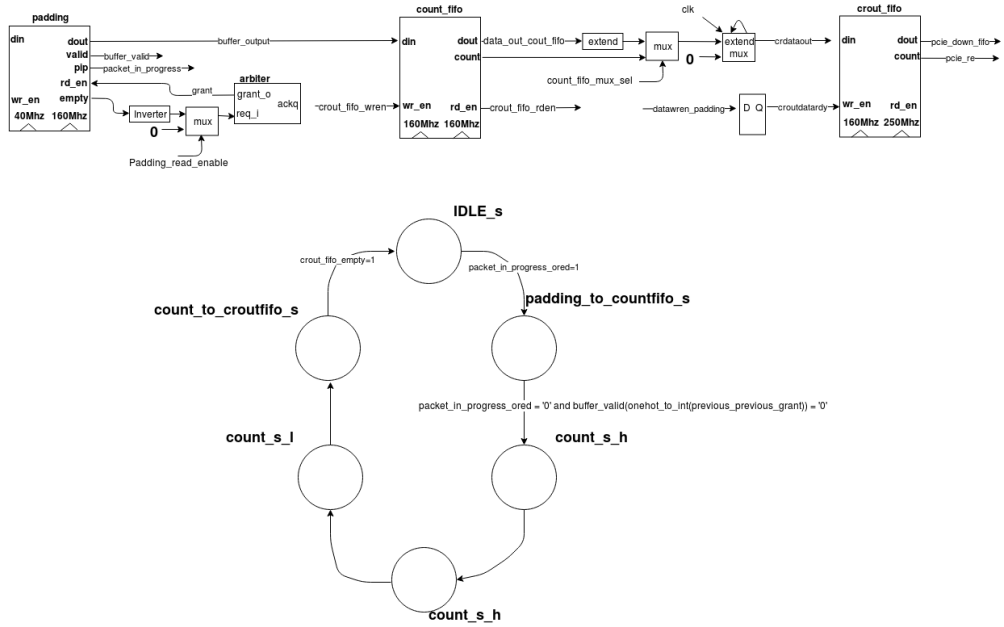


Figure 66: FELIX MVTX Datapath

4.4 Wupper

The Wupper core is a module of the FELIX firmware that provides Direct Memory Access for the 256 bit wide AXI4-Stream interface of the Xilinx Kintex Ultrascale XCKU115 FPGA Gen3 Integrated Block for PCI Express (PCIe) endpoint IP. The main purpose of Wupper is to handle data transfers from a user interface, i.e a FIFO (croutFIFO), to and from the host PC memory (server that holds FELIX). A standard FIFO (croutFIFO) is used with the same width as the Xilinx AXI4-Stream interface (256 bits) and runs at 250MHz. The event header state machine shown in Figure 63 writes to the croutFIFO. Wupper handles the transfer into Host PC memory, according to the addresses specified in the DMA descriptors. Several descriptors can be queued, up to a maximum of 8, and they will be processed sequentially one after the other. The other functionality supported by Wupper is the access to control and monitor registers inside the FPGA, and the surrounding electronics, via a simple register map.

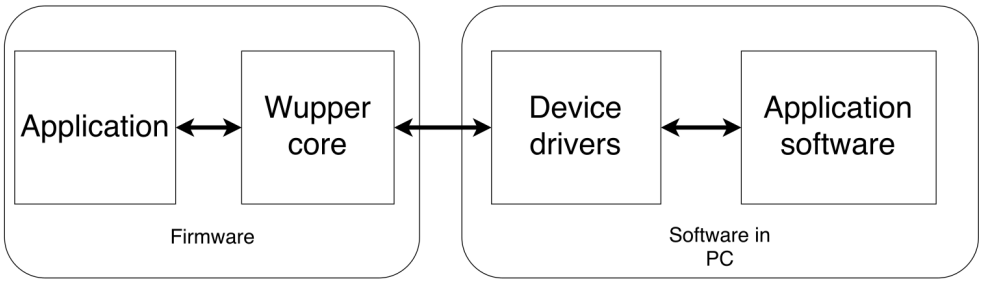


Figure 67: Wupper Package Overview

The Wupper core moves data bidirectionally to a memory without CPU intervention. This method is

used for handling large amounts of data, which is crucial for throughput intensive applications. During a DMA transfer, the DMA control core will take control according to the information provided by a DMA descriptor, and by flagging completion of operations in a per-descriptor status register.

The Xilinx PCIe EndPoint IP core is configured as a PCI express Gen3 (8.0GT/s) End Point with 8 lanes and the Physical Function (PF0) max payload size is set to 1024 bytes. AXI-ST Frame Straddle is disabled and the client tag is enabled. All other options are set to default, the reference clock frequency is 100MHz and the only option for the AXI4-Stream interface is 256 bit at 250MHz

Each Xilinx PCIe endpoint IP creates a memory map containing the three PCIe BARs specified in the IP configuration. Each BAR requests a specific region of the host PC RAM. The BARs are divided in three regions: BAR0, BAR1 and BAR2. BAR stands for Base Address Region. Every BAR has 1 MB of address space. Please refer to the PCIe Appendix for additional details.

BAR0 contains registers associated with DMA, like the DMA descriptors. The descriptors specify the addresses, transfer direction, size of the data and an enable line. BAR1 is reserved for the interrupt mechanism and consists of 8 vectors. BAR2 is used for Control and monitoring to user applications. Inside the Wupper core a DMA read/write process, sends and receives AXI4 commands over the AXI4-Stream bus. The PCIe translates this into differential electrical signals.

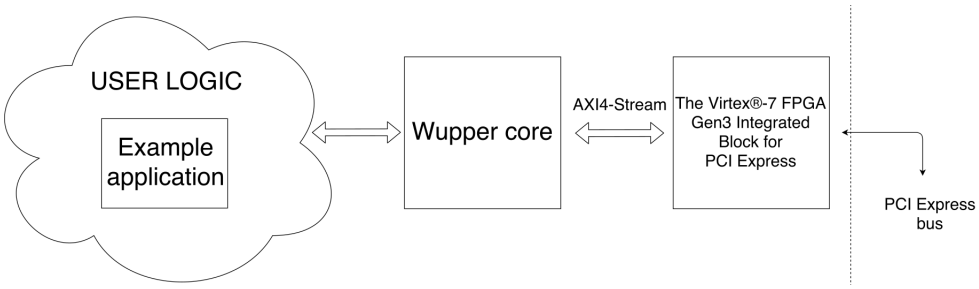


Figure 68: Wupper PCIe interface

Wupper manages a set of DMA descriptors, with an address, a read/write flag, the transfer size (number of 32 bit words) and an enable line. These descriptors are mapped as normal PCIe memory or IO registers. Besides the descriptors and the enable line (one per descriptor), a status register for every descriptor is provided in the register map. In accordance with the AXI4-Stream the interface does not contain any addresses but instead, address and other information are supplied in the header of each PCIe Transaction Layer Packet (TLP). Please see PCIe and AXI appendix, where more details regarding TLP are discussed.

Figure 73 shows the Wupper block diagram which consists of two primary parts DMA_control and DMA_read_write, which are described in detail later in this section. These handle bidirectional data transfer through DMA, and register reads and writes. The Wupper core can also fire MSI-X (Message Signaled Interrupt eXtended) type interrupts by means of the interrupt controller.

The throughput achievable is the theoretical maximum throughput of 64 Gb/s of an 8-lane Gen3 PCIe interface. However, the 16-lane PCIe interface of the FELIX provides a theoretical maximum throughput of 128Gb/s. This is supported by using two instances of the Wupper engine and by splitting the 16 lanes in two sets of 8 lanes, with each set handled by a Wupper engine.

Each DMA transfer To and From Host is achieved by means of setting up descriptors on the server side, which are then processed by Wupper. The descriptors are set in the BAR0 section of the register map. Please see excerpt below.

4.4.1 DMA Operation

Table 5: DMA descriptor definition. The descriptor fields (DMA_DESC_x and DMA_DESC_xa) are written by software; with the exception of RD_POINTER, they are set before the descriptor is enabled. The status fields (DMA_DESC_STATUS_x) are written by the Wupper firmware and update while the descriptor is enabled.

Address	Name/Field	Bits	Type	Description
0x0000	DMA_DESC_0			
	END_ADDRESS	127:64	W	End Address
	START_ADDRESS	63:0	W	Start Address
0x0010	DMA_DESC_0a			
	RD_POINTER	127:64	W	server Read Pointer
	WRAP_AROUND	12	W	Wrap around
	READ_WRITE	11	W	1: FromHost/ 0: ToHost
	NUM_WORDS	10:0	W	Number of 32 bit words
...				
0x0200	DMA_DESC_STATUS_0			
	EVEN_PC	66	R	Even address cycle server
	EVEN_DMA	65	R	Even address cycle DMA
	DESC_DONE	64	R	Descriptor Done
	CURRENT_ADDRESS	63:0	R	Current Address
...				
0x0400	DMA_DESC_ENABLE	7:0	W	Enable descriptors 7:0. One bit per descriptor. Cleared when Descriptor is handled.

Every descriptor has a set of registers (summarized in Table 5), with the following specific functions:

- DMA_DESC_x: the register containing the start (start address) and the end (end address) memory addresses of a DMA transfer; both handled by the server (software API).
- DMA_DESC_xa: integrates the information above by adding (i) the status of the read pointer on the server side (rd pointer), (ii) the wrap around functionality enabling (wrap around), (iii) the FromHost ("1") and ToHost ("0") transfer direction bit (read write), and (iv) the number of 32 bits words to be transferred (num words)
- DMA_DESC_STATUS: status of a specific descriptor including (i) wrap around information bits (even pc and even dma), (ii) completion bit (desc done, (iii) DMA pointer current address (current address)
- DMA_DESC_ENABLE: the descriptors enable register (dma desc enable), one bit per descriptor

In the MVTX software, the DMA is usually initialized through the `dma_to_host()` function of the FlxCard library, which is shown in Listing 3.

Endless DMA with a circular buffer and wrap around The DMA is configured in endless DMA mode where the DMA wraps around at the start address once the end address has been reached. In other words the DMA buffer is full, so if the DMA is not read out it will over write the data at the starting address. The functionality is enabled by setting the wrap around but to '1'.

```
typedef struct
{
  volatile u_long start_address;        /*  low half, bits  63:00 */
  volatile u_long end_address;          /*  low half, bits 127:64 */
  volatile u_long tlp          :11;     /* high half, bits  10:00 */
  volatile u_long read         : 1;     /* high half, bit      11 */
  volatile u_long wrap_around  : 1;     /* high half, bit      12 */
  volatile u_long reserved     :51;     /* high half, bits  63:13 */
  volatile u_long read_ptr;             /* high half, bits 127:64 */
} dma_descriptor_t;

typedef struct
{
  volatile u_long current_address;      /* bits  63:00 */
  volatile u_long descriptor_done : 1;  /* bit      64 */
  volatile u_long even_addr_dma   : 1;  /* bit      65 */
  volatile u_long even_addr_pc    : 1;  /* bit      66 */
} dma_status_t;
```

Listing 1: DMA descriptor structs in software (from FlxCard.h)

The server software provides another address name (PC readpointer) which indicates where the software last read out the memory. After wrapping around the DMA core will transfer To Host memory until the PC read pointer is reached. The server read pointer is updated more often than the wrap-around time of the DMA.

In order to determine whether Wupper is processing an address behind or in front of the server, Wupper keeps track of the number of wrap around occurrences. In the DMA status registers the even cycle bits displays the status of the wrap-around cycle. In every even cycle (starting from 0), the bits are 0, and every wrap around the status bits will toggle. The even pc bit flags a PC read pointer wrap-around, the even dma a Wupper wrap-around. By looking at the wrap-around flags the server software can also keep track of its own wrap-arounds. Note that while in the endless DMA mode (wrap around bit set), the PC read pointer is be maintained by the server software and kept within the start and end address range for Wupper to function correctly. The diagram below shows the two pointers racing each other, and the different scenarios in which they can be found with respect to each other.

Figure 69 shows Endless DMA buffer and pointers representation diagram in ToHost mode

- A : start condition, both the server and the DMA have not started their operation.
- B : normal condition, the PC read pointer stays behind the DMA's current address
- C : normal condition, the DMA's current address has wrapped around and has to stay behind the PC read pointer
- D : the server is reading too slow, the DMA is stalled because the server read pointer is not advancing fast enough, the DMA current address has to stay behind.

```
type dma_descriptor_type is record
  start_address   : std_logic_vector(63 downto 0);
  current_address : std_logic_vector(63 downto 0);
  end_address     : std_logic_vector(63 downto 0);
  dword_count     : std_logic_vector(10 downto 0);
  read_not_write  : std_logic;     --1 means this is a read descriptor, 0: write
   ↪ descriptor
  enable          : std_logic;     --descriptor is valid
  wrap_around     : std_logic;     --1 means when end is reached, keep enabled
   ↪ and start over
  evencycle_dma   : std_logic;     --For every time the current_address
   ↪ overflows, this bit toggles
  evencycle_pc    : std_logic;     --For every time the pc pointer overflows,
   ↪ this bit toggles.
  pc_pointer      : std_logic_vector(63 downto 0); --Last address that the PC has
   ↪ read / written. For write: overflow and read until this cycle.
end record;

type dma_status_type is record
  descriptor_done: std_logic;  -- means the dma_descriptor in the array above has
   ↪ been handled, the enable field should then be cleared.
end record;
```

Listing 2: DMA descriptor structs in firmware (from pcie_package.vhd). Note the division between "descriptor" and "status" differs from the software definition.

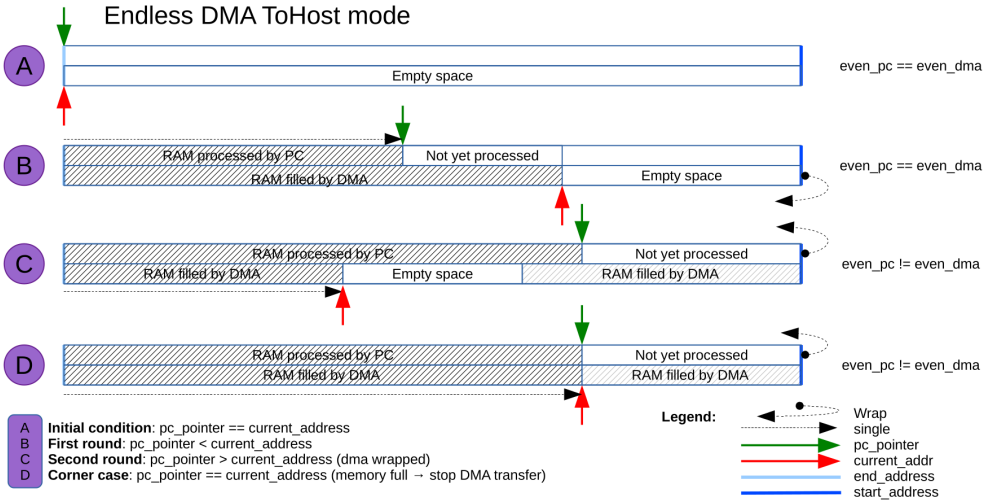


Figure 69: Endless DMA To Host mode

```
/*****************************************************************************/
void FlxCard::dma_to_host(u_int dma_id, u_long dst, size_t size, u_int flags)
/*****************************************************************************/
{
  DEBUG_TEXT(DFDB_FELIXCARD, 15, "FlxCard::dma_to_host: Method called with dma_id =
   ↪  " << dma_id << ", dst = 0x" << HEX(dst) << ", size = " << size << ", flags =
   ↪  " << flags);
  dma_stop(dma_id);

  m_bar0->DMA_DESC[dma_id].start_address = dst;
  m_bar0->DMA_DESC[dma_id].end_address   = dst + size;
  m_bar0->DMA_DESC[dma_id].tlp           = m_maxTLPBytes / 4;
  DEBUG_TEXT(DFDB_FELIXCARD, 20, "FlxCard::dma_to_host: m_bar0->DMA_DESC[" <<
   ↪  dma_id << "].tlp = " << m_maxTLPBytes / 4);
  m_bar0->DMA_DESC[dma_id].read          = 0;
  m_bar0->DMA_DESC[dma_id].wrap_around   = (flags & FLX_DMA_WRAPAROUND) ? 1 : 0;
  m_bar0->DMA_DESC[dma_id].read_ptr      = dst;

  if(m_bar0->DMA_DESC_STATUS[dma_id].even_addr_pc ==
   ↪  m_bar0->DMA_DESC_STATUS[dma_id].even_addr_dma)
  {
    // Make 'even_addr_pc' unequal to 'even_addr_dma', or a (circular) DMA won't
     ↪  start!?
    --m_bar0->DMA_DESC[dma_id].read_ptr;
    ++m_bar0->DMA_DESC[dma_id].read_ptr;
  }

  m_bar0->DMA_DESC_ENABLE |= 1 << dma_id;
  DEBUG_TEXT(DFDB_FELIXCARD, 15, "FlxCard::dma_to_host: DMA started");
}
```

Listing 3: DMA initialization command. This is called by dmaStart() in daq_device_felix, with arguments dma_to_host(_dma_index, _cmemPhysAddr, _buffer_size, FLX_DMA_WRAPAROUND).

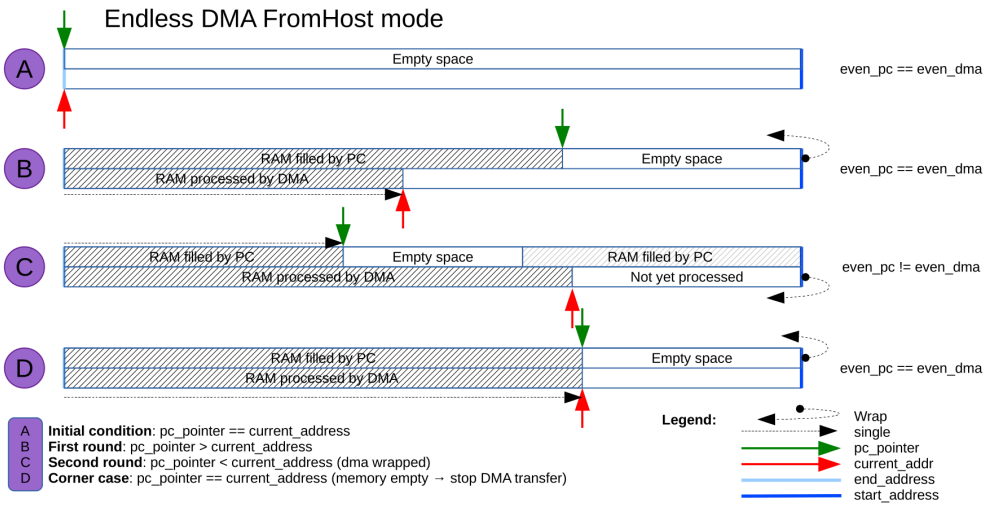


Figure 70: End less DMA From Host mode

Figure 70 Shows Endless DMA buffer and pointers representation diagram in FromHost mode

- A : start condition, both the server and the DMA have not started their operation.
- B : normal condition, the DMA's current address stays behind the PC read pointer
- C : normal condition, the PC read pointer has wrapped around and has to stay behind the DMA's current address
- D : the server is writing too slow, the DMA is stalled because the server read pointer is not advancing fast enough, the DMA current address has to stay behind.

4.4.2 Interrupts

The MSI-X Interrupt table contains eight interrupts; this number can be extended by a generic parameter in the firmware. Four of the interrupts, [0..3], are dedicated to Wupper, four interrupts, [4..7], are called from the logic implemented before the wupper. The interrupts are detailed in the Table below.

Table 6 shows Interrupts 0 to 3 are generated by `dma_control` when controlling the DMA controller and handling the descriptors. Interrupt 0 and 1 have the same functionality, but serve in the other direction (ToHost and FromHost). These interrupts fire when the END ADDRESS in the corresponding descriptor was reached. In endless DMA mode, interrupt 0 and 1 are set every time the END ADDRESS is reached, indicating that the DMA controller wraps back to START ADDRESS, but continues operation. Interrupt 2 is fired when enough data has arrived in the ToHost fifo to fill at least one TLP of data. Interrupt 3 is fired when the FromHost fifo reaches the prog full state, this is a usual thing to happen as the threshold of this fifo is set at a low level.

The server software does not need to take any action on this interrupt as the Wupper core will regulate itself. Interrupt 4 is a special test interrupt that can be fired by writing the interrupt test register in the register map. Interrupt 5 is fired when one of the channel (ELINK) fifo's becomes full, at this point also the XOFF bit is set on the lemo connector on the TTCfx v3 board. Interrupt 6 and 7 indicate the full flag (and prog full which has a lower threshold but has the same functionality) of the ToHost fifo was set.

Table 6: PCIe interrupts

Interrupt	Name	Description
0	FromHost wrap around	This interrupt is fired when the FromHost descriptor reaches the end address, or wraps around
1	ToHost wrap around	This interrupt is fired when the ToHost descriptor reaches the end address, or wraps around
2	ToHost Available	Fired when data becomes available in the ToHost FIFO (falling edge of ToHostFifoProgEmpty)
3	FromHost Full	Fired when the FromHost FIFO becomes full (rising edge of FromHostAppFifoProgFull)
4	Test interrupt #4	Fired when writing data to address BAR2 + 0x7800
5	crDownXoff	ToHost combined full flags (CR xoff)
6	ToHost Prog Full	Fired when the ToHost FIFO becomes filled above a programmable threshold
7	ToHost Full	Fired when the ToHost FIFO becomes full

4.4.3 Xilinx PCIe EndPoint Core AXI4-Stream interface

The interface has the advantage that it has two separate bidirectional AXI4-Stream interfaces. The two interfaces are the requester interface, with which the FPGA issues the requests and the PC replies, and the completer interface where the PC takes initiative. The Completer reQuest cq and Completer Completer are typically used for register access. The Requester reQuest and Requester Completer are used for the high speed DMA access.

bus	Description	Direction
`axis_rq`	**R**equester re**Q**uest. This interface is used for DMA, the FPGA takes the initiative to write to this AXI4-Stream interface and the PC has to answer.	FPGA → PC
`axis_rc`	**R**equester **C**ompleter. This interface is used for DMA reads (from PC memory to FPGA), this interface also receives a reply message from the PC after a DMA write.	PC → FPGA
`axis_cq`	**C**ompleter re**Q**uest. This interface is used to write the DMA descriptors as well as some other registers.	PC → FPGA
`axis_cc`	**C**ompleter **C**ompleter. This interface is used as a reply inteface for register reads, as well as a reply header for a register write.	FPGA → PC

Table 7: AXI4-Stream streams

completer interface under construction

Completer Interface

This interface maps the transactions (memory, I/O read/write, messages, Atomic Operations) received from the PCIe link into transactions on the Completer reQuest (CQ) interface based on the AXI4-Stream protocol. The completer interface consists of two separate interfaces, one for data transfers in each direction.The CQ interface is for transfer of requests (with any associated payload data) to the user application, and the Completer Completion (CC) interface is for transferring the Completion data (for a Non-Posted request) from the user application for forwarding on the link. The two interfaces operate independently. That

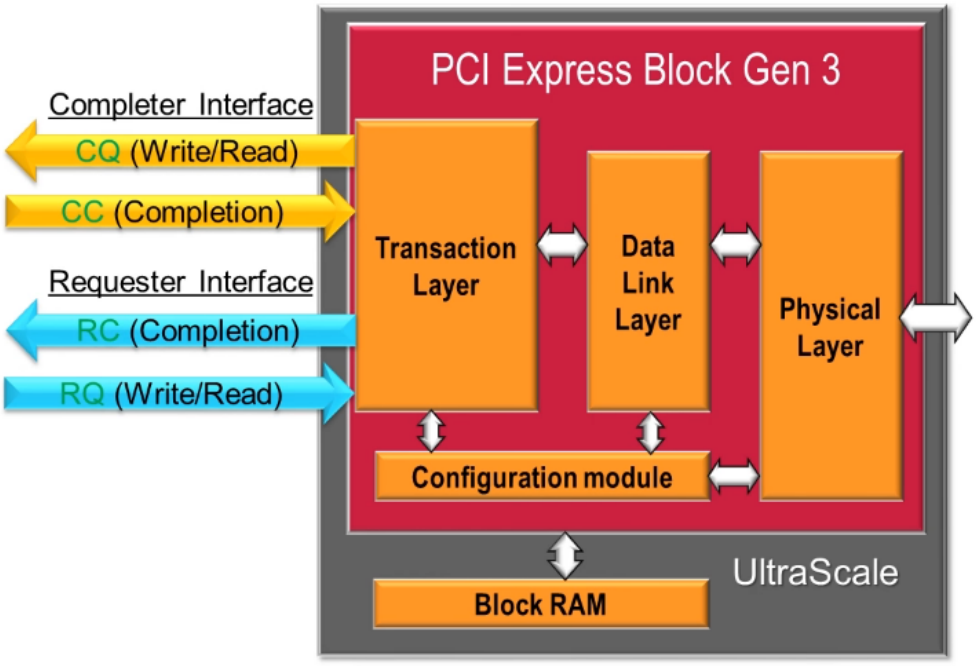


Figure 71: Xilinx PCIe CQ CC RC RQ

is, the integrated block can transfer new requests over the CQ interface while receiving a Completion for a previous request. ——————————————————————————————

A transaction layer packet (TLP) is transferred on each of the AXI4-Stream interfaces as a descriptor followed by payload data (when the TLP has a payload). The descriptor has a fixed size of 16 bytes on the request interfaces and 12 bytes on the completion interfaces.

On its transmit side (towards the link), the integrated block assembles the TLP header from the parameters supplied by the user application in the descriptor.

On its receive side (towards the client), the integrated block extracts parameters from the headers of received TLP and constructs the descriptors for delivering to the user application.

When a payload is present, Dword-aligned mode and Address-Aligned Mode are two options for aligning the first byte of the payload with respect to the datapath. The DMA uses Dword-aligned mode, the descriptor bytes are followed immediately by the payload bytes in the next Dword position, whenever a payload is present.

4.4.4 Firmware Components

The two main parts dma_control and dma_read_write are described above this section will highlight more detail regarding each block in Figure 74.

The Wupper Core is divided into two parts:

1. DMA Read Write This entity contains the process to parse the DMA descriptors and transfer the data from the FIFO to the AXI stream bus and vise versa.

The DMA read and write module handles the transfer from the FIFO‘s according to the direction speci-

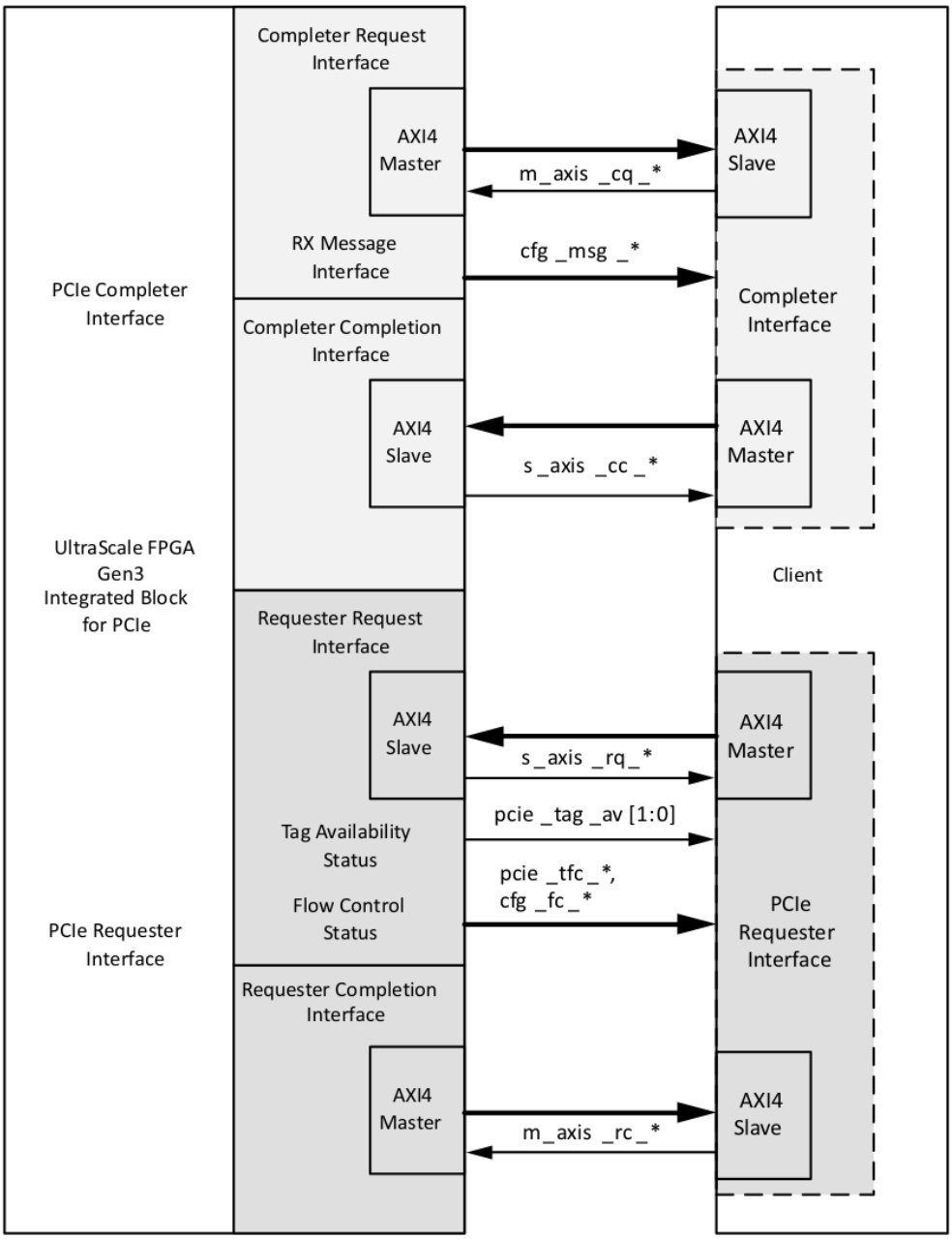


Figure 72: Block Diagram Client Interface: Left side is the PC/software interface, right side (client) is the FPGA/firmware interface

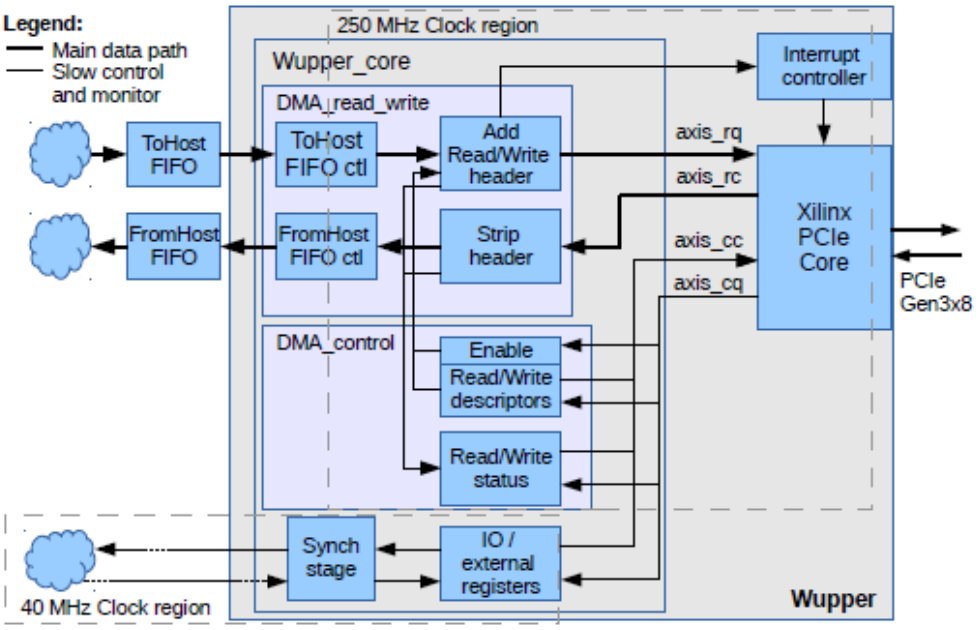


Figure 73: Wupper Block Diagram

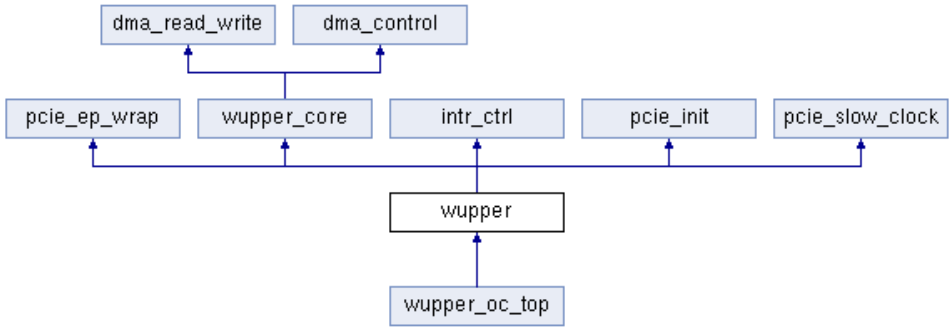


Figure 74: Wupper Inheritance Diagram

fied by the descriptors. If data shifts into the down FIFO, a non-empty flag will be asserted to start the DMA write process, this direction of the flow is defined as the "down link". This process reads the descriptors and creates a header with the information. The header is added when the data shifts out of the down FIFO. For the reversed situation, the data with a header is read from the PC memory. This direction of the flow is then defined as "up link". The information in the header will be parsed by the DMA control and the data fed to the up FIFO.

This DMA Read Write entity contains two state machines that control the AXI-Stream signals for the axis_rc Requester Completer (PC to FPGA) and axis_rq Requester reQuest (PC to FPGA) interfaces:

- ToHost / `add_header`: The first state machine drives the `axis_rq` interface to either request data from the PC (for a read-mode DMA descriptor) or send data to the PC (for a write-mode DMA descriptor). The descriptors are read and a header according to the descriptor is created. If the descriptor is a ToHost descriptor, the payload data is read from the FIFO and added after the header.
- FromHost / `strip_hdr`: The second state machine responds to the `axis_rc` interface, and processes the data received in response to a request sent by `add_header`. The header of the received data is removed and the length is checked; then the payload is shifted into the FIFO.

Figure 75 shows the DMA ToHost process reads the current descriptor and requests a read or write to the server memory. If the descriptor is set to ToHost, it also initiates a FIFO read and adds the data into the payload of the PCIe TLP (Transaction Layer Packet). When the descriptor is set to FromHost this process only creates a header TLP with no payload, to request a certain amount of data from the server memory that fits in one TLP.

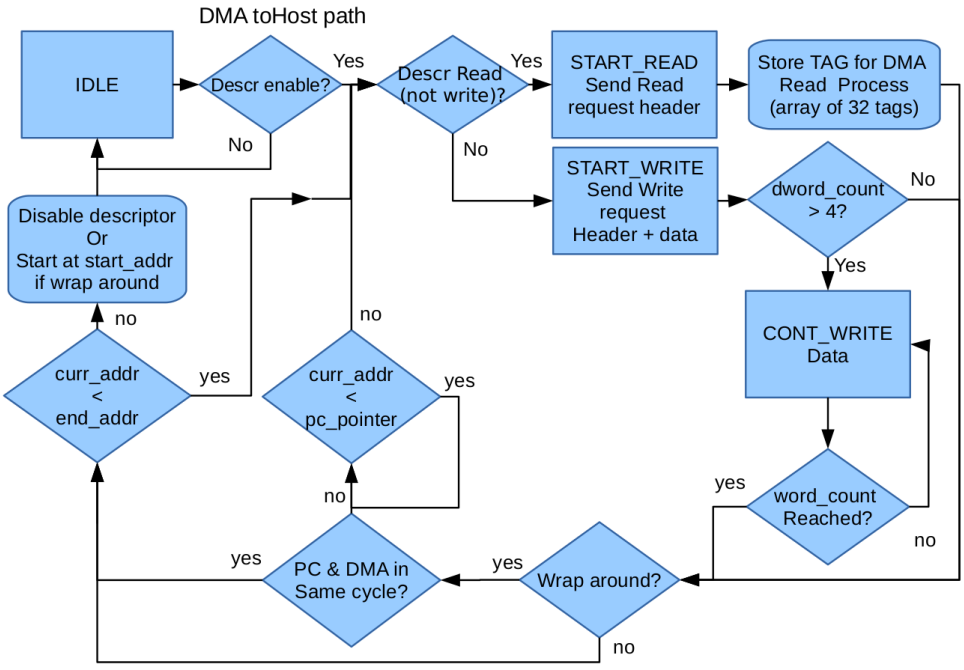


Figure 75: DMA flow for the axis_rq (to PC) interface. This is implemented partly in the add_header state machine of dma_read_write and partly in dma_control.

The add_header state machine, diagrammed in Figure 76, transitions out of IDLE state if axis_rq.tready is asserted by the PCIe endpoint, the active DMA descriptor is enabled (enable is set high), and the relevant FIFO is ready (if the descriptor is in read mode, the FIFO for data transfer to the user application is not full; if the descriptor is in write mode, the FIFO for data transfer from the user application is not empty). When this happens, the descriptor_done field of the descriptor is For a read-mode descriptor, the state machine transitions to START_READ state and drives the axis_rq interface is driven with a memory read transaction as shown in Figure 77. Only the descriptor is sent, so the transaction is always a single AXI transfer (tlast is immediately asserted). For a write-mode descriptor, the state machine transitions to START_WRITE state and drives the axis_rq interface is driven with a memory write transaction as shown in Figure 78. The START_WRITE state drives the first AXI transfer, which contains the descriptor and the first 128 bits of data (four dwords). If more data needs to be transferred, the state machine transitions to CONT_WRITE state and sends 256 bits of data in each AXI transfer.

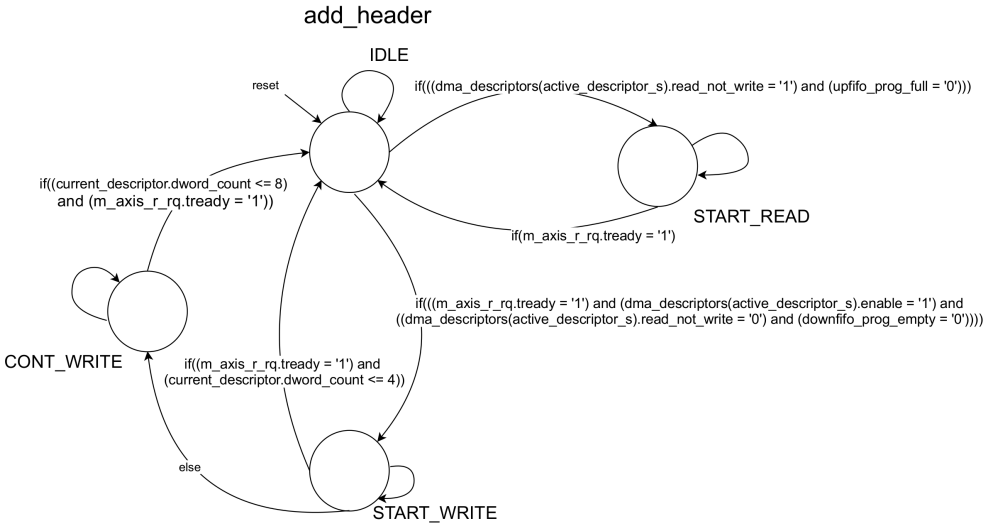


Figure 76: dma_read_write_ add_header

Figure 80 shows the DMA FromHost process checks the size of the payload against the size in the TLP header, the data will be pushed into the FromHost FIFO.

The strip_hdr state machine, diagrammed in Figure 81, transitions out of IDLE state and into PUSH_DATA state if axis_rc.tvalid is asserted by the PCIe endpoint. This initiates a data transfer as shown in Figure 82. The state machine reads the dword count from the descriptor (shown in Figure 83), and pushes data into the FIFO for data transfer to the user application until the correct number of dwords have been read or the PCIe endpoint asserts tlast (in correct operation, both things should happen at the same time).

2. DMA Control This is the entity in which the descriptors are parsed and fed to the engine, and where the status register of every descriptor can be read back through PCIe. DMA control contains a register map, with addresses to the descriptors, status registers and external registers for the user application. Depending on the address range of the descriptor, the pointer of the current address is handled by DMA Control and incremented every time a TLP completes. DMA Control also handles the circular buffer DMA if this is requested by the descriptor. DMA control contains a register map, with addresses to the descriptors, status registers and external registers for the user space register map.

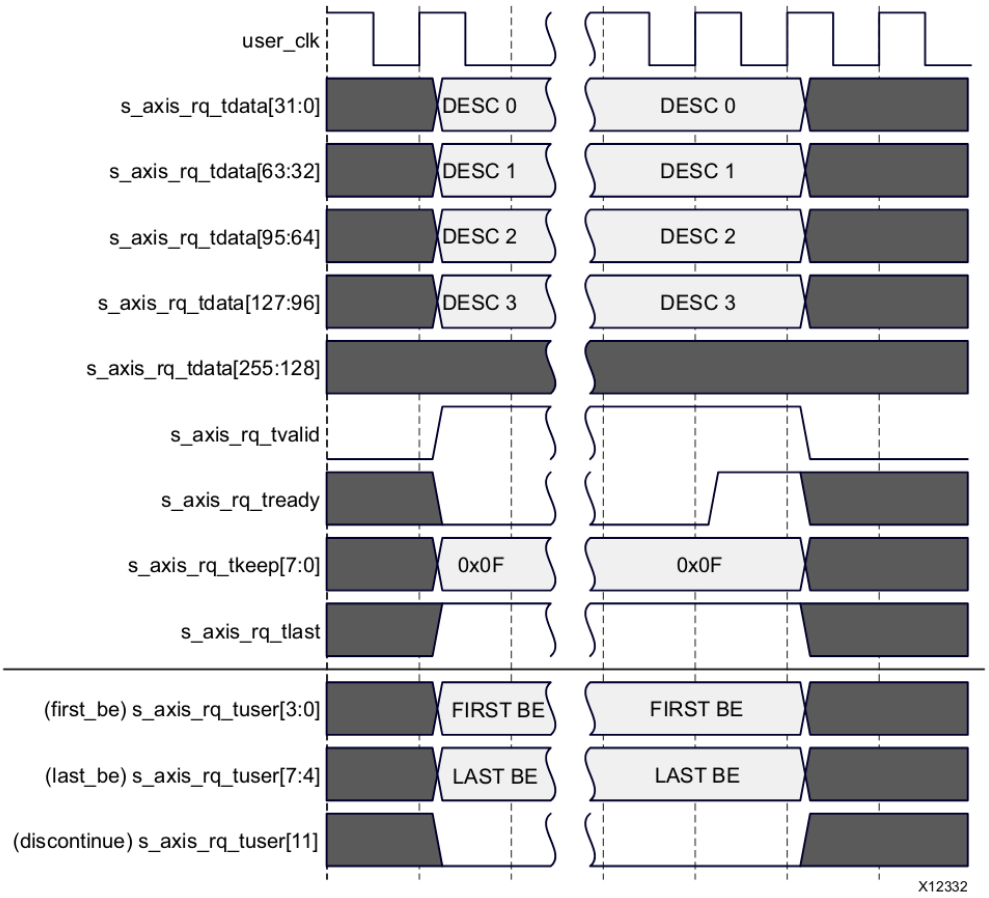


Figure 77: Memory Read Transaction on the Requester Request Interface (Dword-Aligned Mode, 256-Bit Interface)

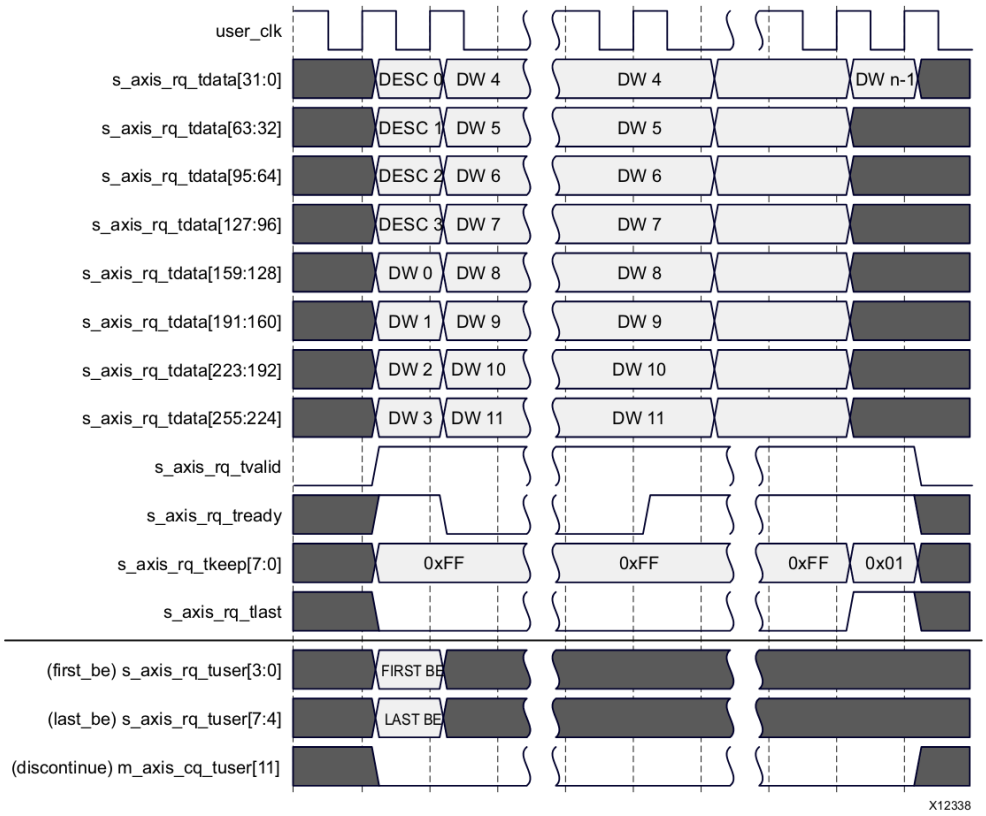


Figure 78: Memory Write Transaction on the Requester Request Interface (Dword-Aligned Mode, 256-Bit Interface)

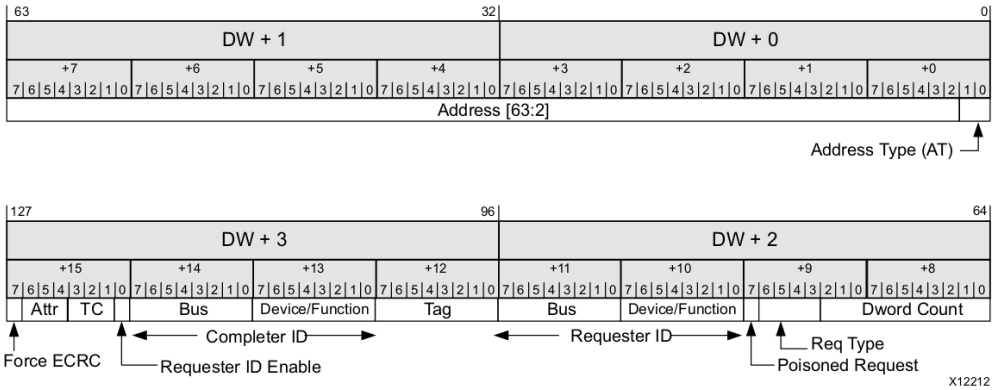


Figure 79: Requester request descriptor format. The request type is “0000” for a memory read and “0001” for a memory write. The address is the physical memory address of the first dword referenced by the request. The dword count is the number of 32-bit dwords to be read or written. The add_header state machine in Wupper sets all other fields to 0.

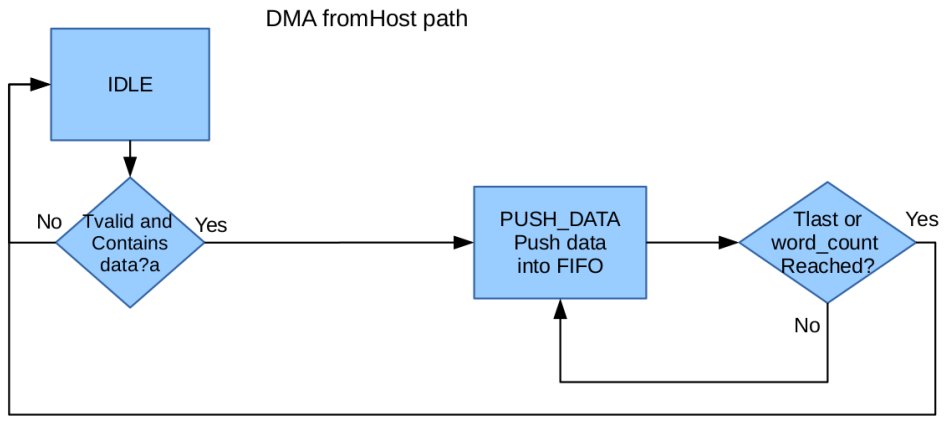


Figure 80: DMA flow for the axis_rc (to FPGA) interface. This is implemented in the strip_hdr state machine of dma_read_write.

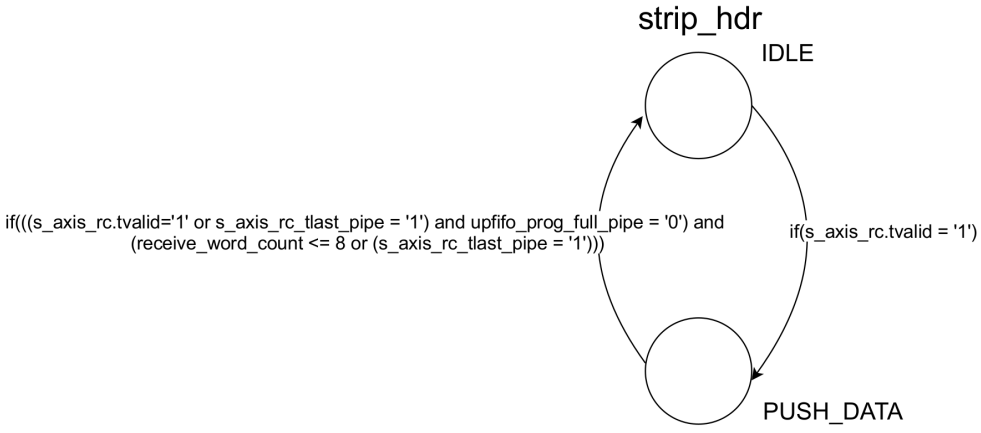


Figure 81: dma_read_write_ strip_header

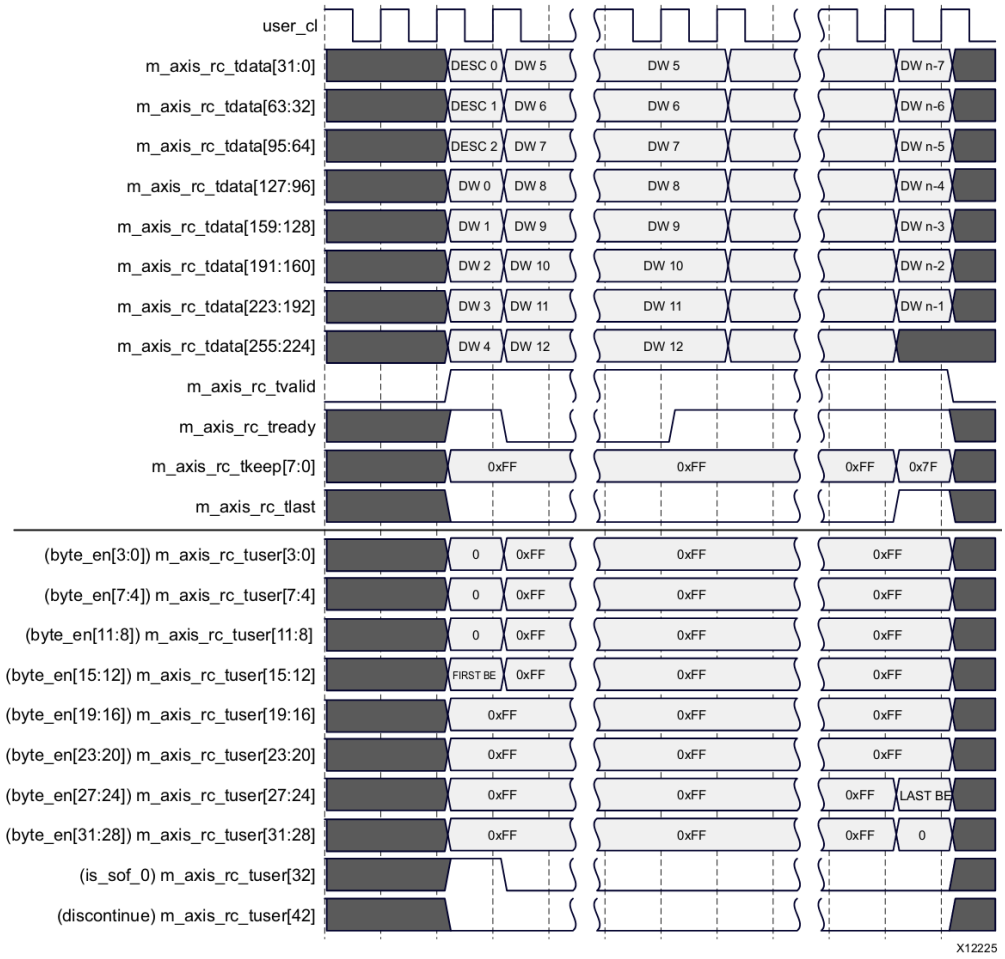


Figure 82: Transfer of a Completion with Data on the Requester Completion Interface (Dword-Aligned Mode, 256-Bit Interface)

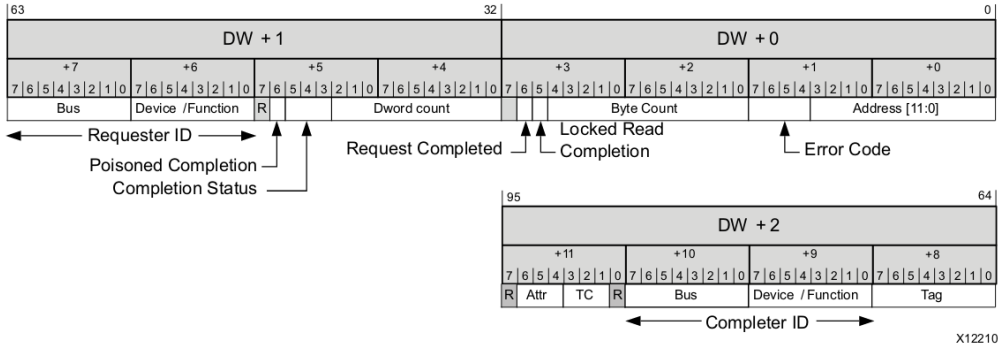


Figure 83: Requester completion descriptor format. Wupper reads the dword count, which is the number of 32-bit dwords in the AXI packet. The strip_hdr state machine in Wupper ignores all other fields.

The DMA Control process always responds to a request with a certain req type from the server. It responds only to IO and Memory reads and writes; for all other request types it will send an unknown request reply. If the data in the payload contains more than 128 bits, the process will send a "completion abort" reply and go back to idle state. The maximum register size has been set to 128 bits because this is a useful maximum register size; it is also the maximum payload that fits in one 250 MHz clock cycle of the AXI4-Stream interface.

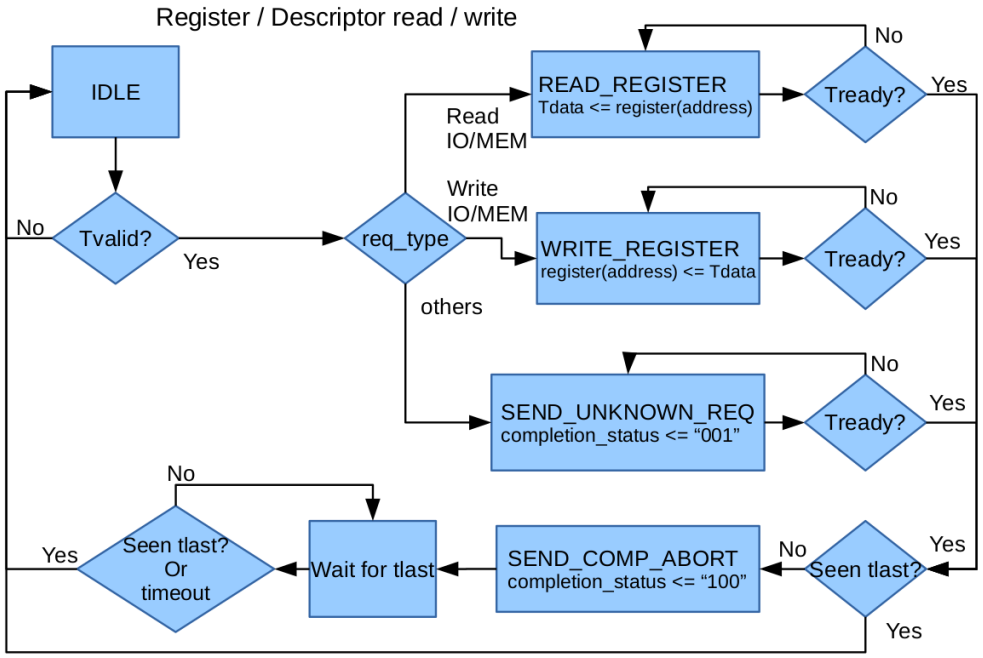


Figure 84: Flow of a register read or write. This is implemented in dma_control.

One part of `dma_control` (inside the `comp` process) is responsible for updating the descriptor status, as shown in Figures 69 and 70, in response to DMA reads and writes requested by `dma_read_write`.

Several `process` statements maintain the register map. Reads and writes to the register map are handled in the process `regrw`, which is clocked at `clkDiv6` (1/6 of the 250 MHz PCIe endpoint clock, therefore nominally 41.7 MHz). Since the state machine runs at the full PCIe endpoint clock, two processes handle the synchronization between clock domains. `regSync40` translates the read/write requests from the PCIe endpoint clock to `clkDiv6`, and `regSync250` translates the data and handshake responses to the PCIe endpoint clock. The `DESCRIPTOR_ENABLE` register is a special case: it is written on the PCIe endpoint clock but read at `clkDiv6`.

The state machine in this unit (diagrammed in Figure 85) responds to the `axis_cq` Completer reQuest (PC to FPGA) interface and drives the `axis_cc` Completer Completer (FPGA to PC) interface. The idle state identifies the incoming descriptor following the Completer Request Descriptor Format for Memory from pg156 Figure 86; the transaction follows the timing diagrams of Figure 87 (for a register write) or Figure 88 (for a register read). If the request type is a memory or I/O read or write, the state machine transitions to the `READ_REGISTER` or `WRITE_REGISTER_READ` state as appropriate. In the `READ_REGISTER` state, the state machine requests a read from the register map and packs the requested register data onto the `axis_cc` interface as shown in Figure 89. The state machine transitions to the `WAIT_RW_DONE` state when

the register map responds and the server asserts `axis_cc.tready`. In the `WRITE_REGISTER_READ` state, the state machine requests a read from the register map[1], and transitions to the `WRITE_REGISTER_MODIFYWRITE` state when the register map responds. In the `WRITE_REGISTER_MODIFYWRITE` state, the new value of the register is written to the register map, and the completion is packed onto the `axis_cc` interface as shown in Figure 89. The state machine transitions to the `WAIT_RW_DONE` state when the register map responds and the server asserts `axis_cc.tready`. In the `WAIT_RW_DONE` state, the state machine waits until the register map deasserts the `done` signal that was asserted in the previous state, then the state machine returns to `IDLE` state. The `axis_cq.tready` signal is deasserted in all states other than `IDLE`, blocking the PCIe endpoint from putting additional transactions on the `axis_cq` interface until the current one has been handled.

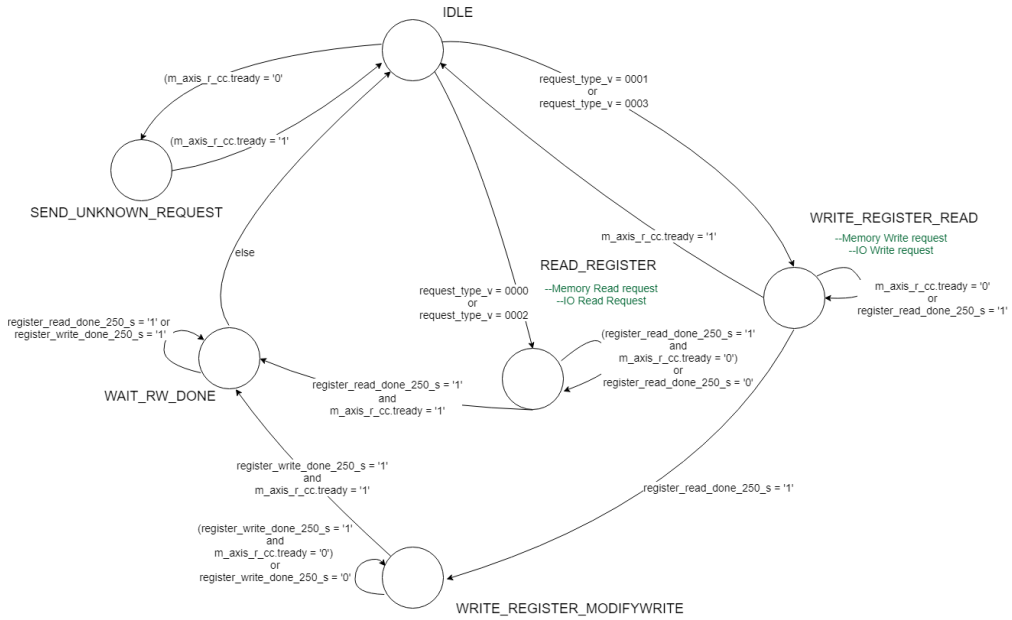


Figure 85: dma_control completer

pcie_ep_wrap.vhd Wrapper unit for the Xilinx UltraScale Gen3 Integrated Block for PCI Express v4.1 IP, and the clock generator

intr_ctrl.vhd Implements the creation of MSIx interrupts. It may be triggered by either of the input bits in interrupt_call or dma_interrupt_call.

pcie_ctl.vhd Contains a process to initialize some registers in the PCI express Gen3 core. Additionally it reads the BAR0 BAR1 and BAR2 registers and outputs their values to be used by dma_control.

pcie_slow_clock.vhd Creates a slow clock of 40 MHz (41.667) by dividing the 250MHz clock by 6.

The file /etc/init.d/drivers_flx allocates a 4 G space for the data. By passing the gfpba_size to the cmem driver. You can make a larger memory space by changing the gfpbpa_size parameter.

Each PCIe instance of Wupper takes 1G of the 4 G space by default. You can get each instance to allocate more by running fdaq with flag -b and desired buffer size.

A control buffer is also allocated that stores various control information for the descriptor.

[1]This is necessary since the write may only affect part of the 128-bit register. To avoid changing the remainder of the register, the current value must first be read.

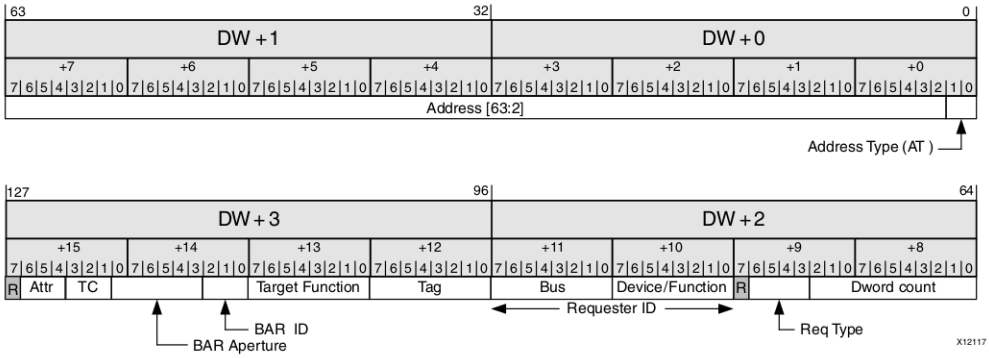


Figure 86: Completer Request Descriptor Format for Memory I/O

```
_fifo_din <= x"DEADBEEF"&cnt&x"00001111"&cnt&x"22223333"&cnt&x"44445555"&cnt;
```

Listing 4: simulation commands

```
address_type_s       <= s_axis_cq.tdata(1 downto 0);
register_address_s   <= s_axis_cq.tdata(63 downto 2)&"00";
dword_count_s        <= s_axis_cq.tdata(74 downto 64);
request_type_v       := s_axis_cq.tdata(78 downto 75);
request_type_s       <= request_type_v;
requester_id_s       <= s_axis_cq.tdata(95 downto 80);
tag_s                <= s_axis_cq.tdata(103 downto 96);
target_function_s    <= s_axis_cq.tdata(111 downto 104);
bar_id_s             <= s_axis_cq.tdata(114 downto 112);
bar_aperture_s       <= s_axis_cq.tdata(120 downto 115);
transaction_class_s  <= s_axis_cq.tdata(123 downto 121);
attributes_s         <= s_axis_cq.tdata(126 downto 124);
register_data_s      <= s_axis_cq.tdata(255 downto 128);
```

Listing 5: simulation commands

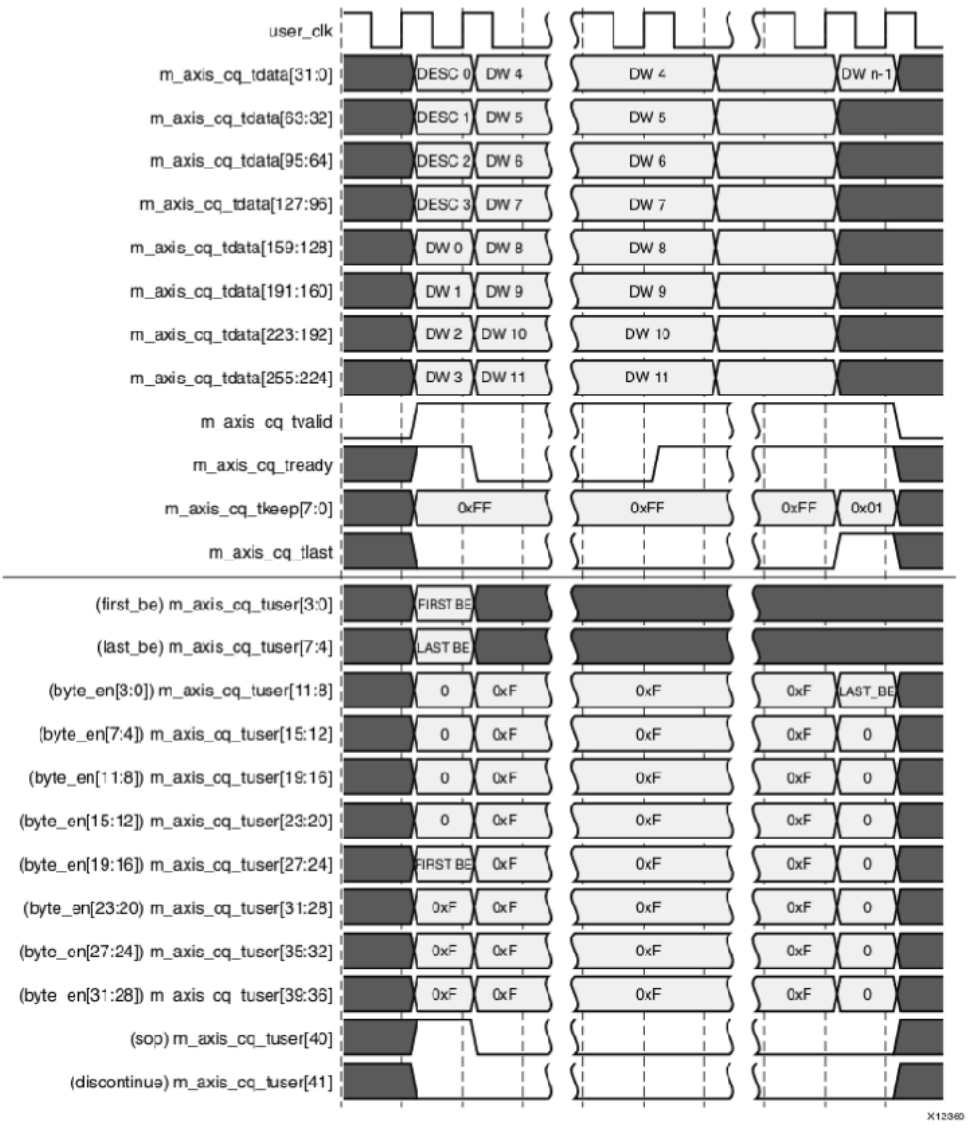


Figure 87: Memory Read Transaction on the Completer Request Interface (Dword-Aligned Mode, Interface Width = 256 Bits)

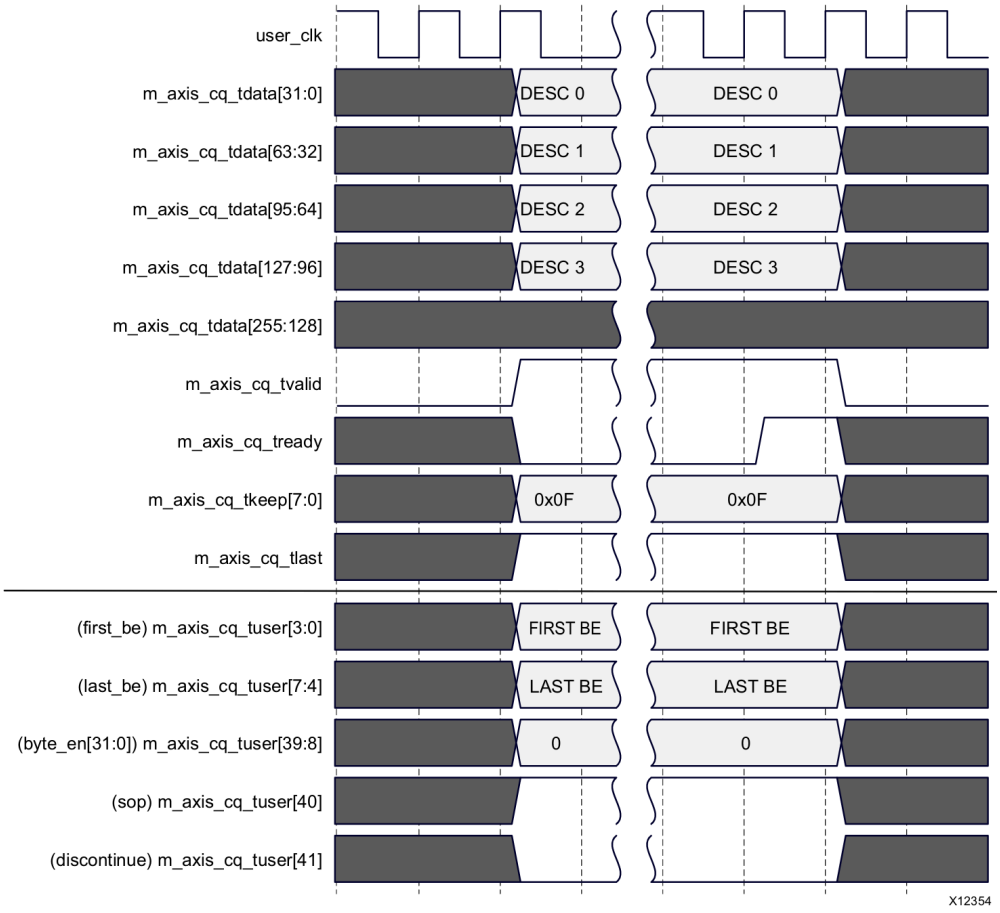


Figure 88: Memory Read Transaction on the Completer Request Interface (Dword-Aligned Mode, Interface Width = 256 Bits)

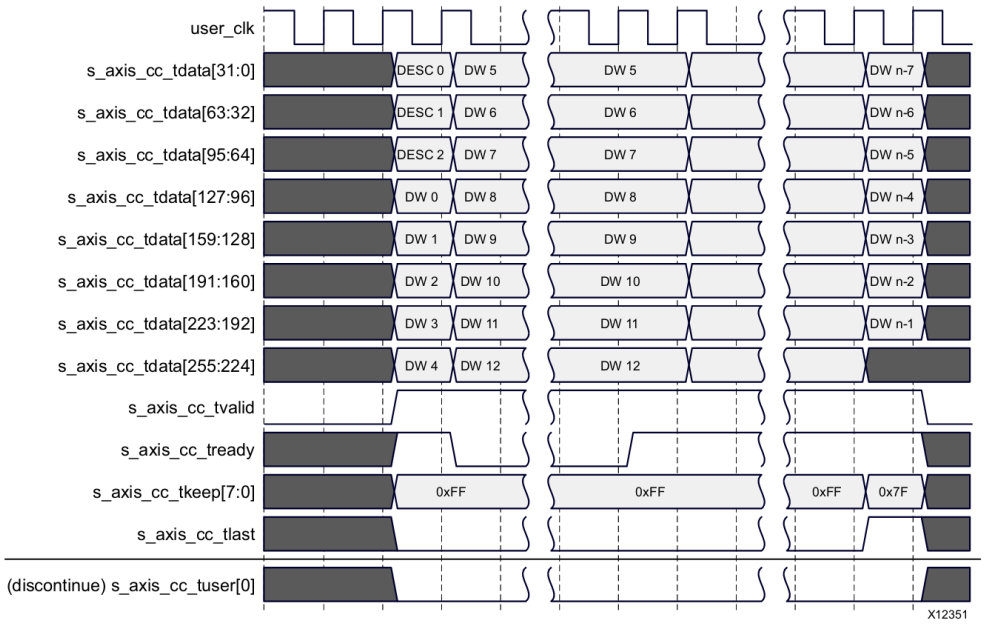


Figure 89: Transfer of a Normal Completion on the Completer Completion Interface (Dword-Aligned Mode, 256-Bit Interface)

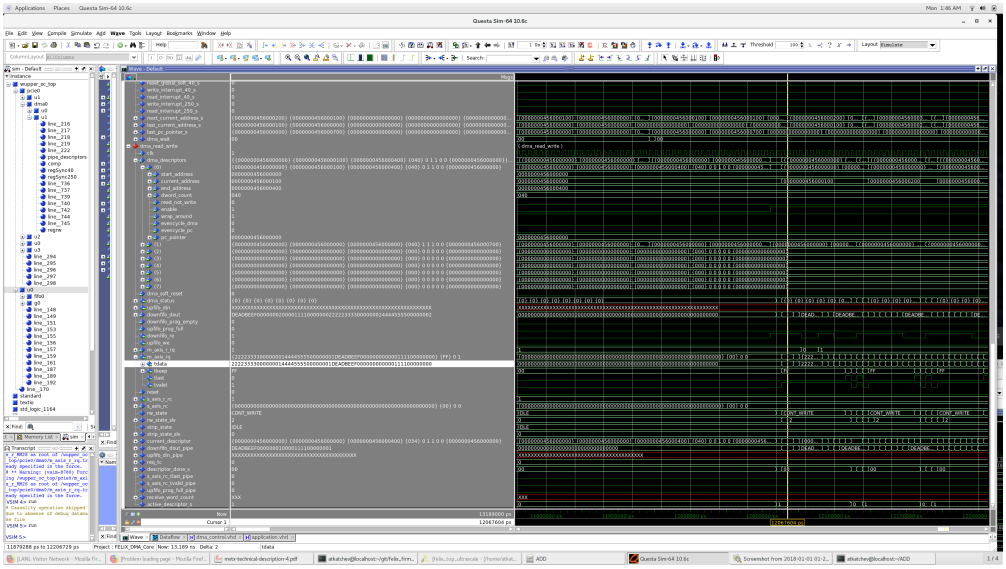


Figure 90: Simulation results

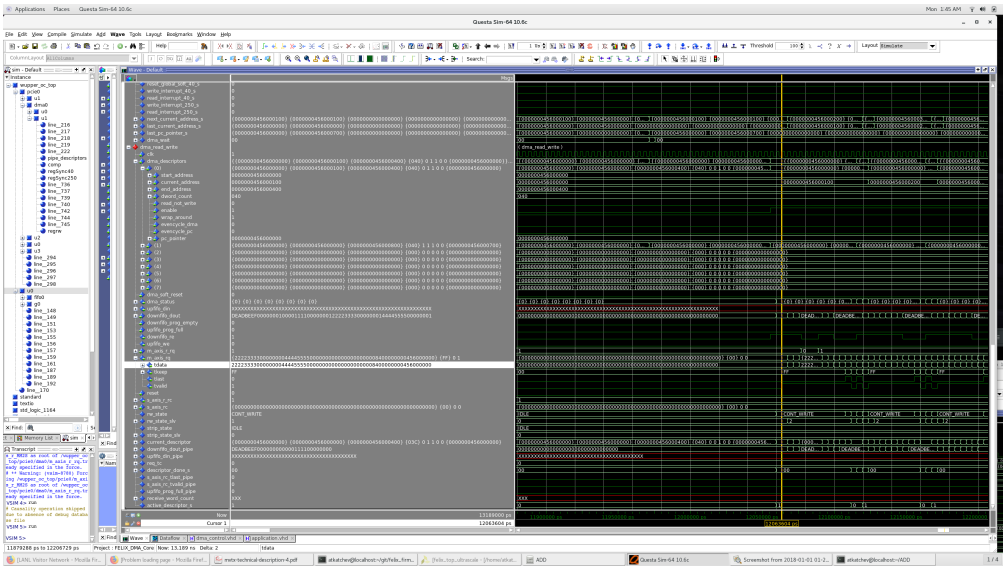


Figure 91: Simulation results

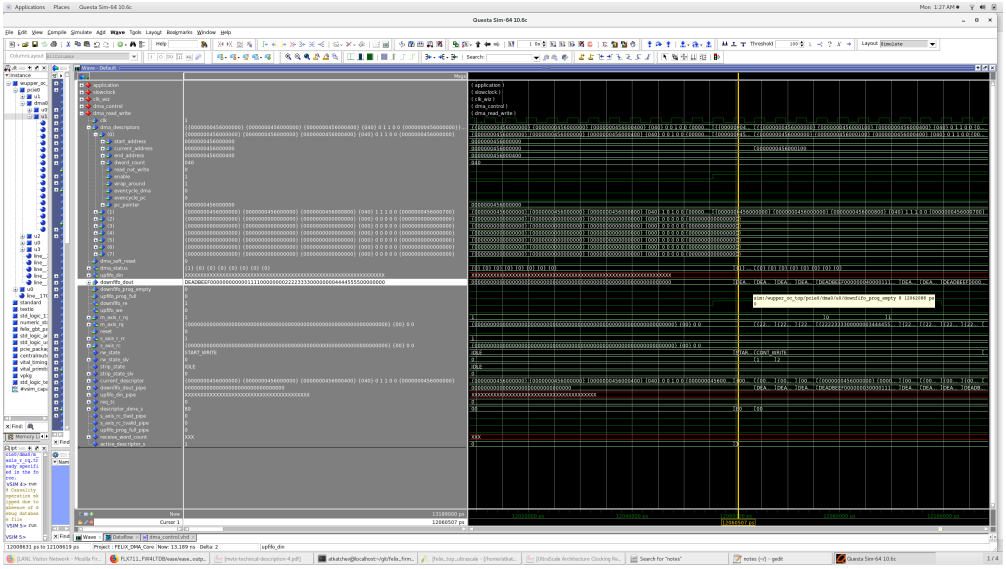


Figure 92: Simulation results

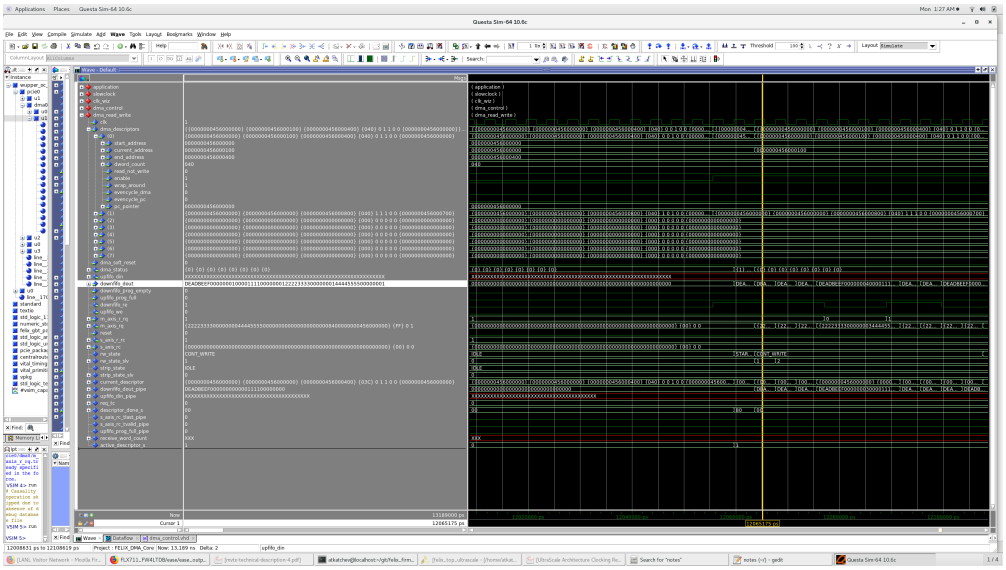


Figure 93: Simulation results

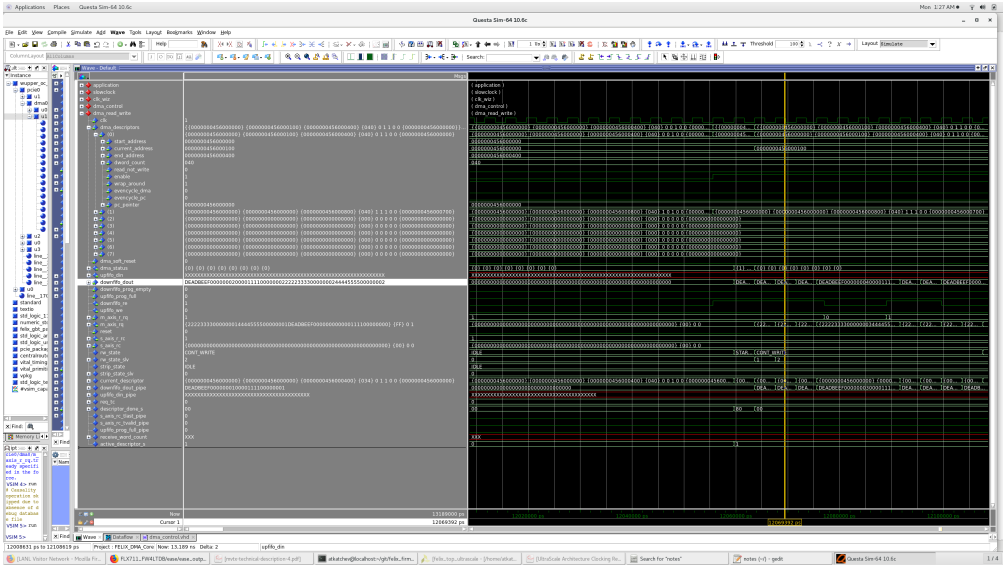


Figure 94: Simulation results

PCIe0 and PCIe1 are aliased in firmware to the two Wupper cores

/etc/init.d/drivers_flx - Change gfpba_size and then reboot
flx-dma-stat -d 0 - Verifies the PCIe0 (Wupper instance)
flx-dma-stat -d 1 - Verifies the PCIe1 (Wupper instance)

cat /proc/mem_rcc
cat /etc/init.d/drivers_flx

Fdaq -d 0 -b 2048 Selects PCIe0 (Wupper instance 0), sets 2Gigs of the 4 Address Space
Fdaq -d 1 -b 0248 Selects PCIe1 (Wupper instance 1), sets 2G of the 4G Address Space

check this insmod felix.ko? gfpbpa_size 2048

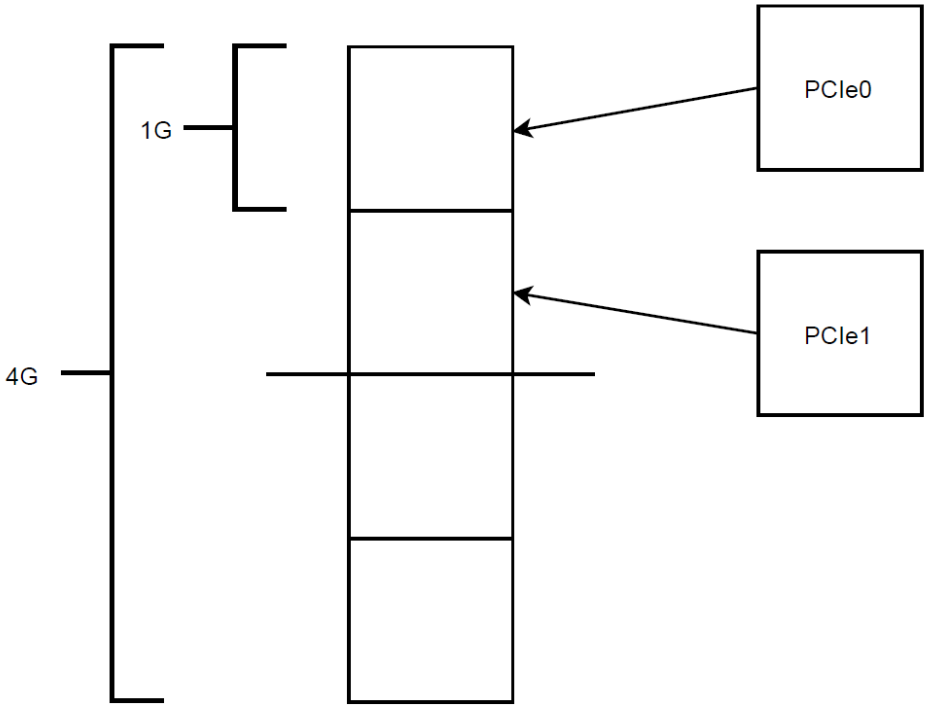


Figure 95: Default Descriptor

Simulation

The directory firmware/simulation/pcie dma top contains all necessary files to run the simulation in Mentor Graphics Modelsim or Questasim The directory contains a file modelsim.ini with some standard information, it is assumed that one have the Xilinx Unisim VCOMPONENTS library compiled and the location is defined in the environment variable $XILLIB.

Also the Library “work” has to be created in the project directory. The simulation project also relies on a simulation model of the FIFOcore, which will be generated when the cores in the Vivado project are

generated. The file that should be generated is ../../Projects/pcie dma top/pcie dma top.srcs/sources 1/ip/fifo generator 0 /fifo generator 0 funcsim.vhdl

Like the Vivado project, also the Questasim project is generated and operated using .tcl scripts. To create and run the project execute the following commands from the Questasim console:

```
cd firmware/simulation/Wupper/
#Create the project:
do project.do
#Start the simulation and load the waveforms:
do VSim_Functional.tcl
#Add stimuli to the AXI bus
do start.do
run 1us
```

Listing 6: simulation commands

4.5 Device Drivers

The FELIX server software consists of four primary parts. 1) cmem_rcc.c, a device driver that allocates large contiguous data buffers for data transfers from PCIe card interface to the host DMA (Direct Memory Access). 2) flx.c, device driver for the FELIX PCIe interface. 3) FlxCard.cpp the user space library that interfaces into the flx.c device driver, as well as serve as the configuration interface to various I2C and SPI devices on the FELIX hardware. 4) daq_device_felix.cc interface between sPHENIX Data acquisition RCDAQ and low level FELIX driver driver software (cmem_rcc.c and flx.c). The drivers are stored in felix/software/drivers_rcc/src.

Another device driver is also available but not used in since it was primarily developed for debugging and testing purposes, this is the io driver, derived from the io rcc driver, which provides access to PCIe configuration and BAR registers and can also be used to write to and read from the buffer allocated by cmem.

For loading the drivers (root privileges required) cd to DIRECTORY and type ./drivers felix start. The script also creates the device files /dev/felix/, /dev/cmem and /dev/io. ./felix stop results in unloading of the drivers and removal of the device files. After loading the drivers for each driver there is a file in the /proc filesystem: /proc/felix, /proc/cmem and /proc/io respectively. The driver associated with the file is invoked upon reading from one of these files and will output some information. With a “cat” shell command this output can be displayed. Using the “echo” command, e.g..: echo “debug” > /proc/felix these files can be used for controlling the driver, in the example the debug mode is switched on. Upon loading and unloading the drivers output some log information, which can be inspected by means of the “dmesg” command.

Driver loading and unloading A driver can be loaded as follows, assuming that the user has the necessary sudo rights: cd driver_location sudo insmod ./driver_name.ko

Unloading the driver, in case of the felix driver (use rmmod cmem or rmmod io for the cmem or io driver) is done in this way: sudo rmmod felix

Device Drivers Common Methods This section will describe the methods in common between the cmem_rcc.c and flx.c device driver. Figure 96 shows a block diagram of how the two low level device drivers communicate with FELIX as well as the rcdaq plugin daq_device_felix.cc. The second point that the diagram tries to depict is how the data is moved between kernel and user space by way of the file operations struct. The

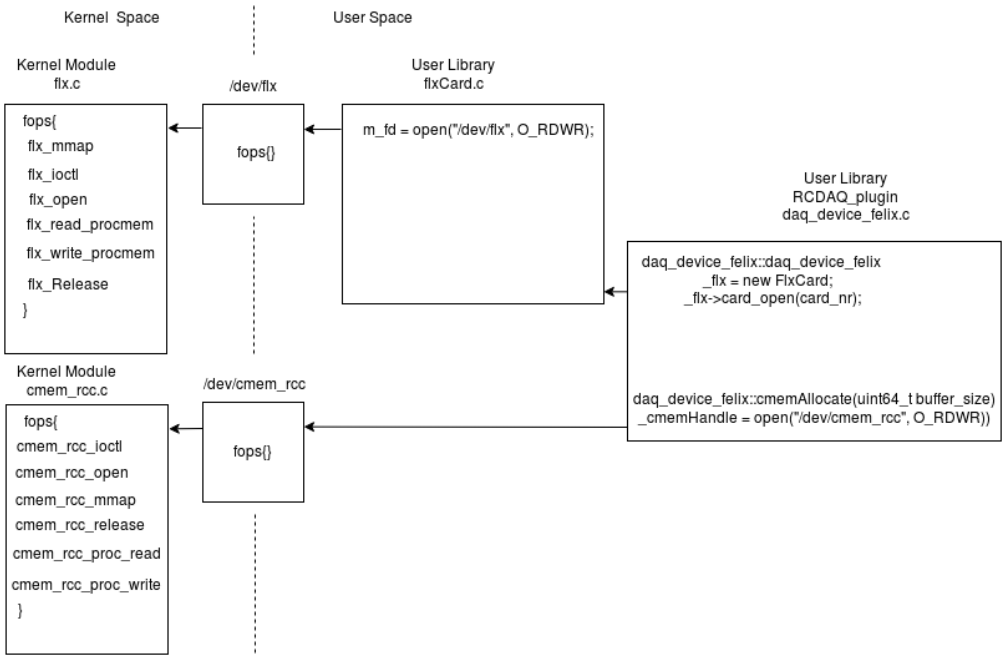


Figure 96: FELIX Software block Diagram

file operations struct is how the driver sets up the connections to device numbers. The struct is defined in <linux/fs.h>, and is a collection of pointers. Each open file represented by the file struct is associated with its own set of functions (by including a field called f_op that points to a file_operations structure). The operations mostly implement system calls and are therefore, named open, read, and so on. Consider it to be an "object" and the functions operating on it to be its "methods."

```
static struct file_operations fops =
{
  .owner          = THIS_MODULE,
  .unlocked_ioctl = cmem_rcc_ioctl,
  .open           = cmem_rcc_open,
  .mmap           = cmem_rcc_mmap,
  .release        = cmem_rcc_release,
  .read           = cmem_rcc_proc_read,
  .write          = cmem_rcc_proc_write,
};
```

Listing 7: struct file_operations for cmem_rcc.c

For interaction with the driver by a user program is to first obtain a file descriptor by means of an open system call for either /dev/felix, u_int FlxCard::number_of_cards(void) in FLXCard.cpp. /dev/cmem, daq_device_felix::cmemAllocate in daq_device_felix.cc.

module_init() and module_exit are kernel macros that inform the kernel to invoke these functions once

```
struct file_operations fops =
{
  .owner          = THIS_MODULE,
  .mmap           = flx_mmap,
  .unlocked_ioctl = flx_ioctl,
  .open           = flx_open,
  .read           = flx_read_procmem,
  .write          = flx_write_procmem,
  .release        = flx_Release,
};
```

Listing 8: struct file_operations for flx.c

```
fd = open("/dev/flx", O_RDWR);
_cmemHandle = open("/dev/cmem\_rcc", O_RDWR
```

Listing 9: Obtain File Descriptor

the module is loaded or removed using insmod/modprobe command. The init macro adds a section to the modules object code where the module's initialization function is to be found. The init functions also contains kernel registration function. The module_exit macro informs the kernel of the location of the clean up code. The two functions called by the kernel macros are described below.

The kernel module can accept parameters from the command line using insmod/modprobe. The parameters are declared using the mode_param_macro which is defined in moduleparam.h. module_param which is in both device drivers (example in Listing 10) takes three parameters: the name of the variable, its type, and a permissions mask to be used for an accompanying sysfs entry.

```
module_param (debug, int, S_IRUGO | S_IWUSR);
MODULE_PARM_DESC(debug, "1 = enable debugging   0 = disable debugging");
```

Listing 10: struct fmodule params

Contiguous Memory Allocation driver cmem_rcc.c is the Linux device driver and library that provide the user with means of allocating buffers of contiguous memory for DMA operations. The driver supports __get_free_pages system call, in which memory is allocated from the pool managed by Linux (available on any Linux kernel). The disadvantage is that __get_free_pages cannot allocate memory buffers above a certain size. For 2.4 kernels this limit is 2 MB. The driver also supports BigPhysArea patch (BPA), which is an extension of the functionality of the kernel not included in the Linux distribution. The CERN Linux support team has decided to includes BPA into all CERN kernels. The main advantage is that BPA does not limit the size of a contiguous buffer but some memory has to be put aside at boot time (by means of a

```
module_param (gfpbpa_size, long, S_IRUGO | S_IWUSR);
MODULE_PARM_DESC(gfpbpa_size, "The amount of RAM in MB that will be used for the
 ↪  internal-BPA (get_free_pages variant)");

module_param (gfpbpa_quantum, int, S_IRUGO | S_IWUSR);
MODULE_PARM_DESC(gfpbpa_quantum, "The size (in MB) of a page allocated via
 ↪  get_free_pages for the internal-BPA (get_free_pages variant)");
```

Listing 11: module param used to pass arguments to driver using insmod call

kernel command line parameter).If BPA is not installed, cmem_rcc can still be used for memory allocation via __get_free_pages. For MVTX, cmem_rcc is configured to use `__get_free_pages`.

The number of buffers (of either type) is limited by the size of some tables in the driver. The dimension of these tables is set by the parameter MAX_BUFFS in cmem_rcc_drv.h. If the default value (currently 1000) is not sufficient it can be increased to the required number of buffers. A change to this parameter requires a re-compilation of the entire package. This function static int cmem_rcc_init passes the "size" parameter to the driver where it gets converted to an "order" which is defined as the base two logarithm of the number of pages to be allocated. The default page size of Linux is 4KB. The maximum value of order for 2.4 and 2.6 kernels is 9 (corresponding to 512 pages = 2 MB). On 2.6 kernels the command "more /proc/buddyinfo" tells you how many blocks of each order are available. The conversion from size to order guarantees that the buffer will at least be size bytes long; it may, however, be larger. The function will fail for size > 2 MB but can also fail for smaller values if a contiguous buffer of the requested size can not be found.

Information about allocated buffers is maintained in two places. First, the static array `buffer_table` in `cmem_rcc` contains the detailed information on the memory addresses and sizes of all buffers that have been allocated. Second, whenever the `/dev/cmem_rcc` device file is opened, the `private_data` field of the file descriptor contains a list of buffer table indices; this is the list of buffers that is associated with the calling program. This list is initialized in `cmem_rcc_open` and is updated whenever buffers are requested or released through ioctl calls. When the file descriptor is released (typically when the calling program terminates), this list is used (via `cmem_rcc_release`) to clean up all buffers that were allocated for that program and are no longer in use.

static int cmem_rcc_init(void) - allocates 1MB of virtual memory by using ioremap function which builds a new page table. The function returns a special virtual address that can be used to access the specified physical address range. The function also calls gfpbpa_init which allocates a contiguous buffer by means of the __get_free_pages system call. The function also dynamically requests a major numbers using alloc_chrdev_region() function. Lastly, the function allocates memory for a buffer table and sets all entries to 0 and registers the using the cdev_add function.

static void cmem_rcc_cleanup(void) - closes the package and releases the access to the device file. Segments that have not been freed will be de-allocated unless they have been locked.

static int cmem_rcc_open(struct inode *inode, struct file *file) - This is called when the `/dev/cmem_rcc` device file is opened. Reserves and initializes space to store a list of memory buffers. This list is stored in

the `private_data` field of the file descriptor.

static int cmem_rcc_release(struct inode *inode, struct file *file) - This is called when a file descriptor for `/dev/cmem_rcc` is released. The list of buffers associated with this file descriptor is read from the descriptor, and all of those buffers are freed.

static long cmem_rcc_ioctl(struct file *file, u_int cmd, u_long arg) - case statement which evaluates the value of the command that is passed in.

CMEM_RCC_GET - used to retrieve the parameters of a buffer. The structure cmem_rcc_t is defined in cmem_rcc_common.h it is mapped to uio_desc which is used by the copy_from_user system call. The system call is described in asm/uaccess.h and moves data to and from user space by way of kernel space for virtual memory. The system call is very similar to memcopy where the first parameter is the destination, the second is the source, and the third is the number of bytes that will be sent. The function takes in cmem_rcc_t described below as the destination and the _cmemDescriptor from the daq_device_felix.cc plugin. The interrupt state is saved and the buffer table is reserved. The plugin passes TYPE_GFPBPA therefore the else if clause will execute and the function in cmem_rcc.c membpa_alloc_pages is called. Since the device driver is doing DMA it has to talk to hardware connected to the interface bus which in this case is PCIe and uses physical addresses and the program code has to use virtual addresses therefore, the virt_to_bus system call is used which performs a simple conversion between kernel logical addresses and bus addresses. virt_to_page takes a kernel logical address and returns its associated struct page pointer.

```
typedef struct
{
unsigned int paddr;          //The physical address the buffer
unsigned int uaddr;          //The used virtual address the buffer
unsigned long kaddr;         //The kernel virtual address the buffer
unsigned int size;           //The size of the buffer
unsigned int order;          //The encoded size of GFP buffers
unsigned int locked;         //A flag indicating if the buffer is locked
unsigned int type;           //The type of the buffer (TYPE\_BPA or TYPE\_GFP)
unsigned int handle;         //The segment identifier
char name[CMEM_MAX_NAME];    //The name of the buffer
{ cmem_rcc_t;
```

Listing 12: struct cmem_rcc_t

CMEM_RCC_SETUADDR - takes a user address passed in through the ioctl call, and records it in the buffer table. The source for this ioctl call is shown in Listing 13.

CMEM_RCC_FREE - returns a buffer to the pool of free memory.

CMEM_RCC_LOCK - locks a buffer. Once a buffer is locked it can no longer be freed with the CMEM_BPASegmentFree() or CMEM_BPASegmentFree() functions. It also does not get de-allocated by the garbage collector in the driver if the application that created the buffer exits. Usually buffers should not

```
copy_from_user(&uio_desc, (void *)arg, sizeof(cmem_rcc_t))
// Check if the handle makes sense
      if (buffer_table[uio_desc.handle].used == 0)
buffer_table[uio_desc.handle].uaddr = uio_desc.uaddr;
```

Listing 13: CMEM_RCC_SETUADDR

be locked as this may lead to memory leaks. This function is provided for the rare case were, e.g. during the boot process, one application has to allocate a buffer that will be used by other applications later on.

CMEM_RCC_UNLOCK - unlocks a buffer.

CMEM_RCC_structure_DUMP - dumps the system parameters of all currently open buffers. You can get the same information with the command "more /proc/cmem_rcc".

CMEM_RCC_GETPARAMS

static int cmem_rcc_mmap(struct file *file, struct vm_area_struct *vma) -allows the mapping of device memory directly into a user process address space static ssize_t cmem_rcc_proc_write(struct file *file, const char *buffer, size_t count, loff_t *startOffset) static ssize_t cmem_rcc_proc_read(struct file *file, char *buf, size_t count, loff_t *startOffset)

static int membpa_init2(int priority, u_int btype) static void *membpa_alloc_pages(int count, int align, int priority, u_int btype) static void membpa_free_pages(void *base, u_int btype) static int gfpbpa_init(void)

FELIX PCIe device driver flx.c is a PCIe device driver for the FELIX PCIe interface. Supports the detection of FELIX cards on the basis of the PCIe bus enumeration by reading the Device ID and Vendor ID (DID/VID), specified by the Xilinx PCIe IP endpoint, handles interrupts and provides a number of ioctl functions.

The driver is registered with the kernel using the pci_driver struct. The name can be verified in /sys/bus/pci/drivers/ when the driver is in the kernel. The pcie_device id struct is used to define a list of PCI devices that a driver supports. The structure needs to be exported to user space to using the MODULE_DEVICE_TABLE however it has been disabled to prevent the driver from auto loading as pointed out by the comments in the code.

Similar to the cmem_rcc driver module_init and module_exit are used to inform the kernel that `felix_init()` should be called when the module is loaded, and `felix_exit` should be called when the module is removed.

The devices supported by the flx.c device driver are specified by the pci device id struct, for each device the vendor id and the device id is specified:

The probe function is called by the PCI core when it has a struct pci_dev that the driver wants to control. A Pointer to the pci_device_id struct that the PCI core used to make this decision is also passed to this function. The remove is a pointer to the PCI core pci_dev structure that is called when the device is removed from the system or the driver module is being unloaded. The flx_Probe and flx_Remove functions are are made known to the kernel as functions to be called at these times by means of the `pci_driver` struct, as shown in Listing 16.

```
/*****************************************************************/
static int cmem_rcc_mmap(struct file *file, struct vm_area_struct *vma)
/*****************************************************************/
{
  u_long offset, size;

  kdebug(("cmem_rcc(cmem_rcc_mmap): cmem_rcc_mmap called\n"));
#if LINUX_VERSION_CODE < KERNEL_VERSION(3,7,0)
  vma->vm_flags |= VM_RESERVED;
#else
  vma->vm_flags |= VM_DONTEXPAND;
  vma->vm_flags |= VM_DONTDUMP;
#endif
  vma->vm_flags |= VM_LOCKED;
  kdebug(("cmem_rcc(cmem_rcc_mmap): vma->vm_end    = 0x%016lx\n",
   ↪  (u_long)vma->vm_end));
  kdebug(("cmem_rcc(cmem_rcc_mmap): vma->vm_start  = 0x%016lx\n",
   ↪  (u_long)vma->vm_start));
  kdebug(("cmem_rcc(cmem_rcc_mmap): vma->vm_offset = 0x%016lx\n",
   ↪  (u_long)vma->vm_pgoff << PAGE_SHIFT));
  kdebug(("cmem_rcc(cmem_rcc_mmap): vma->vm_flags  = 0x%08x\n",
   ↪  (u_int)vma->vm_flags));

  size = vma->vm_end - vma->vm_start;
  offset = vma->vm_pgoff << PAGE_SHIFT;

#if LINUX_VERSION_CODE < KERNEL_VERSION(2,6,18)
  if (remap_page_range(vma, vma->vm_start, offset, size, vma->vm_page_prot))
#else
  if (remap_pfn_range(vma, vma->vm_start, vma->vm_pgoff, size, vma->vm_page_prot))
#endif
  {
    kerror(("cmem_rcc(cmem_rcc_mmap): function remap_page_range failed \n"));
    return(-CMEM_RCC_MMAP);
  }
  kdebug(("cmem_rcc(cmem_rcc_mmap): vma->vm_start(2) = 0x%016lx\n",
   ↪  (u_long)vma->vm_start));

  vma->vm_ops = &cmem_rcc_vm_ops;
  kdebug(("cmem_rcc(cmem_rcc_mmap): cmem_rcc_mmap done\n"));
  return(0);
}
```

Listing 14: cmem_rcc_mmap

```
#define PROC_MAX_CHARS        0x10000
#define PCI_VENDOR_ID_FLX_FW  0x10ee
#define PCI_DEVICE_ID_FLX_FW1 0x7038
#define PCI_DEVICE_ID_FLX_FW2 0x7039

static struct pci_device_id FLX_IDs[] =
{
  { PCI_DEVICE(PCI_VENDOR_ID_FLX_FW, PCI_DEVICE_ID_FLX_FW1) },
  { PCI_DEVICE(PCI_VENDOR_ID_FLX_FW, PCI_DEVICE_ID_FLX_FW2) },
  { 0, },
};
```

Listing 15: Vendor ID

```
// needed by pci_register_driver fcall
static struct pci_driver flx_PCI_driver =
{
  .name     = "flx",
  .id_table = FLX_IDs,
  .probe    = flx_Probe,
  .remove   = flx_Remove,
};
```

Listing 16: flx_PCI_driver

The flx_probe function does the following: The device driver is enabled by a call to the pci_enable_device function. Once the device driver enables the device it reads the configuration registers in the PCI controller using the pci_read_config_dword function. The pci_resource_start function returns the address of the region information of where the device has been mapped. The ioremap_nocache remaps a physical address range into the processors virtual address space without caching the data, making it available to the kernel. PCI offers two extensions of Message signal Interrupts (MSI and MSI-X) are an in-band method of signaling an interrupt to the host system. In-band meaning exchanging special messages that indicate interrupts (emulated pin assertion or de-assertion) through the main data path (software) as oppose to the traditional out-of-band dedicated interrupt pin. The primary difference between MSI (PCI 2.2) and MSI-X (PCI 3.0) is the former supports up to 32 interrupts and the latter permits devices to allocate up to 2048 interrupts. The pci_find_capability function verifies PCI message signal interrupt capability, and the remainder of the code uses the function called out above to obtain the message signal interrupt Base address register configurations and sets up the msix table structure.

The flx_init function does the following: It initializes interrupt request count mask and flags to 0, registers the pci driver using the pci_register_driver which points to the flx_pci_driver struct. Device numbers are special device files or nodes in the file-system tree; they are located in the /dev directory. Device numbers are divided into two groups major and minor. The major number identifies the driver associated with the device. The minor number is used by the kernel to determine exactly which device is being to referred to. alloc_chrdev_region dynamically obtains the device number from the kernel. first_dev is an output-only parameter that will, on successful completion, hold the first number in the allocated range. FIRSTMINOR should be the requested first minor number to use; it is 0 . The MAXCARDS parameter is the total number of contiguous device numbers requested. The devName of the device that should be associated with this number range; it will appear in /proc/devices and sysfs. The kernel internally represents the devices by using the cdev struct. If alloc_chrdev_region is successful cdev_alloc() is set to flx_cdev and flx_cdev is pointed to the fops struct. cdev_add is used to inform the kernel that cdev has been setup. flx_cdev is the cdev structure, first_dev is the first device number to which this device responds, and 1 is the number of device numbers that should be associated with the device. Device drivers typically export information via the software-created /proc filesystem Each file under /proc is tied to a kernel function that generates the files "contents" on the fly when the file is read. The proc_create function creates a read only entry into the proc file system and the proc_read_text function is used to allocate the maximum output

flx_init takes care of registering the driver by means of a call to pci register driver. flx_init also creates /proc/flx by calling create proc entry and associates the felix_read_procmem and felix_write_procmem functions with it. Upon reading from /proc/flx (e.g. more /proc/flx) felix_read_procmem is called, which provides information about the status of the hardware. Upon writing to /proc/flx (e.g. echo debug >/proc/flx), felix_write_procmem is called, which enables the user to enable or disable debugging or error logging. These options are listed in Table 8; these options (and msiblock) are also settable when the module is loaded.

Interaction with the driver from user space takes place using /dev/felix. Upon opening the device felix open is called, a close causes calling of felix Release. The ioctl and mmap system calls result in felix ioctl and felix mmap being called respectively, if these system call are supplied with the file descriptor obtained from opening /dev/felix. Reading from and writing to /dev/felix is not supported by the driver.

In flx_open memory is allocated for a struct of type card params t and the private data pointer of the struct associated with the file descriptor is set to point to the card params t struct:

The `private_data` associated with the file descriptor is used to pass the card information (BAR addresses and extents) between the user-mode software and the kernel driver. This allows the `arg` field of

Table 8: Options for the FELIX driver.

Parameter Name	User Interface	Description
debug	/proc/flx and insmod	Setting the parameter to 1 enables verbose debug output to /var/log/messages. This can be done as follows: insmod flx.ko debug=1 echo debug >/proc/flx Setting the parameter to 0 disables verbose debug output to /var/log/messages. This can be done as follows: insmod flx.ko debug=0 echo nodebug >/proc/flx Default: disabled
errorlog	/proc/flx and insmod	Setting the parameter to 1 enables verbose error output to /var/log/messages. This can be done as follows: insmod flx.ko errorlog=1 echo elog >/proc/flx Setting the parameter to 0 disables verbose error output to /var/log/messages. This can be done as follows: insmod flx.ko errorlog =0 echo noelog >/proc/flx Default: enabled
msiblock	insmod	This parameter controls the number of MSI-X interrupts that the FELIX card can handle. It can be selected in the range 1 - 8.

```
struct file_operations fops =
{
  .owner          = THIS_MODULE,
  .mmap           = flx_mmap,
  .unlocked_ioctl = flx_ioctl,
  .open           = flx_open,
  .read           = flx_read_procmem,
  .write          = flx_write_procmem,
  .release        = flx_Release,
};
```

Listing 17: file_operations structure

```
/*****************************************************/
int flx_mmap(struct file *file, struct vm_area_struct *vma)
/*****************************************************/
{
    u32 moff, msize;

    // it should be "shared" memory
    if ((vma->vm_flags & VM_WRITE) && !(vma->vm_flags & VM_SHARED))
    {
        kerror(("flx(flx_mmap): writeable mappings must be shared, rejecting\n"));
        return(-EINVAL);
    }

    msize = vma->vm_end - vma->vm_start;
    moff = vma->vm_pgoff;
    kdebug(("flx(flx_mmap): offset: 0x%x, size: 0x%x\n", moff, msize));
    moff = moff << PAGE_SHIFT;
    if (moff & ~PAGE_MASK)
    {
        kerror(("flx(flx_mmap): offset not aligned: %u\n", moff));
        return(-EINVAL);
    }

#if LINUX_VERSION_CODE < KERNEL_VERSION(3,7,0)
    vma->vm_flags |= VM_RESERVED;
#else
    vma->vm_flags |= VM_DONTEXPAND;
    vma->vm_flags |= VM_DONTDUMP;
#endif

    // we do not want to have this area swapped out, lock it
    vma->vm_flags |= VM_LOCKED;
    if (remap_pfn_range(vma, vma->vm_start, vma->vm_pgoff, msize,
     ↪  vma->vm_page_prot) != 0)
    {
        kerror(("flx(flx_mmap): remap page range failed\n"));
        return(-EAGAIN);
    }

    vma->vm_ops = &flx_vm_ops;
    return(0);
}
```

Listing 18: flx_mmap

```
/*********************************************/
int flx_open(struct inode *ino, struct file *file)
/*********************************************/
{
    card_params_t *pdata;

    %kdebug(("flx(flx_open): called\n"));
    pdata = (card_params_t *)kmalloc(sizeof(card_params_t), GFP_KERNEL);
    if (pdata == NULL)
    {
        %kerror(("flx(flx_open): error from kmalloc\n"))
        return(-ENOMEM);
    }

    pdata->slot = 0;
    file->private_data = (char *)pdata;
    return(0);
}
```

Listing 19: flx_open

the ioctl call to be used for information specific to the ioctl call (in contrast with `cmem_rcc`, where every ioctl call involving buffer information must pass the buffer information struct in the `arg`). The program that requested the card information is tracked using a static array `owner[]` in `flx`.

In felix release the memory allocated in felix open is freed.

Interrupt The FELIX cards use MSI-X interrupts in order to signal asynchronous events to the operating system. Per installed card the driver supports up to 8 interrupts, enumerated from 0 to 7. The number of interrupts is specified in the source of the driver by a constant (MAXMSI) and is determined by the firmware. There is no specific dependence of the driver on the nature of the interrupts.

ioctl The ioctl function is called with a file descriptor, the ioctl name and, if applicable, a user parameter. After opening the driver, the first ioctl function to be called is the SETCARD function, with a pointer to a struct of type card_params_t as user parameter. The card number has to be stored in this struct before calling the ioctl SETCARD function. The function copies the BAR0, BAR1 and BAR2 addresses as well as the sizes of the memory areas associated with these base addresses to the struct. The addresses are physical addresses. After mapping to virtual addresses the registers can be addressed directly from used space.

An interrupt causes a flag associated with it to be set and a process can wait for this to occur. By setting all flags with the ioctl CANCEL_IRQ_WAIT function any process waiting for an interrupt can be woken up. The user parameter of type int* sends data to the driver

CANCEL_IRQ_WAIT - wakes up processes currently waiting for the interrupt with the number specified in the integer pointed to from the FELIX card by setting the corresponding interrupt flag.

GETCARDS - returns the number of FELIX cards that have been found in the computer in the int vari-

able pointed to by the user parameter. The user parameter of type int* receives data from driver.

The contents of slot in the card params struct (type card params t) is set in the felix ioctl function, using the SETCARD command shown in Listing 20. SETCARD allows user processes to obtain physical addresses for the resources of the FELIX card.

The MSI-X interrupts are initialized in felix Probe. Currently there are 8 different interrupts, all with vector pointing to the same function with name irqHandler. The function counts interrupt requests for each of the 8 different interrupts. It also flags whether an an interrupt has been seen for each of the possible interrupts. The counts and flags can be obtained by reading from /proc/felix. A thread waiting for an interrupt that has itself registered on the wait queue (using ioctl) will be blocked until the interrupt occurs. This is the code of irqHandler:

The code below in felix ioctl WAIT_IRQ case that causes blocking until the interrupt occurs, which is signaled by the flag becoming equal to 1. wait_event_interruptible(waitQueue, irqFlag[card][interrupt] == 1); irqFlag[card][interrupt] = 0;

The flag set by irqHandler is reset here once execution continues after wait event.

4.6 Register Map

The control interface between the FELIX firmware and software is a set of configuration registers mapped to three BARs (Base Address Register). BAR0 contains the DMA descriptors and DMA control registers, BAR1 contains the PCIe interrupt table, and BAR2 contains all other FELIX configuration and status registers.

Both the firmware and software need to translate between register name and raw BAR address offset, and this mapping must remain consistent when the register list is updated. This is accomplished by having a single "register map" file and generating the necessary firmware and software files automatically based on the current register map.

The register map file is written in YAML (Yet Another Markup Language), and is kept in the firmware source tree (in `sources/templates/`). A Jinja template file exists for every firmware or software file that depends on the register map. The template files are kept in `sources/templates/`) for the firmware and `regmap/source/`) for the software. Each template file contains Jinja tags where register map information needs to be substituted; static blocks of code (not dependent on the register map) are written in their native language. A Python script (`wuppercodegen/wuppercodegen/cli.py`) reads the YAML register map file, calculates the BAR address offsets for all registers, and substitutes the register information into the template file.

4.7 Interfaces

Each FELIX will interface eight RUs over the radiation tolerant Fiber Optic Links implementing the standard GBT protocol. The data will be aggregated and placed into the hosts memory through the DMA and PCIE Gen 3 interface.

FELIX will receive GBT frames from the RU see figure 97

Following the Standard GBT commands defined below
Data Valid=1 80 bit GBT word marked as Data
Data Valid=0 80 bit GBT word marked as Control
Control Commands IDLE SOP EOP SWT
TTC

```
case SETCARD:
        if (copy_from_user( (void *) &temp, (void *)arg, sizeof(card_params_t)) !=
        ↪  0)
        {
          return(-EFAULT);
        }
        card = temp.slot;
        if (card >= MAXCARDS)
        {
              return(-EINVAL);
        }
        if (cards[card].pciDevice == NULL)
        {
              return(-EINVAL);
        }
        if (mutex_lock_interruptible(&ownerMutex))
        {
            return(-ERESTARTSYS);
        }
        owner[card] = file;
        mutex_unlock(&ownerMutex);

        cardParams = (card_params_t *)file->private_data;
        cardParams->slot = card;
        cardParams->baseAddressBAR0 = cards[card].baseAddressBAR0;
        cardParams->sizeBAR0 = cards[card].sizeBAR0;
        cardParams->baseAddressBAR1 = cards[card].baseAddressBAR1;
        cardParams->sizeBAR1 = cards[card].sizeBAR1;
        cardParams->baseAddressBAR2 = cards[card].baseAddressBAR2;
        cardParams->sizeBAR2 = cards[card].sizeBAR2;
        // OK, we have a valid slot, copy configuration back to user
        if (copy_to_user(((card_params_t *)arg), &cards[card],
        ↪  sizeof(car_params_t)) != 0)
        {
            return(-EFAULT);
        }
        break;
```

Listing 20: SETCARD option of felix ioctl

```
/*********************************************/
static irqreturn_t irqHandler(int irq, void *dev)
/*********************************************/
{
    struct irqInfo_struct *info;

    info = (struct irqInfo_struct*) dev;
    irqCount[info->card][info->interrupt] += 1;
    irqFlag[info->card][info->interrupt] = 1;
    wake_up_interruptible(&waitQueue);      //MJ: would it have any performance
    ↪  advantages if we used one wait queue per card?
    return(IRQ_HANDLED);
}
```

Listing 21: irqreturn

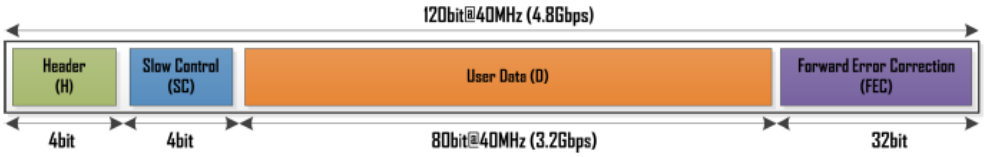


Figure 97: Felix GBT Frame

4.7.1 Fiber Mapping

The mapping from GBT link number and FPGA pin is set in a constraint file (felix_gbt_minipod_BNL711_transceiver_16 or felix_gbt_minipod_BNL711_transceiver_24ch.xdc, depending on the number of links). The mapping from FPGA pin to position on the MTP multifiber connectors is determined by the PCB connectivity between the FPGA and the MiniPOD optical modules, and fiber patch cables (provided by BNL with FELIX) from the MiniPODs to the MTP connectors. The resulting mapping from GBT link number to fiber position is shown in Figure 98.

Breakout box back (MTP connector)

MTP connector key

FELIX RX	12	13	14	15	16	17	18	19	20	21	22	23
FELIX TX	12	13	14	15	16	17	18	19	20	21	22	23
FELIX RX	0	1	2	3	4	5	6	7	8	9	10	11
FELIX TX	0	1	2	3	4	5	6	7	8	9	10	11

Breakout box front (LC connectors)

FELIX TX							FELIX RX					
22	20	18	16	14	12	22	20	18	16	14	12
23	21	19	17	15	13	23	21	19	17	15	13
10	8	6	4	2	0	10	8	6	4	2	0
11	9	7	5	3	1	11	9	7	5	3	1

Figure 98: Mapping of the GBT link numbers at the MTP-LC breakout box. This mapping assumes FELIX v1.5 is programmed for 24 GBT links, and a 48-fiber cable is plugged into the top (further from motherboard) MTP connector on FELIX. Both maps are drawn as if looking into the connectors on the breakout box.

4.8 Hardware

Xilinx Kintex Ultrascale XCKU115-FLVF1924-2-E
48 bi directional Fiber Optic Links
16 lane PCIE Gen3
Timing Mezzanine site

4.8.1 Clock Distribution and Configuration

FELIX has two main clocks: the fabric clock, which drives the FPGA logic, and the GTH transceiver reference clock, which sets the GBT line clock. The fabric clock is generated by the clock_and_reset firmware module, and is derived from a fixed 200 MHz oscillator. The transceiver reference clock is input directly to dedicated clock pins on the FPGA, and is generated by a jitter cleaner ASIC. The jitter cleaner connections are listed in Table 9.

The jitter cleaner normally used is the SiLabs Si5345, which can be configured to use any of 3 inputs or generate a free-running clock. The Si5345 is controlled by the FPGA through the I2C bus. The FELIX software provides (in the FlxCard API) functions to control the I2C bus through register reads and writes, and the utility `flx-i2c` allows I2C accesses from the command line.

The MVTX firmware distribution provides shell scripts that use `flx-i2c` to configure the Si5345. Si5345_40p8mhz_BNL-711v1p5.sh configures for a free-running clock, while Si5345_40p8mhz_BNL-711v1p5_EXTERNAL_IN02.sh configures for an input on a pair of FELIX test points. Generating a shell

Table 9: Inputs and outputs for the Si5345 and LMK03200 jitter cleaners. The MVTX firmware uses the Si5345 OUT0 and OUT2 for the GBT transceivers.

Si5345	
IN0	FPGA I/O bank 66
IN1	200 MHz SI530 oscillator
IN2	test points
IN3	feedback (tied to OUT9)
OUT0	FPGA GTH bank 127 REFCLK1 (for GBT)
OUT1	FPGA GTH bank 127 REFCLK0 (for GBT)
OUT2	FPGA GTH bank 132 REFCLK1 (for GBT)
OUT3	FPGA GTH bank 132 REFCLK0 (for GBT)
OUT4	FPGA GTH bank 228 REFCLK0 (for GBT)
OUT5	FPGA GTH bank 231 REFCLK1 (for GBT)
OUT6	FPGA GTH bank 231 REFCLK0 (for GBT)
OUT7	FPGA I/O bank 45 (for DDR module A)
OUT8	FPGA I/O bank 52 (for DDR module B)
OUT9	feedback (tied to IN3)
RSVD2/3	FPGA GTH bank 229 REFCLK1
RSVD4/5	FPGA GTH bank 233 REFCLK0
LMK03200	
OSCIN	FPGA (bank 66)
CLKOUT0	FPGA GTH bank 126 REFCLK1 (for GBT)
CLKOUT1	FPGA GTH bank 128 REFCLK0 (for GBT)
CLKOUT2	FPGA GTH bank 131 REFCLK1 (for GBT)
CLKOUT3	FPGA GTH bank 133 REFCLK0 (for GBT)
CLKOUT4	FPGA GTH bank 228 REFCLK1 (for GBT)
CLKOUT5	FPGA GTH bank 230 REFCLK1 (for GBT)
CLKOUT6	FPGA GTH bank 232 REFCLK1 (for GBT)
CLKOUT7	FPGA GTH bank 232 REFCLK0 (for GBT)

script for a new configuration requires two steps: first the SiLabs ClockBuilderPro software is used to generate a C header file containing register values, then the script make_si5345script.awk parses the header file and generates a shell script.

4.9 Test and Validation

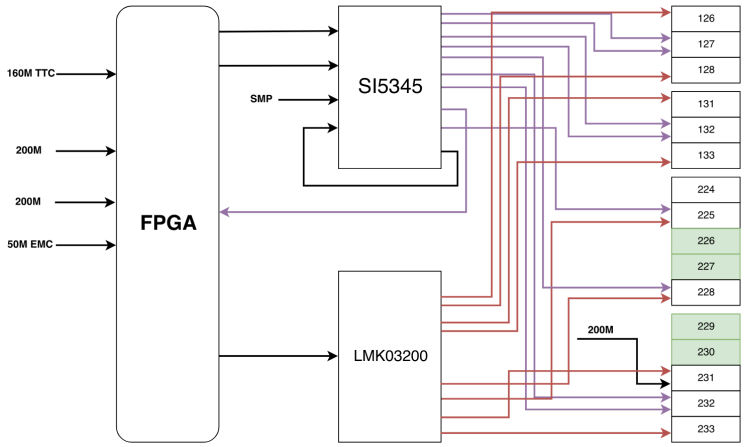


Figure 99: Mapping from jitter cleaners to GTH transceiver banks. The green banks are used for PCIe endpoints and are clocked using an Si53154 jitter clock buffer, not shown.

5 Data Acquisition System

The data acquisition (DAQ) for MVTX is being developed using the RCDAQ software package.

5.1 Description

5.1.1 RCDAQ

RCDAQ is the DAQ package used for sPHENIX R&D. The source repository is https://github.com/sPHENIX-Collaboration/rcdaq. RCDAQ is primarily developed and maintained by Martin Purschke, and the documentation can be found at www.phenix.bnl.gov/~purschke/rcdaq/rcdaq_doc.pdf.

RCDAQ shares the same event structure with standard PHENIX and sPHENIX DAQ, which is called PRDF (PHENIX Raw Data Format)[2].

5.1.2 FELIX Plugin

The MVTX-specific DAQ functionality is contained in a plugin, which is loaded by RCDAQ at runtime. The plugin configures FELIX at start-of-run and end-of-run, copies data from the FELIX DMA buffer to RCDAQ, and (optionally) triggers RCDAQ based on interrupts generated by FELIX.

The source repository is https://gitlab.cern.ch/sphenix-mvtx/felix_rcdaq/. The FELIX plugin is developed and maintained by the MVTX team, and this document is the primary documentation. Additional documentation is available at https://gitlab.cern.ch/sphenix-mvtx/felix_rcdaq/wikis/home.

5.1.3 Data Format and Decoder

The MVTX data is decoded by an MVTX-specific decoder that reads in an MVTX packet, interprets the hit information, and exposes the hit information through standard PRDF software interfaces.

The source repository for the decoder is https://github.com/sPHENIX-Collaboration/online_distribution/ (for MVTX, files `newbasic/oncsSub_idmvtxv0.h`, `newbasic/oncsSub_idmvtxv0.cc`). The decoder framework is maintained by Martin Purschke, and the MVTX decoder is developed and maintained by the MVTX team. This document is the primary documentation for the MVTX decoder.

5.1.4 Online Monitoring

The online monitoring framework for RCDAQ is called pmonitor. An application compiled against pmonitor (a pmonitor "project") can connect to a running RCDAQ instance or read in a PRDF file, and display real-time information on decoded events.

The source repository for the pmonitor framework is https://github.com/sPHENIX-Collaboration/online_distribution/ (subdirectory pmonitor). The pmonitor framework is maintained by Martin Purschke.

The pmonitor project for MVTX monitoring is in the same repository as the FELIX plugin: https://gitlab.cern.ch/sphenix-mvtx/felix_rcdaq/, subdirectory online_monitoring.

5.1.5 Offline Analysis

The offline analysis framework for sPHENIX is Fun4All. MVTX offline calibration and tracking are implemented in Fun4All.

https://github.com/sPHENIX-Collaboration/analysis (MvtxTelescope/AnaTestbeamV1)

https://github.com/damcglinchey/coresoftware/tree/telescope (offline/packages/MvtxPrototype1, offline/packages/mvtx, offline/packages/TrackBase)

[2] The format used in RCDAQ has slight differences from the PHENIX PRDF. From Martin:
"RCDAQ has, currently as a compile-time option, the ability to write what I like to think of as the 'PHENIX legacy' format that you enable with the '#define WRITEPRDF' in rcdaq.cc. You should not enable that. This supports PHENIX readout gear that encodes 'legacy' headers in firmware, most notably the HBD electronics that we used during the past testbeams."

5.2 Functionality

5.2.1 RCDAQ and FELIX Plugin

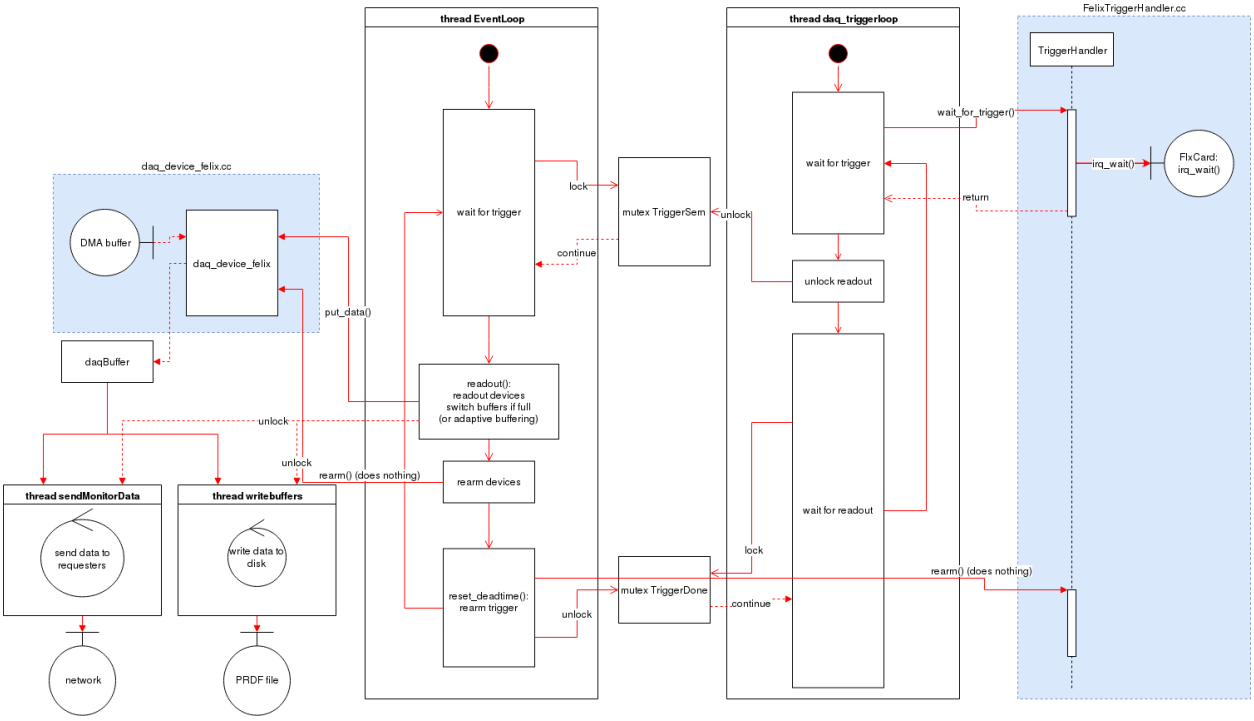


Figure 100: Block diagram of RCDAQ and the FELIX plugin, showing the four RCDAQ threads dealing with data and trigger (threads for web server and monitoring request listener are not shown). The two components of the FELIX plugin are highlighted in blue. The "RCDAQ buffer" in the figure is really a ping-pong buffer: the readout thread writes data to a "fill buffer" while the output threads transfer data from a "transport buffer." Periodically (either when the fill buffer is full, or when an "adaptive buffering" time threshold is reached), the function switch_buffer() switches the fill buffer and transport buffer.

RCDAQ provides basic functionalities like control panel, run start/stop, and I/O buffers. RCDAQ can be controlled and monitored through the command-line, a local GUI, or a web GUI. Since the RCDAQ server is controlled using the RPC (Remote Procedure Call) network protocol, the command-line and GUI clients can run on separate computers from the server.

Separate threads handle readout, trigger, file output, and output to monitoring. Figure 100 illustrates the communication between threads and with the FELIX plugin. RCDAQ stores run configuration by reading in files and storing them in PRDF events.

All readout devices, including FELIX for MVTX, are implemented as plugins that are loaded dynamically at runtime. The FELIX plugin extends three abstract RCDAQ classes, `RCDAQPlugin`, `daq_device`, and `Triggerhandler`, which defines a series of virtual functions regulating how a device is initialized, registered and read out.

Figure 101 shows the connection between the FELIX plugin and the low-level drivers. The low-level device software is explained in Section 4.5.

felix_plugin This extends the RCDAQ class `RCDAQPlugin`, which is the interface for plugin instantiation. `create_device()` parses the plugin arguments given to RCDAQ (through the `rcdaq_client create_device`

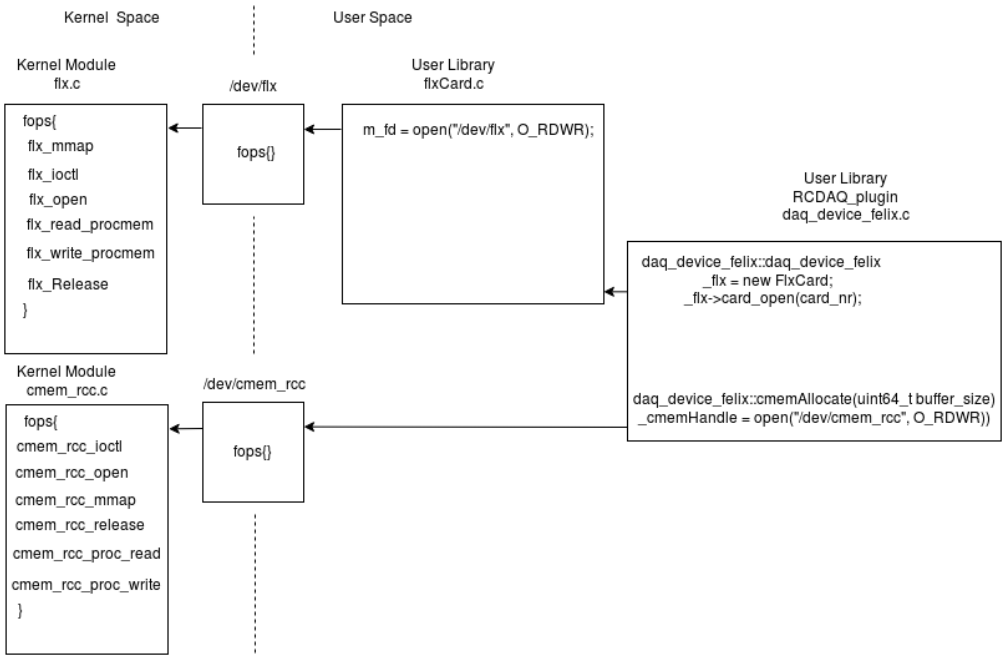


Figure 101: FELIX software block diagram. The interface between user-mode software (RCDAQ and the FELIX plugin) and kernel-mode drivers (flx.c and cmem_rcc.c) is the fops methods defined in the drivers.

command). The plugin parameters are described in Table 10. `felix_plugin` does not directly interface to or control FELIX; instead it instantiates `daq_device_felix` and `felixTriggerHandler`, which do.

daq_device_felix This extends the RCDAQ class `daq_device`, which is the interface for readout. This interfaces with MVTX through the FELIX DMA buffer (allocated by cmem_rcc) and the FELIX registers (accessed through FlxCard).

`put_data()` copies data from the FELIX DMA buffer into a PRDF packet. The other RCDAQ interface methods implemented in `daq_device_felix` are `init()` and `endrun()` (called on `rcdaq_client daq_begin` and `rcdaq_client daq_end` respectively). These methods start and stop the DMA engine, and switch the "upstream fanout" and "downstream fanout" muxes in the FELIX data path. Much of this functionality is adapted from the `fdaq` utility in the standard FELIX software distribution.

felixTriggerHandler This extends the RCDAQ class `TriggerHandler`, which is the interface for triggering (and is only implemented if the device is capable of triggering RCDAQ). This interfaces with the FELIX interrupts through FlxCard.

`wait_for_trigger()` waits until a trigger is received from FELIX. This is a wrapper around `FlxCard::irq_wait()`.

RCDAQ interface to device drivers This section will describe the rcdaq.cc interface to the daq_device_felix.cc plugin and the flow of how it controls the rest of the user space libraries and device driver software described above. The purpose is to inform the reader of the general signal/data flow prior to describing low level details. Therefore, the description is limited the function calls between the software packages and not a full description of each function. The plugin receives the following commands from the command line, constructor, destructor, daq_begin, trigger, and daq_end, daq_.

There are five main events in the plugin control sequence. Each is described in detail below.

- When the FELIX device is created by `rcdaq_client create_device daq_device_felix`, the plugin constructor is called.
- When the run is started by `rcdaq_client daq_begin`, the plugin initializes the DMA and starts running the readout chain.
- During the run, the `daq_triggerloop` thread of RCDAQ waits for a trigger from the plugin. When a trigger is received, the `EventLoop` thread of RCDAQ requests data from the plugin.
- When the run is stopped by `rcdaq_client daq_end`, the plugin stops the DMA and the readout chain.
- When RCDAQ is shutdown by `rcdaq_client daq_shutdown`, the plugin destructor is called.

create_device An RCDAQ plugin is compiled as a shared object library (.so file), and is loaded using the command `rcdaq_client load <plugin file>`. In `rcdaq_client.cc`, the `main()` function calls `command_execute()`. Via RPC, `command_execute()` tells `rcdaq_server.cc` to run the function `r_action_1_svc()` with action `DAQ_LOAD`; this in turn calls the function `daq_load_plugin()`, which loads the plugin via the C library function `dlopen()`. When the plugin is loaded, RCDAQ adds it to an internal plugin list.

A readout device is initialized using the command `rcdaq_client create_device <device name>`. In `rcdaq_client.cc`, the `main()` function calls `command_execute()`, which calls `handle_device()`. Via RPC, `handle_device()` tells `rcdaq_server.cc` to run the function `r_create_device_1_svc()`, which loops through the plugin list and attempts to call the `create_device()` function for each plugin.

For `felix_plugin`, the arguments to `create_device` are defined as shown in Table 10. The typical `create_device` command for MVTX is `rcdaq_client create_device device_felix 1 2000 0 1 0 4 5`.

Table 10: Arguments for the FELIX plugin.

index	name	typical value	description
1	event type	1	event type for which the readout should run
2	subevent ID	2000	packet ID for PRDF
3	card number	0	card number (as interpreted by `FlxCard::card_open()`)
4	buffer size	1	DMA buffer size (units of MB)
5	DMA index	0	not used, may be necessary for multiple DMAs
6	interrupt ID	4	PCIe interrupt ID used for trigger
7	flags	5	bitmask (see below)
	flags(0)		enable trigger
	flags(1)		fanoutEna (must be set to 0)
	flags(2)		externalEmu (must be set to 1)

`create_device()` also calls the constructor for `daq_device_felix`; if FELIX trigger is enabled (as is usual), this in turn calls the constructor for `felixTriggerHandler` and registers `felixTriggerHandler` as the trigger handler for RCDAQ.

The `daq_device_felix` constructor calls the function `cmemAllocate()` and the FlxCard method `card_open()`.

`cmemAllocate()` in `daq_device_felix.cc` uses the `CMEM_RCC_GET` ioctl call of the cmem_rcc driver to allocate a memory buffer of the size requested by the plugin parameter. This buffer is then mapped

to virtual memory using the system call `mmap()`, which acts through the mmap method of the `cmem_rcc` file_operations interface. The obtained virtual address is then communicated back to the driver using the `CMEM_RCC_SETUADDR` ioctl call of the cmem_rcc driver.

`card_open()` in `FlxCard.cc` uses the `SETCARD` ioctl call of the flx driver to obtain the BAR addresses for the PCIe endpoint on FELIX, and maps the BARs to memory. It also uses the `GET_TLP` ioctl call of the flx driver to get the TLP parameter.

daq_begin A RCDAQ run is started by the command `rcdaq_client daq_begin`. The `main()` function in `rcdaq_client.cc` calls `command_execute()`, which uses RPC to call `r_action_1_svc()` in `rcdaq_server.cc` with an argument of `DAQ_BEGIN`. This calls `daq_begin()` in `rcdaq.cc`, which calls the functions `device_init()`, which calls the `init()` function for every device in the device list, and `enable_trigger()`, which calls the `enable()` method for the defined trigger handler.

In `daq_device_felix`, `init()` calls `dmaStart()` and (in FlxCard) `irq_enable()`. `dmaStart()` calls `dma_to_host()` in `FlxCard.cc`, which maps BAR 0 (the DMA descriptor), then initializes descriptor 0, and enables descriptor 0. `irq_enable()` in `FlxCard.cc` calls an ioctl method in flx.c which will select `UNMASK_IRQ` of the case statement.

`felixTriggerHandler::enable()` does nothing, since the interrupt is already enabled by `daq_device_felix::init()`.

During the run While FELIX is receiving triggers, the `daq_triggerloop` thread of `rcdaq.cc` calls the `wait_for_trigger()` function in felixTriggerHandler, which calls `irq_wait()` in FlxCard.c. `irq_wait()` calls an ioctl method in flx.c which will select WAIT_IRQ of the case statement.

When `wait_for_trigger()` completes, the `EventLoop` thread of `rcdaq.cc` interprets this as a trigger and calls `put_data()` for all defined devices. For `daq_device_felix`, `put_data()` copies data from the FELIX DMA buffer into a PRDF packet. Figure 102 shows the source and destination buffers. `put_data()` transfers the data as follows:

- Get the DMA write pointer (`dma_status_t.current_address`) and compare to the read pointer (`_prevAddr`). If equal, do nothing, since there is no data; otherwise, proceed.
- Read the count from the DMA buffer, at the address `_prevAddr`. This is in units of 256-bit FELIX words.
- Copy the count and data words out of the DMA buffer and into the RCDAQ buffer: the start address is `_prevAddr`, the number of bytes to copy is $(\text{count}+1)\times 32$ (each FELIX word is 32 bytes). If the data
- Increment the read pointer (`_prevAddr`) by $8\times\text{ceil}((\text{count}+1)/8)\times 32$: the DMA transfer size is 8 FELIX words, so we move to the next multiple of 8 in order to skip the padding words added by the FELIX firmware. Now `_prevAddr` will point to the count from the next FELIX event.
- Add a 0xF000F000 "end of data" word to the packet, and add the padding required by RCDAQ.

daq_end The daq_end method within the rcdaq.cc call disable_trigger(), and device_endrun(). device_endrun() will call the int daq_device_felix::endrun() in the plugin daq_device_felix.cc. The daq_device_felix::endrun(), will call _flx->irq_disable(_interrupt_id), _flx->irq_cancel(_interrupt_id), where interrupt_id is 4 as defined by the plugin parameters (see Table 10) in the start_felix_rcdaq_felixTrigger.sh. The two irq calls described above will call void FlxCard::irq_disable(u_int interrupt) and void FlxCard::irq_cancel(u_int interrupt) in FlxCard.cpp The FLXCard methods previously mentioned will call make an ioctl method in flx.c which will select CANCEL_IRQ_WAIT and MASK_IRQ of the case statement.

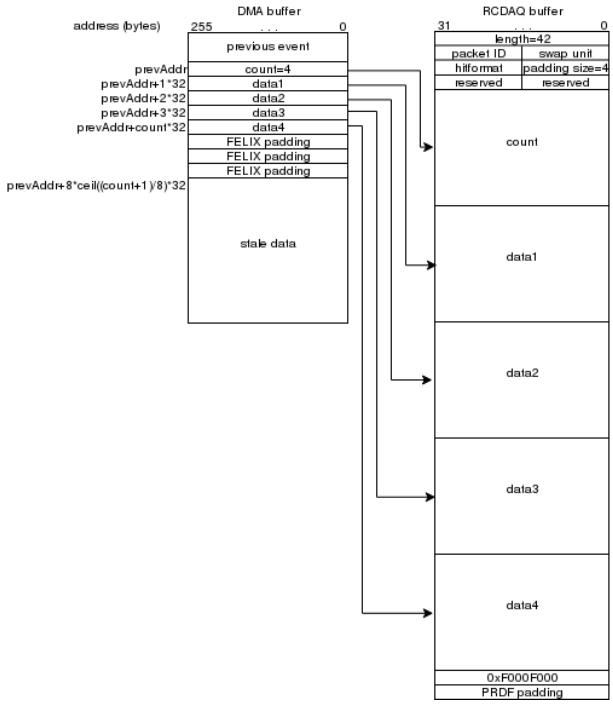


Figure 102: Example of the data transfer between the DMA buffer and RCDAQ buffer. The FELIX buffer is drawn with a word width of 256 bits (the width of the Central Router FIFO), and the RCDAQ buffer is drawn with a width of 32 bits. The transfer strips off the padding in the FELIX buffer (necessary to align the data to the DMA transfer size) and adds padding to the RCDAQ packet (necessary to align the packet to the 64-bit RCDAQ alignment unit).

daq_shutdown When RCDAQ is shut down, the destructors for all devices are called. The destructor for the FELIX plugin stops any current run using the `endrun()` function (same as in `daq_end`), then closes the FlxCard object using `FlxCard::card_close()` and releases the CMEM buffer using the system call `munmap()` and the CMEM ioctl call `CMEM_RCC_FREE`.

5.2.2 Data Format

PRDF organizes events as a collection of packets coming from different readout devices. Fig. 103 shows the content of a typical PHENIX raw event, which consists of 827 packets in total with different packet ID (column 2), various length (column 3) and type (column 6 and 7). Within one event the order of packets from different readout devices is irrelevant, since each readout device has a unique packet ID.

A typical PRDF packet structure is shown in Fig. 104. When a chunk of data is received from a certain readout device, RCDAQ will add a 4-word header (or envelope information) to form a packet. The length of a packet is required to be a multiple of 64 bits[3]; this is done by extending the length of a packet with "padding" words that are not decoded.

The packet header includes:

- length of the data block in this packet including padding (in units of 32-bit words)

[3]The padding alignment is not enforced in the PRDF libraries, and some presentations refer to 128-bit alignment, but the sPHENIX plugins align to 64 bits. From Martin:
"indeed, I always go on about 128bit alignment, which we *can* do. (And probably will in the future). All headers obey that. I do 64bit at this point, which is adequate as long as no 128bit CPU comes along."

```
$ dlist  /a/eventdata/EVENTDATA_P00-0000459344-0000.PRDFF
Packet  14001    52  0 (Unformatted)    714 (IDGL1)
Packet  14007    10  0 (Unformatted)    716 (IDNTCZDC_LL1)
Packet  14002     9  0 (Unformatted)    701 (IDBBC_LL1)
Packet  14009    14  0 (Unformatted)    717 (IDGL1_EVCLOCK)
Packet  14011    13  0 (Unformatted)    914 (IDGL1PSUM)
Packet   8180    21  0 (Unformatted)   1508 (IDEMC_FPGA3WORDS0SUP)
Packet   8165    42  0 (Unformatted)   1508 (IDEMC_FPGA3WORDS0SUP)
Packet   8166    48  0 (Unformatted)   1508 (IDEMC_FPGA3WORDS0SUP)
. . .
Packet  25121    83  0 (Unformatted)    425 (IDFVTX_DCM0)
Packet  25122   198  0 (Unformatted)    425 (IDFVTX_DCM0)
Packet  25123    99  0 (Unformatted)    425 (IDFVTX_DCM0)
Packet  25124    46  0 (Unformatted)    425 (IDFVTX_DCM0)
Packet  21351   356  0 (Unformatted)   1121 (IDMPCEXTEST_FPGA0SUP)
Packet  21352   319  0 (Unformatted)   1121 (IDMPCEXTEST_FPGA0SUP)
Packet  21353   238  0 (Unformatted)   1121 (IDMPCEXTEST_FPGA0SUP)
Packet  21354   323  0 (Unformatted)   1121 (IDMPCEXTEST_FPGA0SUP)
```

Figure 103: PRDF packet listing.

- packet ID to uniquely define the origin of this packet
- swap unit to identify the word size (1, 2, or 4 bytes) to use for byte-swapping when changing endianness
- hit format identifier for decoder
- length of padding (in units of bytes)
- two 16-bit words reserved for a future use

Word	16 bit	16 bit
0	Length	
1	Packet id	Swap unit
2	Hitformat	Padding size
3	Reserved	reserved
4 +	DATA	
n+4	padding	

Figure 104: PRDF packet structure.

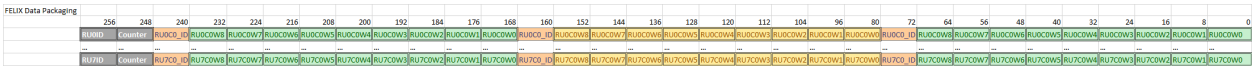
Given a data stream from RCDAQ or a PRDF file, the PRDF libraries are responsible for breaking the data stream up into events and the events into packets. The libraries provide a generic interface (`packet.h`) by which analysis code can retrieve information from a packet. Each data format has a specific implementation of this interface, which is the decoder. The MVTX decoder is called `oncsSub_idmvtxv0`. When

analysis code requests a packet from an event, the packet's hitformat determines which decoder interprets the packet.

The MVTX readout chain consists of two steps of data aggregation, and this is reflected in the data format as shown in Figure 105. The RU aggregates data from multiple ALPIDE sensors, so each 80-bit RU data word contains 9 data bytes from a single ALPIDE, tagged with the ALPIDE ID. FELIX aggregates data from multiple RUs, so each 256-bit FELIX data word contains 3 RU data words, tagged with the RU ID.

The decoder reverses this process to reconstruct the data stream from each ALPIDE:

- Iterate through the FELIX data words. For each:
  - Identify the RU ID.
  - For each of the three RU data words, identify the ALPIDE ID, and add the ALPIDE data bytes to the appropriate ALPIDE data stream.
- For each ALPIDE data stream, iterate through the bytes:
  - If a multi-byte data code (CHIP_HEADER, CHIP_EMPTY, DATA_SHORT, DATA_LONG) was in progress, interpret the new byte together with the previous byte(s). If this completes a DATA_SHORT or DATA_LONG code, add the hit data to the decoder hit map.
  - Otherwise, identify and interpret the new byte as a data code.



RU Data Packaging														
	Data Valid	72	64	56	48	40	32	24	16	8	0			
Control	0										SOP	SOP	Start of packet	0001
Data	1	C0_ID	C0W8	C0W7	C0W6	C0W5	C0W4	C0W3	C0W2	C0W1	C0W0			
	...	...	...	...	...	...	...	...	...	...	...			
Data	1	C8_ID	C8W8	C8W7	C8W6	C8W5	C4W4	C8W3	C8W2	C8W1	C8W0			
	0										EOP	EOP	End of packet	0010

	ALPIDE Data Packaging			
IDLE(8bits)	Chip Header(16 bits)	Region Header(8bits)	DATA SHORT(16 bits)	CHIP TRAILER(8bits)
1111_1111	1010<chip id[3:0]><BUNCH COUNTER FOR FRAME[10:3] >	110<region id[4:0]>	01<encoder id[3:0]><addr[9:0]>	1011<readout flags[3:0]>

Figure 105: From top to bottom: the format in which FELIX data words are parsed to extract RU data words, the format in which RU words are parsed to extract ALPIDE data bytes, and the ALPIDE data codes.

The decoded data is accessed through so-called "iValue" methods that are part of the PRDF packet interface. For example, `int iValue(const int chip, const int region, const int row)` specifies an ALPIDE sensor and a readout region and row. Since each readout region has 32 columns, this corresponds to a group of 32 pixels, and the return value is a 32-bit bitmap where the high bits correspond to pixels with hits in this event.

The decoder also provides a text dump interface for debugging purposes. This interface is conveniently accessed through the command-line utility `ddump`, which is part of the PRDF libraries. The MVTX decoder provides two dump formats: a decoded format, which prints the bitmaps accessed through the iValue interface, and a "raw" format, which prints the data directly. Both ddump formats are shown in Figure 106.

Table 11: An example of an MVTX analysis pipeline.

name	description
Fun4AllPrdfInputManager	Input manager: read events from a PRDF file.
MvtxRunInfoUnpackPRDF	Analysis module: unpack beam information and ALPIDE parameters from the PRDF begin of run event.
MvtxUnpackPRDF	Analysis module: unpack hit information from PRDF events and create hit objects.
MvtxApplyHotDead	Analysis module: reject hits in hot pixels.
MvtxClusterizer	Analysis module: make clusters.
MvtxAlign	Analysis module: apply alignment corrections.
AnaMvtxPrototype1	Analysis module: make tracks and fill histograms.
Fun4AllDstOutputManager	Output manager: write event information to a DST (Data Summary Tape) file.

5.2.3 Online Monitoring (pmonitor)

MVTX online monitoring is implemented using the pmonitor framework. A pmonitor project is compiled against pmonitor, PRDF, and ROOT libraries, and is executed by ROOT. The pmonitor framework receives PRDF data, decodes the events using the PRDF libraries and packet decoders as described in 5.2.2, and then runs user-supplied code to analyze the events and fill ROOT histograms. pmonitor is multithreaded, and can use separate threads for the event transfer, analysis, and control.

pmonitor projects run separately from RCDAQ, and communicate with RCDAQ over the network. Multiple pmonitor projects (for different monitoring displays) can connect to the same RCDAQ instance. A pmonitor project can also read data from a PRDF file.

The MVTX pmonitor project was developed for the 2018 test beam. Figure 107 shows the monitoring display from the test beam. The MVTX-specific code is contained in a `mvtx.cc` file, and interfaces to the pmonitor framework through two functions: `pinit()` and `process_event(Event * e)`. `pinit()` initializes the histograms when the project is loaded; `process_event()` is called by the pmonitor framework for every event that is received.

5.2.4 Offline Software

sPHENIX uses the Fun4All analysis framework. As shown in Figure 108, Fun4All is based on "analysis modules" that can be registered with a Fun4AllServer to create a analysis pipeline. Table 11 shows the list of modules that make up a typical MVTX analysis pipeline. Since the modules are loaded and configured at runtime based on a macro, the pipeline can be completely reconfigured without recompiling anything.

5.3 Interfaces

The overall architecture of the sPHENIX DAQ system is shown in Fig. 109.

The plug-in developed for test beam RCDAQ should be largely the same as the one used on DAM [4].

[4] From Martin:
"Think of RCDAQ as a place to plug your detector's readout in. It is clear that the full sPHENIX DAQ will be bigger than RCDAQ, but the 'plug' (APIs, configuration etc) will be largely the same — once you are setup in RCDAQ, the integration work still required in 2021 will be minimal."

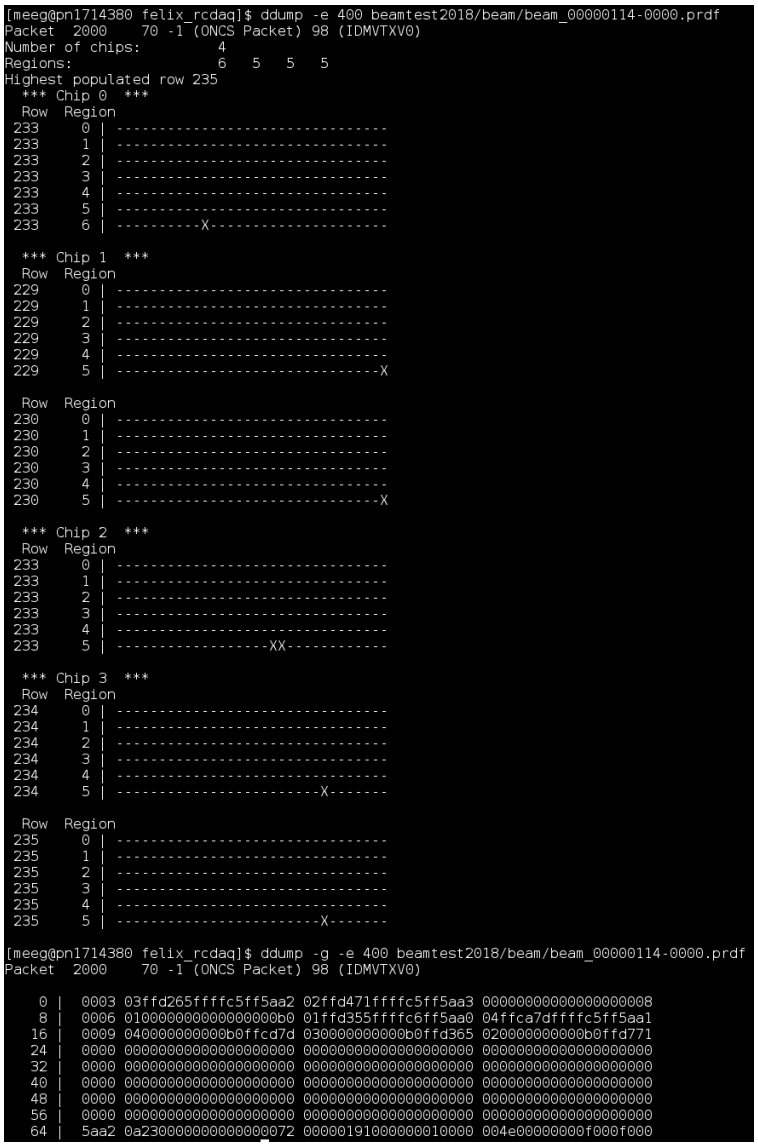


Figure 106: The decoded and raw ddump outputs.

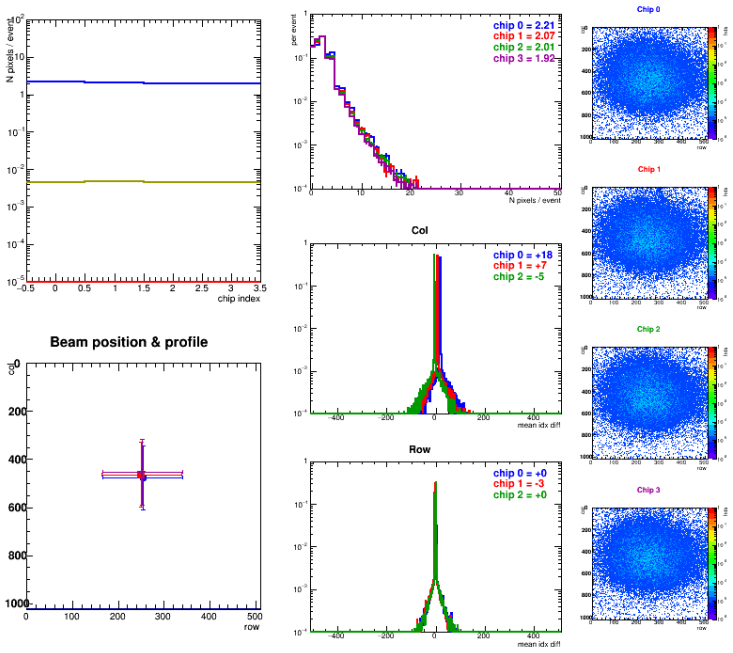


Figure 107: Online monitoring display for MVTX. The data displayed is from the 2018 test beam at Fermilab, with a telescope of four single ALPIDE sensors. The top left histograms show the mean number of hits per sensor per event; the top center histograms show the distributions of hit counts per event. The 2-D histograms on the right show the hit distributions for each sensor; the bottom left plot shows the results of Gaussian fits to those distributions. The bottom two plots in the center column show the event-by-event hit correlations (the differences between hit positions in different sensors), which is a measure of the relative alignment of the sensors.

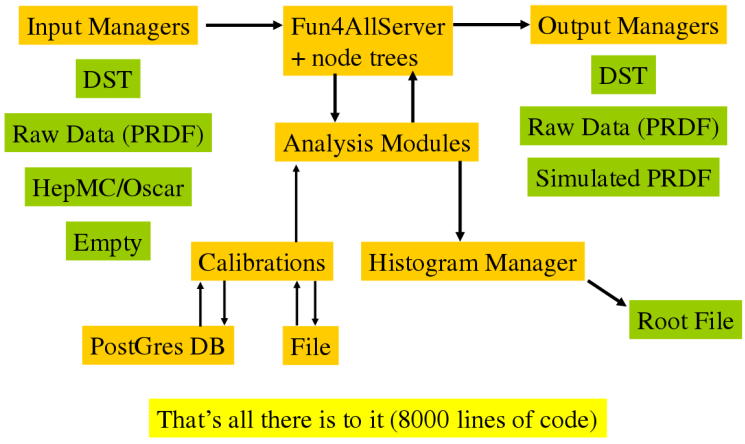


Figure 108: Structure of the Fun4All framework. All configuration is done through the Fun4AllServer by registering different input managers, analysis modules, and output managers. Most functionality is contained in analysis modules, which exchange event information with the server via "node trees." Analysis modules can output histograms directly through a histogram manager, or event data from the node trees can be dumped by an output manager.

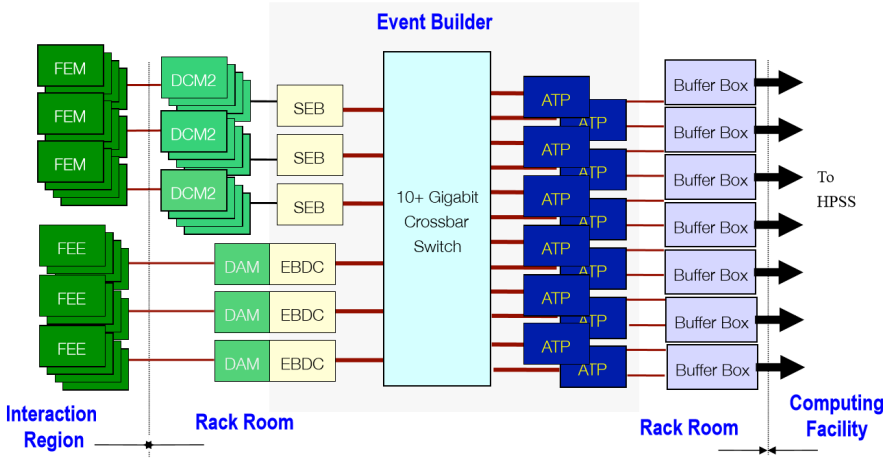


Figure 109: The bottom half with FEE (Front End Electronics) and DAM (Data Aggregation Module) represents TPC and MVTX readout. For MVTX, FEE refers to the RU and DAM refers to FELIX. Each EBDC (Event Buffering and Data Compressor) is a server hosting a single FELIX board.

6 Timing and Triggering

6.1 Timing specification

sPHENIX follows more or less the similar approach we have used in the previous PHENIX experiment:

- Uniform Timing, Trigger & Controls
- Uniform raw data format
- Limited information available at, https://wiki.bnl.gov/sPHENIX/index.php/Electronics
- Timing System:
  - Distribute RHIC Clock to front-end electronics(FEMS)
  - Send accepted Lvl-1 trigger to the FEMs
  - Manage multi-event buffers (5-event deep buffer)
- One level trigger system - Level-1
  - trigger latency 4μS (37 RHIC Clocks in PHENIX)
  - Handle subsystem busy
  - Control minimal trigger interval (âĂIJdead for 4âĂİ in PHENIX)

RHIC clock is 9.4MHz, much slower than the LHC clock of 40MHz. All sPHNIX and MVTX trigger and timing is synchronized to the 9.4MHz RHIC clock. We plan to operate ALIPDE chips at the same 40MHz LHC clock. This new 40MHz clock will be generated and phase locked to the RHIC 9.4MHz clock, mostly likely at FELIX end. A dedicated daughter card will be developed by the BNL ATLAS group for the next version FELIX board v.20, to allow 9.4MHz RHIC clock as input.

6.2 Timing interfaces

6.2.1 Trigger specification

6.3 Trigger interfaces

7 Control and Monitoring

7.1 Description

high voltage & bias

7.2 Functionality

7.2.1 Specifications

7.3 Interfaces

7.4 Hardware

7.5 Test and Validation

8 Test and Validation

8.1 Full Chain Test

8.1.1 FELIX and RCDAQ

program FELIX Open Vivado:

```
source /opt/Xilinx/Vivado/2015.4/settings64.sh
vivado
```

- open hardware manager
- connect: always fails the first time after server boot
- program with bitstream `/home/maps/felix_firmware/bitstreams/FLX711_RM0304_6CH_LOCALCLK_SVNBLAHHHH_180124_16_53.bit` (above is out of date use `FLX711_RM0304_6CH_LOCALCLK_SVNBLAHHHH_180406_14_29.bit`)

configure FELIX after programming If any step fails, reprogram; if server freezes and crashes, power cycle after it reboots itself.

```
su
cd /home/maps/meeg/felix/daq/felix_rcdaq/build/
source /home/maps/meeg/rcdaq/setup.sh;source /home/maps/meeg/felix/setup.sh
/home/maps/meeg/felix/software/pcie_hotplug/pcie_hotplug_remove.sh
/home/maps/meeg/felix/software/pcie_hotplug/pcie_hotplug_rescan.sh
/home/maps/felix_firmware/Si5345_40p8mhz_BNL-711v1p5_EXTERNAL_IN02.sh
flx-init
flx-config setraw -r 0x5890 -w 64 -v 0x1
```

run online monitoring

```
cd /home/maps/meeg/felix/daq/felix_rcdaq/online_monitoring
source /home/maps/meeg/rcdaq/setup.sh;source /home/maps/meeg/felix/setup.sh
root -l ./scripts/run_om.C
```

list pixels with more than 100 hits (noisy pixels): `process_histos(100)`

start and configure RCDAQ

```
su
cd /home/maps/meeg/felix/daq/felix_rcdaq
source /home/maps/meeg/rcdaq/setup.sh;source /home/maps/meeg/felix/setup.sh
./setup_felix_rcdaq_felixTrigger.sh
daq_set_runtype beam
```

Valid runtypes are “beam,” “junk,” and “calib.”

take a run

```
daq_begin
daq_end
```

Other useful RCDAQ commands: `daq_status` for status and event count, `daq_shutdown` to kill RCDAQ.

8.1.2 RU

Power on and program the RU.

```
source /home/maps/setup_anaconda.sh
../../modules/board_support_software/software/py/initGBTx_v1.py ../../modules/gbt/software/GBTx_
```

Program the RU again.

```
./testbench_JS_AT_SU_nopulse.py initialize_boards
```

Power on the ALPIDEs. If `setup_sensors` fails, repeat starting at `initialize_boards`.

```
./testbench_JS_AT_SU_nopulse.py setup_sensors
./testbench_JS_AT_SU_nopulse.py setup_readout
../../modules/board_support_software/software/py/wrreg.py 8 29 0x3
```

8.2 Setup

RCDAQ is only needed on the server where FELIX is installed, but the decoder and online monitoring software can be run elsewhere. The FELIX software and plugin are only needed on the server where FELIX is installed. The FELIX firmware can be installed and compiled on any computer.

These instructions assume 64-bit CentOS 7 or similar (RHEL, Scientific Linux, or CentOS CERN — some tweaks might be needed) and bash shell.

8.2.1 General System Setup

set up CERN gitlab Generate SSH keys (you will get asked some questions, just hit enter each time):

```
ssh-keygen
```

Now you should have a private key and public key in `~/.ssh`. Open `~/.ssh/id_rsa.pub` in a text editor.

Log in to the CERN gitlab website, go to settings (https://gitlab.cern.ch/profile), click on "SSH Keys." Paste your `id_rsa.pub` into the text box to add your SSH key.

As root, install git:

```
yum install git
```

Make Vivado recognize the Ethernet dongle (only needed to access licenses) To use the Ethernet dongle for licensing on CentOS, you need the interface name to be of the form "ethX." You need to change some boot settings. As root, edit `/etc/default/grub` so the `GRUB_CMDLINE_LINUX` line has the following options: `net.ifnames=0 biosdevname=0`.

Then (still as root, and the filename may be slightly different - sometimes it is /etc/grub2-efi.cfg):

```
grub2-mkconfig -o /etc/grub2.cfg
```

Now reboot. If you run 'ifconfig' the network interfaces should be "ethX," "wlanX," etc. If you run the Vivado license manager (Help->Manage licenses) the dongle's MAC address should show up under HostID.

8.2.2 FELIX Firmware

The FELIX firmware will only build on Vivado 2015.4.

Compiling the firmware Set up the Vivado environment:

```
source /opt/Xilinx/Vivado/2015.4/settings64.sh
```

Change to the scripts directory:

```
cd felix_firmware/FLX711_FW4LTDB/scripts/FELIX_top/
```

Run Vivado from the command line to create the project (this could also be done from the Tcl console in Vivado). This will take a while, maybe 5 minutes.

```
vivado -mode batch -source vivado_import_felix_bnl711_ltdb_nocr.tcl
```

Start Vivado ('vivado') and open the new project file `felix_firmware/FLX711_FW4LTDB/Projects/felix_top_ultrascale/felix_top_ultrascale.xpr`. Then in the Tcl console, run the following (path will be different if you didn't start Vivado in `FELIX_top` but the script should still work):

```
source ./do_implementation_BNL711.tcl
```

On the FELIX server, building the firmware with RU_NUM=2 (the default) takes roughly 20 minutes for synthesis and 40 minutes for implementation. Peak RAM usage is 5.6 GB.

Once the bitstream generation is complete you will find it in `felix_firmware/FLX711_FW4LTDB/output`.

8.2.3 RCDAQ

These instructions are based on those in https://www.phenix.bnl.gov/~purschke/rcdaq/rcdaq_doc.pdf.

```
mkdir rcdaq; cd rcdaq
export RCDAQ_ROOT=$PWD
git clone https://github.com/sPHENIX-Collaboration/rcdaq.git
mkdir build install
cd build
mkdir rcdaq; cd rcdaq
../../rcdaq/autogen.sh --prefix=$RCDAQ_ROOT/install
make install
```

Create a `setup.sh` script that looks like this, or add these lines to an existing `setup.sh`:

```
export RCDAQ_ROOT=<the rcdaq directory previously created>
export PATH=$PATH:$RCDAQ_ROOT/install/bin
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:$RCDAQ_ROOT/install/lib"
source $RCDAQ_ROOT/install/bin/aliases.sh
```

8.2.4 FELIX Software and Plugin

This is based on the instructions provided with the FELIX software: https://gitlab.cern.ch/atlas-tdaq-felix/software/blob/master/README.md.

All steps can be done as a regular user, except for installing dependencies.

You will end up with a `setup.sh` script that you can source (`source setup.sh`) to setup your software environment, a `cvmfs` directory containing the CERN software distribution, and a `software` directory containing the FELIX software. These instructions assume you're putting all three in `~/felixsoftware`, but they can all be in different places.

install dependencies Need Git, various FELIX software dependencies, and DKMS (needed to install the FELIX driver). DKMS is not part of the default CentOS repositories so you need to enable the EPEL repository.

As root:

```
yum install git gcc make mesa-libGL libpng libSM libXrender fontconfig
↪ xkeyboard-config redhat-lsb
yum install epel-release
yum install dkms
```

install driver Download the package from https://atlas-project-felix.web.cern.ch/atlas-project-felix/user/dist/software/driver/. Latest version should be OK (2.0.2 works).

Install the package (as root):

```
rpm -i tdaq_sw_for_Flx-2.0.2-2dkms.noarch.rpm
```

get and configure CVMFS Download one of the CVMFS tarballs from https://atlas-project-felix.web.cern.ch/atlas-project-felix/user/dist/software/cvmfs/. There are two types of tarballs, multiple versions of each: some with GCC 4.9 and some with GCC 6.2 (see the description column on the right). Download the latest GCC 6.2 tarball and unpack it to a directory of your choice.

get FELIX software Check out the software:

```
cd ~/felixsoftware
git clone ssh://git@gitlab.cern.ch:7999/sphenix-mvtx/felix_software/software
cd software
./clone_all.sh ssh
```

This directory contains a `software/setup.sh` script that adds the FELIX software to your environment. Change the first line of `software/setup.sh` so it matches where you unpacked the cvmfs tarball. For example:

```
export LCG_BASE="$HOME/felixsoftware/cvmfs/sft.cern.ch/lcg"
```

Source the script (`source setup.sh`), then create a build directory and build the software:

```
cd ~/felixsoftware/software
source setup.sh
cmake_config $FELIX_ARCH
cd $FELIX_ARCH
make
```

build the FELIX plugin On the FELIX server, source the `setup.sh` scripts for the FELIX software and RCDAQ, then:

```
git clone ssh://git@gitlab.cern.ch:7999/sphenix-mvtx/felix_rcdaq.git
cd felix_rcdaq
mkdir build
cd build
cmake ..
make
```

8.2.5 Decoder and Online Monitoring

These instructions are based on those in https://www.phenix.bnl.gov/~purschke/rcdaq/rcdaq_doc.pdf.

```
mkdir rcdaq; cd rcdaq
export RCDAQ_ROOT=$PWD
git clone https://github.com/sPHENIX-Collaboration/online_distribution.git
mkdir build install
cd build
mkdir newbasic; cd newbasic
../../online_distribution/newbasic/autogen.sh --prefix=$RCDAQ_ROOT/install
make install
cd ../build
mkdir pmonitor; cd pmonitor
../../online_distribution/pmonitor/autogen.sh --prefix=$RCDAQ_ROOT/install
make install
```

Create a `setup.sh` script that looks like this, or add these lines to an existing `setup.sh` (if you installed RCDAQ, you only need to define `ONLINE_MAIN`):

```
export RCDAQ_ROOT=<the rcdaq directory previously created>
export PATH=$PATH:$RCDAQ_ROOT/install/bin
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:$RCDAQ_ROOT/install/lib"
source $RCDAQ_ROOT/install/bin/aliases.sh
export ONLINE_MAIN=$RCDAQ_ROOT/install
```

Source the `setup.sh` script, then:

```
git clone ssh://git@gitlab.cern.ch:7999/sphenix-mvtx/felix_rcdaq.git
cd felix_rcdaq/online_monitoring
make
```

8.2.6 RU Firmware and Software

Clone the RU repository:

```
git clone ssh://git@gitlab.cern.ch:7999/sphenix-mvtx/RUv1_Test_sync2018-08.git
```

The RU firmware requires Vivado 2017.4. The `doc` directory contains some information on setting up the firmware. You may need to make the following changes:

- In the file `env_var.mk`, change the `VIVADO_VER` and `VIVADO_PATH` to match your Vivado installation
- In `modules/common/software/py/create_filelist.py`, change the line `ofile.write("%s\n" % item)` to `ofile.write(u"%s\n" % item)`

Now, `make batch` will create a Vivado project for the RU firmware. Open the project in Vivado and generate the bitstream.

Install dependencies (as root):

```
yum install centos-release-scl
yum install rh-python36 rh-python36-python-tkinter
source scl_source enable rh-python36
pip install fire pyserial pyusb imageio matplotlib
```

Add the line `source scl_source enable rh-python36` to `.bashrc`.
+

A PCIe

The purpose of this appendix is to discuss PCIe 3.0 architecture and prime the basics required to have a better overall picture regarding the Back end section. The first point to make that may make this an easier read is that the primary difference between PCI, PCI-X which behave as actual buses and PCIe can be viewed as a packet based network with communication taking the form of packets through switches etc.

PCI Express is a serial, point-to-point interface. The connection between two PCI Express devices is referred to as a link. A link consists of a number of lanes, much the same way that a highway consists of a number of driving lanes. With PCI Express, a lane is the term used for a single set of differential transmit and receive pairs. A lane contains four signals, a differential pair for unidirectional transmission in both directions (referred to as dual unidirectional). The link shown in Figure 110 is 4 lanes wide.

PCIe uses clock recovery for each signal pair, a pair receiver looks at the signal transition 0->1 or 1->0 from which it can infer the position of the surrounding bits. To resolve the problem of many successive bits of the same value, extra bits are transmitted to ensure that the single transitions are not too far apart which resyncs the clock recovery mechanism. 8b/10b for each 8 bit sent 10 bits are transmitted 20% overhead that guarantees enough signal transitions.

As shown in Figure 111 PCI Express consists of three primary devices: a root complex, a PCI Express-PCI bridge, an endpoint and a switch.

The root complex is the head or root or the host controller that connects the CPU of the host machine to the rest of the PCIe devices. PCIe has its own address space consisting of either 32 or 64 bits depending upon the Root-Complex and it is only visible by PCIe components like the Root-Complex, end-points, switches and bridges. Root-complex can interrupt the CPU for any of the events generated by the Root-Complex itself or by any of the PCIe devices. Moreover, it can also access the memory without CPU intervention (acting as a sort of DMA). PCIe end-points can use this feature to write/read data to/from the memory. In order to do so, Root-complex makes the end-point the bus master (giving the permission to access the memory) and generates the corresponding memory address.

An endpoint is a device that can request or complete PCIe transactions for itself (for example, an FPGA with PCIe endpoint controller) or on behalf of a non-PCI Express device (PCI Express-USB interface, for instance).

Switches are used to provide fan-out for the I/O bus. From a PCIe configuration standpoint, they are considered a collection of "virtual" PCIe-to-PCIe bridges whose sole purpose is to act as the traffic director between multiple links. They are responsible for properly forwarding transactions to the appropriate link. Unlike a root complex, they must always manage peer-to-peer transactions between two downstream devices (downstream meaning the side further away from the root complex).

A PCIe to PCI bridge provides forward and reverse bridging allowing designers to migrate local bus, PCI, PCI-X and USB bus interfaces to the serial PCIe architecture.

In the case of Root-Complex or switches, in order to implement a point-to-point topology (which means that a single serial link connects two devices) multiple Virtual PCI to PCI bridges are used. These are the devices that connects multiple buses together providing a (virtual) PCI bridge for the up-stream PCIe

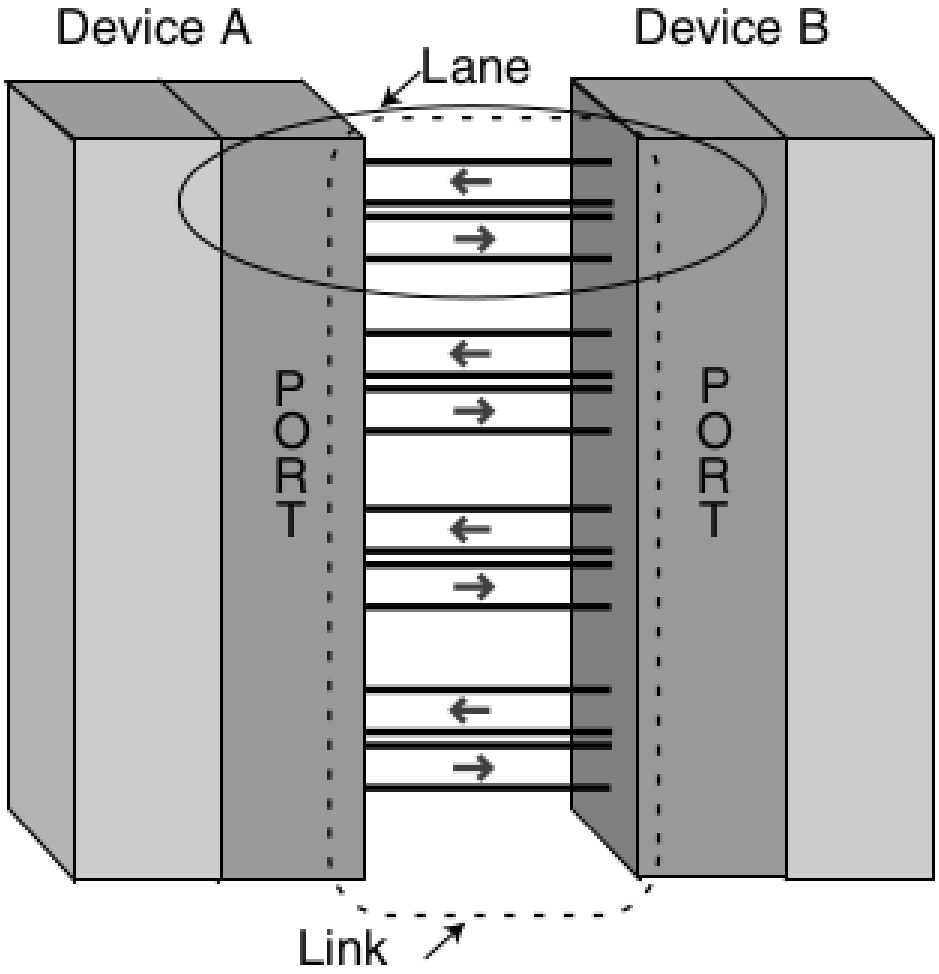


Figure 110: PCIe Links Lanes and Ports

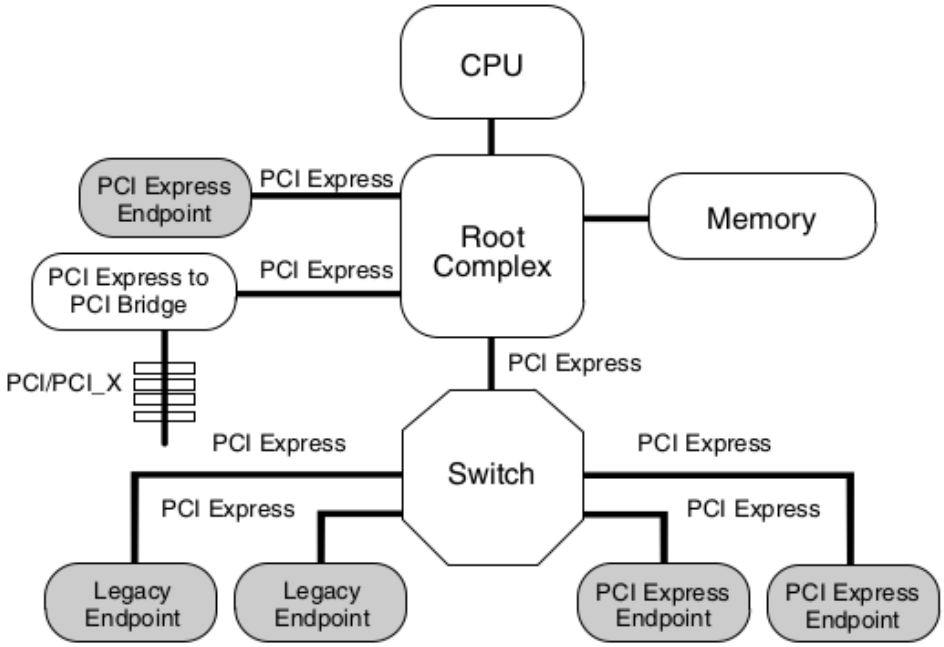


Figure 111: PCIe Topology

connection and one (virtual) PCI bridge for each down-stream PCIe connection. An identification number is assigned to each bus by the software during the enumeration process that is used by switches and bridges to identify the path of a transaction. Every switch or bridge must store the information about three bus numbers: the primary bus number (that reflects the number of the bus the switch is connected to), the secondary bus number (identifying the bus with the lowest number that can be reached) and subordinate bus number (the bus with the highest number that can be reached).

Transactions form the basis for the transportation of information between PCI Express devices. PCI Express uses a split-transaction protocol. This means that there are two transaction phases, the request and the completion. The transaction initiator, referred to as the requester, sends out the request packet. It makes its way towards the intended target of the request, referred to as the completer. For requests that require completions, the completer later sends back a completion packet (or packets) to the requester. A completion is not necessarily required for each request.

Even though PCI Express links are point-to-point, this does not always mean that one of the devices on the link is the requester and the other the completer. For example, say that the root complex in Figure 111 wants to communicate with a PCIe endpoint that is downstream of the switch. The root complex is the requester and the endpoint is the completer. Even though the switch receives the transaction from the root complex, it is not considered a completer of that transaction. Even though the endpoint receives the transaction from the switch, it does not consider the switch to be the requester of that transaction. The requester identifies itself within the request packet it sends out, and this informs the completer (and/or switch) where it should return the completion packets (if needed)

The PCI Express architecture defines four transaction types: memory, I/O, configuration, and message. This is similar to the traditional PCI transactions, with the notable difference being the addition of a message transaction type.

Memory Transactions Transactions targeting the memory space transfer data to or from a memory-mapped location. There are several types of memory transactions: Memory Read Request, Memory Read Completion, and Memory Write Request. Memory transactions use one of two different address formats, either 32-bit addressing (short address) or 64-bit addressing (long address).

I/O Transactions Transactions targeting the I/O space transfer data to or from an I/O mapped location. PCI Express supports this address space for compatibility with existing devices that utilize this space. There are several types of I/O transactions: I/O Read Request, I/O Read Completion, I/O Write Request, and I/O Write Completion. I/O transactions use only 32-bit addressing (short address format).

Configuration Transactions Transactions targeting the configuration space are used for device configuration and setup. These transactions access the configuration registers of PCI Express devices. For each function of each device, PCI Express defines a configuration register block four times the size of PCI. There are several types of configuration transactions: Configuration Read Request, Configuration Read Completion, Configuration Write Request, and Configuration Write Completion.

Message Transactions PCI Express adds a new transaction type to communicate a variety of miscellaneous messages between PCI Express devices. Referred to simply as messages, these transactions are used for things like interrupt signaling, error signaling or power management. This address space is a new addition for PCI Express and is necessary since these functions are no longer available via sideband signals such as PME#, IERR#, and so on. Described in further details after the layer defintions

Figure 112 shows the three abstract layers that "build" a PCI Express transaction.

Transaction Layer is the first layer whose main responsibility is to begin the process of turning requests or completion data from the device core into a PCI Express transaction. In other words it is responsible for turning that request/data into a Transaction Layer Packet (TLP). A TLP is simply a packet that is sent from the Transaction Layer of one device to the Transaction Layer of the other device. The TLP uses a header to identify the type of transaction that it is (for example, I/O versus memory, read versus write, request versus completion, and so on).The Transaction Layer also has several other functions, such as flow control and power management. Lastly the transaction layer is how a given software device driver communicates with an FPGA where the software device driver communicates with the core and the core sends the message to the FPGA via the three layers. Communication in the opposite direction works the same way.

The Data Link Layer is the middle PCI Express architectural layer and interacts with both the Physical Layer and the Transaction Layer. The main responsibility of this layer is to ensure that the transactions going back and forth across the link are received properly. This layer receives TLPs from the transmit side of the Transaction Layer and continues the process of building that into a PCI Express trans- action. It does this by adding a sequence number to the front of the packet and an LCRC error checker to the end. The sequence number serves the purpose of making sure that each packet makes it across the link. For example, if the last sequence number that Device A successfully received was #6, it expects the next packet to have a sequence number of 7. If it instead sees #8, it knows that packet #7 got lost somewhere and notifies Device B of the error. The LCRC serves to make sure that each packet makes it across intact. As mentioned previously, if the LCRC does not check out at the receiver side, the device knows that there was a bit error sometime during the transmission of this packet. This scenario also generates an error condition. Once the transmit side of the Data Link Layer applies the sequence number and LCRC to the TLP, it submits them to the Physical Layer. The receiver side of the Data Link Layer accepts incoming packets from the Physical Layer and checks the sequence number and LCRC to make sure the packet is correct. If it is correct, it then passes it up to the receiver side of the Transaction Layer. If an error occurs (either wrong sequence number or bad data), it does not pass the packet on to the Transaction Layer until the issue has been resolved. In this way, the Data Link Layer acts a lot like the security guard of the link. It makes sure that only the packets

that are "supposed to be there" are allowed through. The Data Link Layer is also responsible for several link management functions. To do this, it generates and consumes Data Link Layer Packets (DLLPs). Unlike TLPs, these packets are created at the Data Link Layer.

Finally, the lowest PCI Express architectural layer is the Physical Layer. This layer is responsible for actually sending and receiving all the data to be sent across the PCI Express link. The Physical Layer interacts with its Data Link Layer and the physical PCI Express link (wires, cables, optical fiber, and so on). This layer contains all the circuitry for the interface operation: input and output buffers, parallel-to-serial and serial-to-parallel converters, PLL(s) and impedance matching circuitry. It also contains some logic functions needed for interface initialization and maintenance. On the transmit side of things, the Physical Layer takes information from the Data Link Layer and converts it into the proper serial format. It goes through an optional data scrambling procedure, 8-bit/10-bit conversion, and parallel-to-serial conversion. The Physical Layer also adds framing characters to indicate the beginning and ending of a packet. It is then sent out across the link using the appropriate transfer rate (for example 2.5 gigahertz) as well as link width (for example, using 4 lanes if the link is a x4).On the receive side, the Physical Layer takes the incoming serial stream from the link and turns it back into a chunk of data to be passed along to its Data Link Layer. This procedure is basically the reverse of what would occur on the transmit side. It samples the data, removes the framing characters, descrambles the remaining data characters and then converts the 8-bit/10-bit data stream back into a parallel data format.

The three architectural build layers accomplish this by "building up" the packets into a full scale PCI Express transaction. This buildup is shown in Figure 113

Each layer has specific functions and contributes to a portion of the pyramid. As a transaction flows through the transmitting PCI Express device, each layer adds on its specific information. The Transaction Layer generates a header and adds the data payload (if required) and an optional ECRC (end-to-end CRC). The Data Link Layer adds the sequence number and LCRC (link CRC). The Physical Layer frames it for proper transmission to the other device. When it gets to the receiver side, the complete reversal of this build occurs. The Physical Layer decodes the framing and passes along the sequence number, header, data, ECRC, and LCRC to its Data Link Layer. The Data Link Layer checks out the sequence number and LCRC, and then passes the header, data, and ECRC on to the Transaction Layer. The Transaction Layer decodes the header and passes the appropriate data on to its device core.

As previosuly mentioned memory, IO and configuration transactions are supported PCIe architectures, but the message transaction is new to PCIE. Transactions are defined as a series of one or more packet transmissions required to complete an information transfer between a requester and a completer. Figure 114 is a more detailed list of transactions. These transactions can be categorized into non-posted transactions and posted transactions.

Non-posted transactions

For Non-posted transactions, a requester transmits a TLP request packet to a completer. At a later time, the completer returns a TLP completion packet back to the requester. Non-posted transactions are handled as split transactions. The purpose of the completion TLP is to confirm to the requester that the completer has received the request TLP. In addition, non-posted read transactions contain data in the completion TLP. Non-Posted write transactions contain data in the write request TLP.

Posted transactions

For Posted transactions, a requester transmits a TLP request packet to a completer. The completer however does NOT return a completion TLP back to the requester. Posted transactions are optimized for best performance in completing the transaction at the expense of the requester not having knowledge of successful reception of the request by the completer. Posted transactions may or may not contain data in the

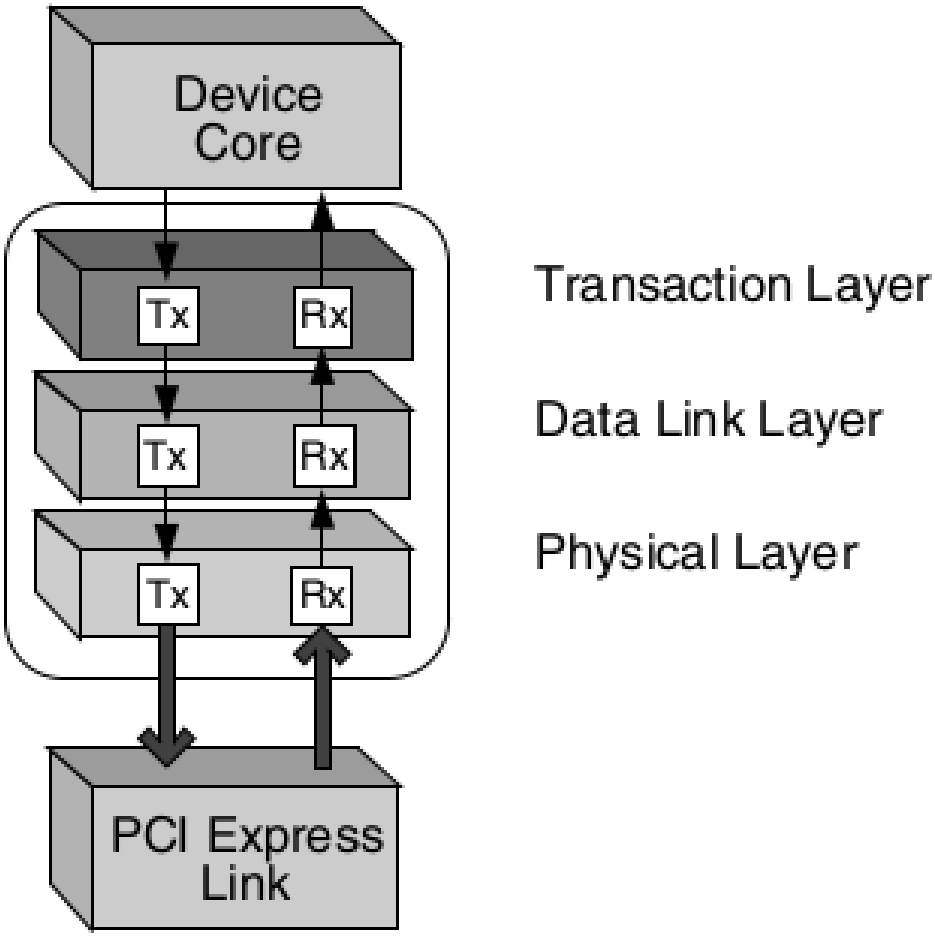


Figure 112: PCIe Three Architectural Layers

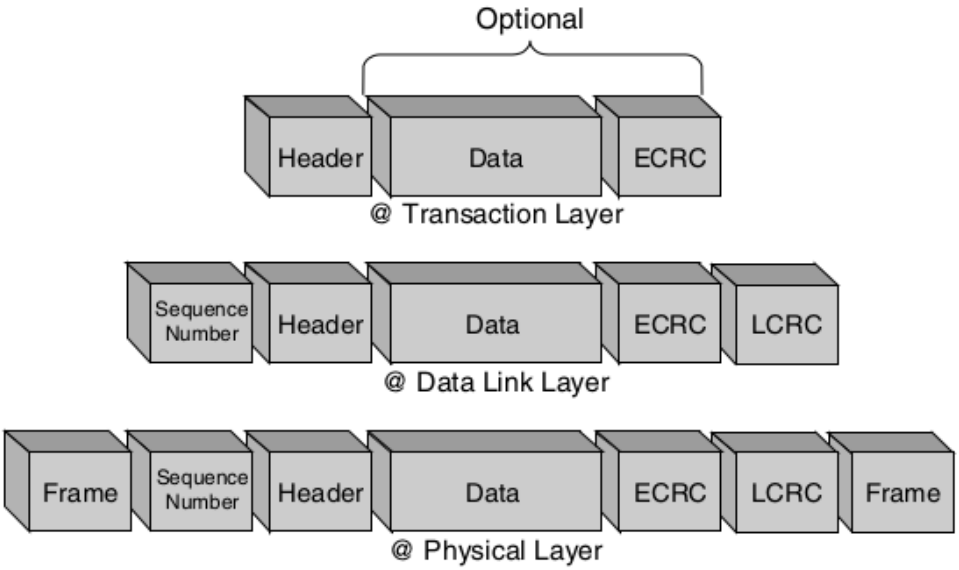


Figure 113: PCIe Transaction Buildup

Transaction Type	Non-Posted or Posted
Memory Read	Non-Posted
Memory Write	Posted
Memory Read Lock	Non-Posted
IO Read	Non-Posted
IO Write	Non-Posted
Configuration Read (Type 0 and Type 1)	Non-Posted
Configuration Write (Type 0 and Type 1)	Non-Posted
Message	Posted

Figure 114: PCIe non-posted and posted transactions

request TLP.

TLP Packet Types	Abbreviated Name
Memory Read Request	MRd
Memory Read Request - Locked access	MRdLk
Memory Write Request	MWr
IO Read	IORd
IO Write	IOWr
Configuration Read (Type 0 and Type 1)	CfgRd0, CfgRd1
Configuration Write (Type 0 and Type 1)	CfgWr0, CfgWr1
Message Request without Data	Msg
Message Request with Data	MsgD
Completion without Data	Cpl
Completion with Data	CplD
Completion without Data - associated with Locked Memory Read Requests	CplLk
Completion with Data - associated with Locked Memory Read Requests	CplDLk

Figure 115: PCIe TLP Packet Types

Figure 115 lists all of the TLP request and TLP completion packets. These packets are used in the transactions referenced in Figure 114.

Non-Posted Read Transactions

Figure 116 shows the packets transmitted by a requester and completer to complete a non-posted read transaction. To complete this transfer, a requester transmits a non-posted read request TLP to a completer it intends to read data from. Non-posted read request TLPs include memory read request (MRd), IO read request (IORd), and configuration read request type 0 or type 1 (CfgRd0, CfgRd1) TLPs. Requesters may be root complex or endpoint devices (endpoints do not initiate configuration read/write requests however).

The request TLP is routed through the fabric of switches using information in the header portion of the TLP. The packet makes its way to a targeted completer. The completer can be a root complex, switches, bridges or endpoints.

When the completer receives the packet and decodes its contents, it gathers the amount of data specified in the request from the targeted address. The completer creates a single completion TLP or multiple completion TLPs with data (CplD) and sends it back to the requester. The completer can return up to 4 KBytes of data per CplD packet.

The completion packet contains routing information necessary to route the packet back to the requester.

This completion packet travels through the same path and hierarchy of switches as the request packet.

Requesters uses a tag field in the completion to associate it with a request TLP of the same tag value it transmitted earlier. Use of a tag in the request and completion TLPs allows a requester to manage multiple outstanding transactions.

If a completer is unable to obtain requested data as a result of an error, it returns a completion packet without data (Cpl) and an error status indication. The requester determines how to handle the error at the software layer.

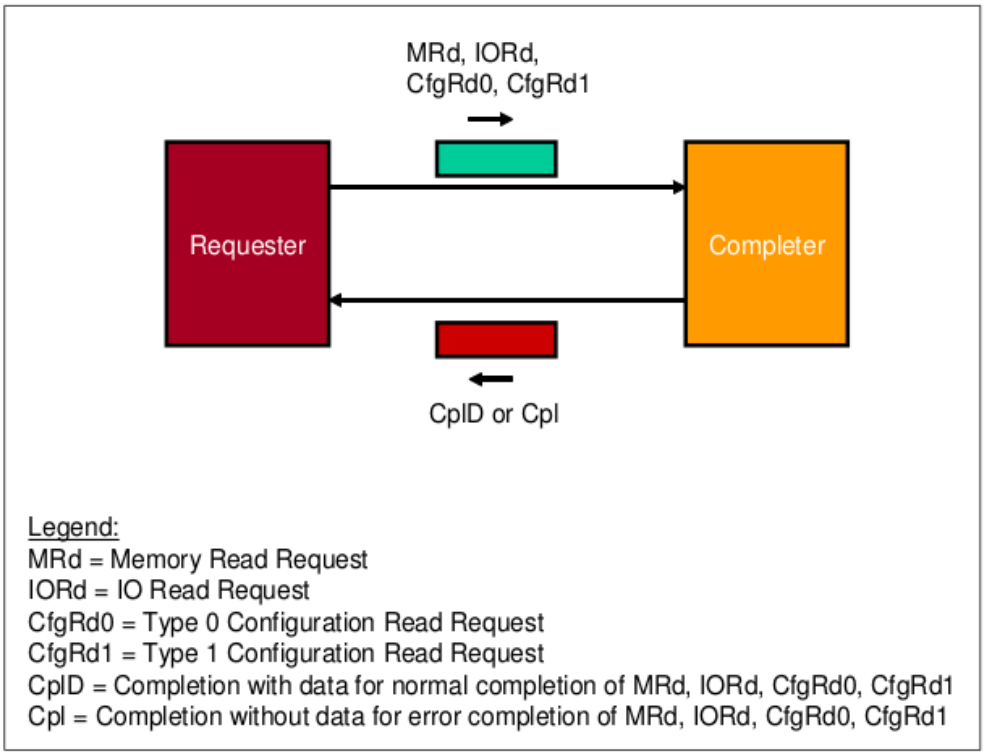


Figure 116: Non-Posted Read Transaction Protocol

Non-Posted Read Transaction for Locked Requests

Figure 117 shows packets transmitted by a requester and completer to complete a non-posted locked read transaction. To complete this transfer, a requester transmits a memory read locked request (MRdLk) TLP. The requester can only be a root complex which initiates a locked request on the behalf of the CPU. Endpoints are not allowed to initiate locked requests.

The locked memory read request TLP is routed downstream through the fabric of switches using information in the header portion of the TLP. The packet makes its way to a targeted completer. The completer can only be a legacy endpoint. The entire path from root complex to the endpoint (for TCs that map to VC0) is locked including the ingress and egress port of switches in the pathway.

When the completer receives the packet and decodes its contents, it gathers the amount of data specified in the request from the targeted address. The completer creates one or more locked completion TLP with data (CplDLk) along with a completion status. The completion is sent back to the root complex requester via the path and hierarchy of switches as the original request.

The CplDLk packet contains routing information necessary to route the packet back to the requester. Requesters uses a tag field in the completion to associate it with a request TLP of the same tag value it transmitted earlier. Use of a tag in the request and completion TLPs allows a requester to manage multiple outstanding transactions.

If the completer is unable to obtain the requested data as a result of an error, it returns a completion packet without data (CplLk) and an error status indication within the packet. The requester who receives the error notification via the CplLk TLP must assume that atomicity of the lock is no longer guaranteed and thus determine how to handle the error at the software layer.

The path from requester to completer remains locked until the requester at a later time transmits an unlock message to the completer. The path and ingress/egress ports of a switch that the unlock message passes through are unlocked.

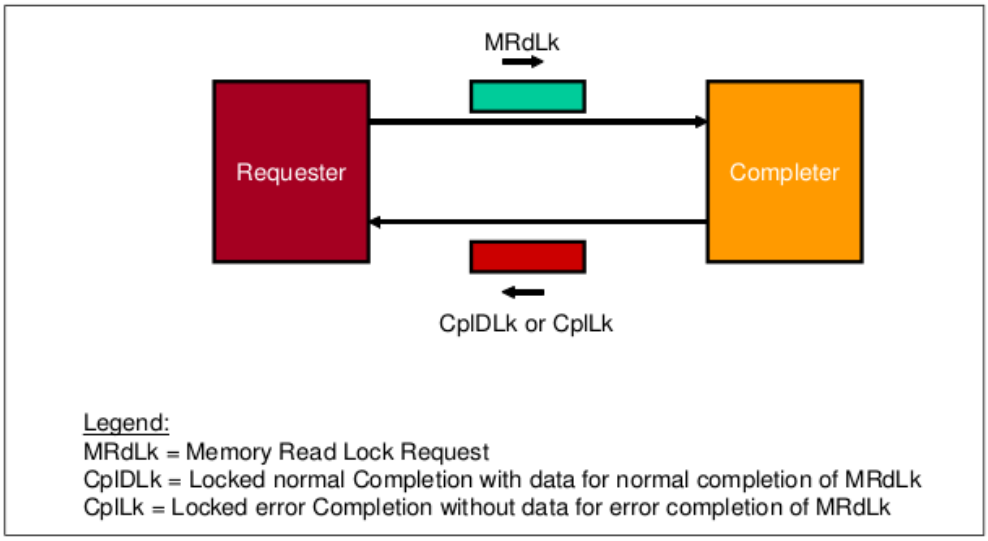


Figure 117: Non-Posted Locked Read Transaction Protocol

Non-Posted Write Transactions

Figure 118 shows the packets transmitted by a requester and completer to complete a non-posted write transaction. To complete this transfer, a requester transmits a non-posted write request TLP to a completer it intends to write data to. Non-posted write request TLPs include IO write request (IOWr), configuration write request type 0 or type 1 (CfgWr0, CfgWr1) TLPs. Memory write request and message requests are posted requests. Requesters may be a root complex or endpoint device (though not for configuration write requests).A request packet with data is routed through the fabric of switches using information in the header of the packet. The packet makes its way to a completer.

When the completer receives the packet and decodes its contents, it accepts the data. The completer creates a single completion packet without data (Cpl) to confirm reception of the write request. This is the purpose of the completion.

The completion packet contains routing information necessary to route the packet back to the requester. This completion packet will propagate through the same hierarchy of switches that the request packet went through before making its way back to the requester. The requester gets confirmation notification that the write request did make its way successfully to the completer.

If the completer is unable to successfully write the data in the request to the final destination or if the write request packet reaches the completer in error, then it returns a completion packet without data (Cpl) but with an error status indication. The requester who receives the error notification via the Cpl TLP determines how to handle the error at the software layer.

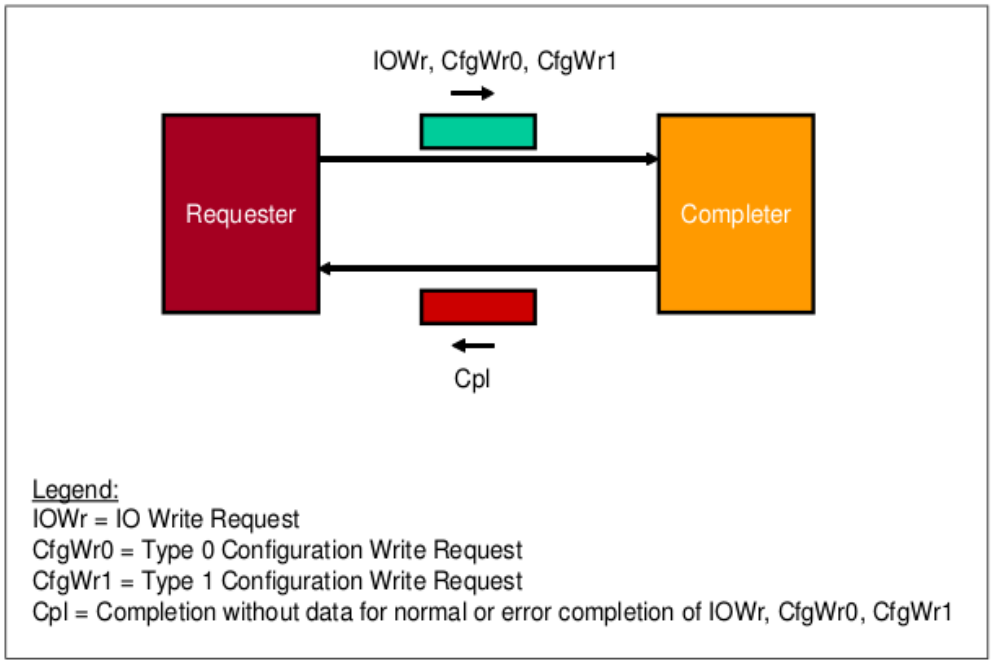


Figure 118: Non-Posted Write Transaction Protocol

Posted Memory Write Transactions

Memory write requests shown in Figure 118 are posted transactions. This implies that the completer returns no completion notification to inform the requester that the memory write request packet has reached its destination successfully. No time is wasted in returning a completion, thus back-to-back posted writes complete with higher performance relative to non-posted transactions. The write request packet which contains data is routed through the fabric of switches using information in the header portion of the packet. The packet makes its way to a completer. The completer accepts the specified amount of data within the packet. Transaction over.

If the write request is received by the completer in error, or is unable to write the posted write data to the final destination due to an internal error, the requester is not informed via the hardware protocol. The completer could log an error and generate an error message notification to the root complex. Error handling software manages the error.

Posted Message Transactions

Message requests are also posted transactions as pictured in Figure 120 There are two categories of message request TLPs, Msg and MsgD. Some message requests propagate from requester to completer, some are broadcast requests from the root complex to all endpoints, some are transmitted by an endpoint to the root complex. Message packets may be routed to completer(s) based on the message's address, device ID or routed implicitly.

The completer accepts any data that may be contained in the packet (if the packet is MsgD) and/or

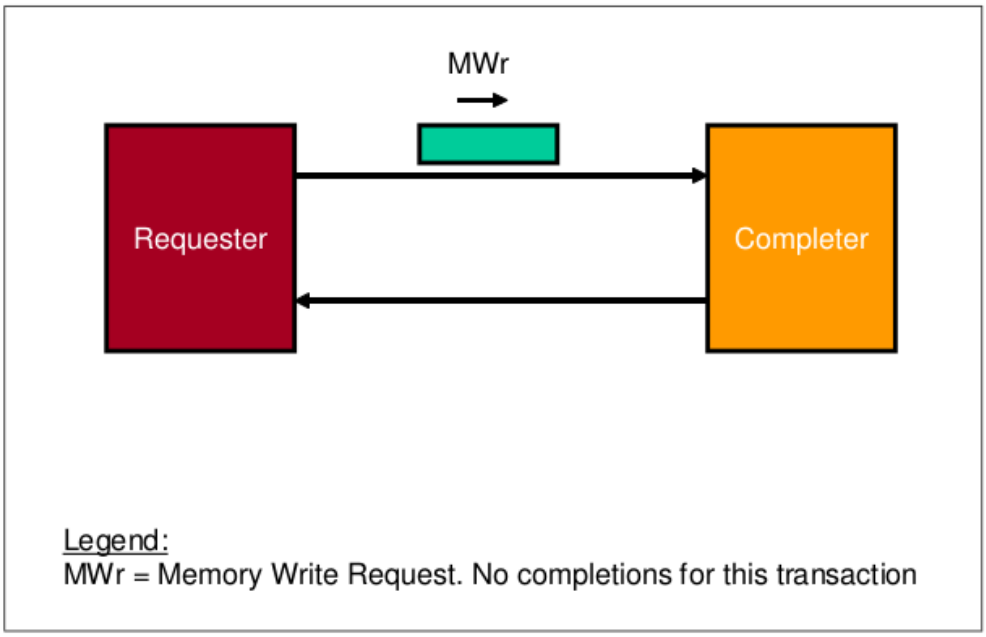


Figure 119: Posted Memory Write Transaction Protocol

performs the task specified by the message. Message request support eliminates the need for side-band signals in a PCI Express system. They are used for PCI style legacy interrupt signaling, power management protocol, error signaling, unlocking a path in the PCI Express fabric, slot power support, hot plug protocol, and vender defined purposes.

PCIe Address Space The cpu of the host system uses the root complex to accesses PCIe end-points by using the virtual PCIe Address Space. The primary parts of a Root Complex are the Configuration Space (Config, I/O, Memory, Message defined above), Registers Space, Address Translation Unit, and Configurable address space. Configuration Space contains all the information regarding end-points such as device ID and vendor ID and has also registers to configure the end-points it is 4kB. The first 64bytes of the 4kb configuration space is refereed to as the standard header and comes in two flavors; type 1 (containing info regarding root-ports, bridges and switches (such as primary, secondary and subordinate bus numbers)) and type 0 (containing info regarding end-points). The configuration registers are used to configure the various parameters of the hardware such as lane widths PCIe generation paramaeters and a way to configure the Address Translation Unit etc. The Address Translation Unit translates CPU addresses to to PCIe addresses. The configurable address space is used by the CPU in order to access the PCIe address space.

The PCIe endpoint also contains a configuration space of 4kb and the first 64 bytes will contain header type 0 information but will have a different header of the type 1 described in the the root complex. The type 0 header will contain Device ID and Vendor ID as well as the status and command space which the host system will control or configure the PCIe endpoint. The class code which differentiates whether its a type 0 header or type 1 header. It will also contain the Base Address Registers which are used to configure the memory or I/O space. The header also contains a capability register pointer and interrupt pin information. The 64 bit header of type 0 for the endpoint and type 1 for the root complex are shown in fig foo

Type0 header partial description highlightning important fields Command - host system configure or control PCIe endpoint. Header type - differentiates type 0 or type 1 Base address registers which is used to

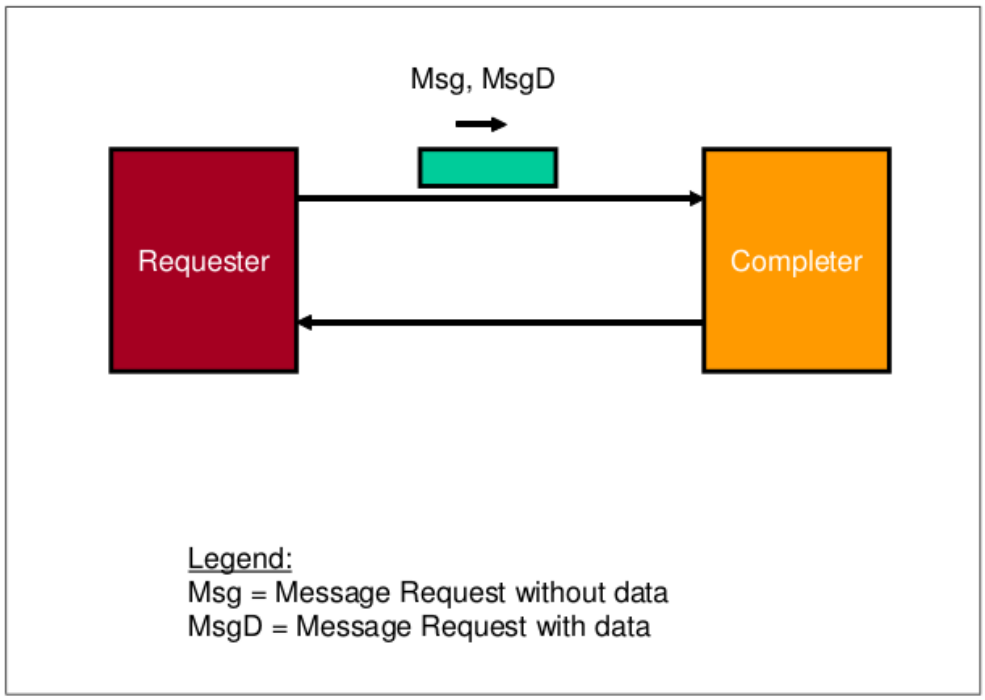


Figure 120: Posted Message Transaction Protocol

configure the memory space or io space pointer to the capability registers and interrupt pin information
Type1 header will be explained at a later time.
Figure 121 will show the Address translation between the CPU and the PCIe enpoint.

1. In order for the host system to read the configuration space of the PCIe endpoint during Enumeration process, the enpoint has to be mapped in the PCIe Address Space. The mapping is done through the type 0 header described above.

2. The mechanism that determines the address to which the configuration space of a particular endpoint should be mapped is called the Enhanced Configuration Access Mechanism (ECAM). ECAM creates a PCIe address using the bus number, device number, function number, and then the configuration space of the particular endpoint is mapped onto this address created by ECAM.

   So if the PCIe endpoint is present in Bus 1 Device 0 Function 0, PCIe address is creadted using ECAM. Which in this case is 100000h and the configuration of space of the enpoint is mapped onto this address of the PCIe address space therefore the host system can acccess it.

3. Now the host system can access the same address in the PCIe address space in order to read the configuration space of this endpoint. This is done by the CPU making use of the configurable space in the root complex. The root complex allocates 4kb in the configurable address space for for CFG 0 by using the address translation table.

4. The Address translation unit records the mapping of the Physical to virtual address. The source Address is A and the destination address is the ECAM address which is 100000h.

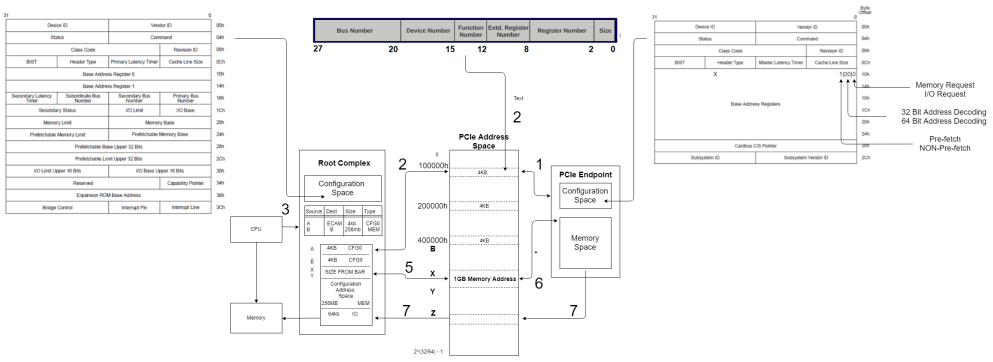


Figure 121: PCIe Address Space

5. Once the host system understand how much memory the PCIe endpoint is requesting it will allocate the the requested amount in the Configurable address Space in the dedicated memory space region. This space will then be used to access the PCIe endpoint. The Address Trabsaktuib Unit will map the configurable region in the Configurable address Space in the dedicated memory space region and take note of the source and destination.

6. So due to the mapping the root complex has access to the virtual address but the PCIe endpoint has to be configured

7. Then the Root Complex will make the PCIe endpoint he Bus Master giving it permission to access the Memory. Once completed the PCIe endpiont can directly read and write the host memory without CPU intervention becuase there is a 1:1 memory mapping i.e. DMA (Direct Memory Access).

B AXI

AXI4-Stream used for streaming packets between nodes. Unlike the memory-mapped buses transactions are not targeting an address. They consist of one or many data-beats which are grouped to packets. The below definitions should help the reader better understand the DMA transactions described in the Back end section.

beat - an individual data transfer within an AXI burst.

An AXI4-Stream interface signal definitions: axi_clk - interface clock. All other signals are synchronous to this clock meaning that they are sampled at a rising edge of this clock. axi_tvalid - asserted by the master in the same clock cycles as data signal, indicates that the master is driving a valid transfer. A transfer takes place when both TVALID and TREADY are asserted. axi_tdata - holds a valid value. axi_tready - asserted by the slave as soon as it is ready to accept data. A data beat is transferred as soon as both the axi_tvalid and axi_tready are simultaneously asserted. axi_tready - indicates that the slave can accept a transfer in the current cycle. Might depend on axi_tvalid but not the other way around. axi_tlast - asserted by the master together with axi_tvalid to mark the end of a packet. Indicates the boundary of a packet. axi_tdata - driven by the master and contains the data beats. Xilinx requires this signal to have a width which is a multiple of 8 bit. Primary payload that is used to provide the data that is passing across the interface. The width of the data payload is an integer number of bytes.

Handshake process The TVALID and TREADY handshake determines when information is passed across the interface. A two-way flow control mechanism enables both the master and slave to control the rate at which the data and control information is transmitted across the interface. For a transfer to occur both the TVALID and TREADY signals must be asserted. Either TVALID or TREADY can be asserted first or both can be asserted in the same ACLK cycle.

A master is not permitted to wait until TREADY is asserted before asserting TVALID. Once TVALID is asserted it must remain asserted until the handshake occurs.

A slave is permitted to wait for TVALID to be asserted before asserting the corresponding TREADY.

If a slave asserts TREADY, it is permitted to deassert TREADY before TVALID is asserted.

axi_tdest - provides routing information for the data stream. axi_tid - data stream identifier that indicates different streams of data. axi_tuser - user defined sideband information that can be transmitted alongside the data stream. axi_tkeep - byte qualifier that indicates whether the content of the associated byte of TDATA is processed as part of the data stream. Associated bytes that have the TKEEP byte qualifier deasserted are null bytes and can be removed from the data stream.

axi_tstrb - byte qualifier that indicates whether the content of the associated byte of TDATA is processed as a data byte or a position byte. ARESETn - active-LOW

AXI4-Stream definitions Transfer - A single transfer of data across an AXI4-Stream interface. A single transfer is defined by a single TVALID, TREADY handshake Packet - A group of bytes that are transported together across an AXI4-Stream interface. A packet is similar to an AXI4 burst. A packet may consist of a single transfer or multiple transfers. Infrastructure components can use packets to deal more efficiently with a stream in packet-sized groups Frame - The highest level of byte grouping in an AXI4-Stream. A frame contains an integer number of packets. A frame can be a very large number of bytes, for example an entire video frame buffer. Data Stream - The transport of data from one source to one destination. A data stream can be; a series of individual byte transfers or a series of byte transfers grouped together in packets Byte stream - A byte stream is the transmission of a number of data and null bytes. On each TVALID, TREADY handshake, any number of data bytes can be transferred. Null bytes have no meaning and can be inserted or removed from the stream.

C PCIe Endpoint Configuration

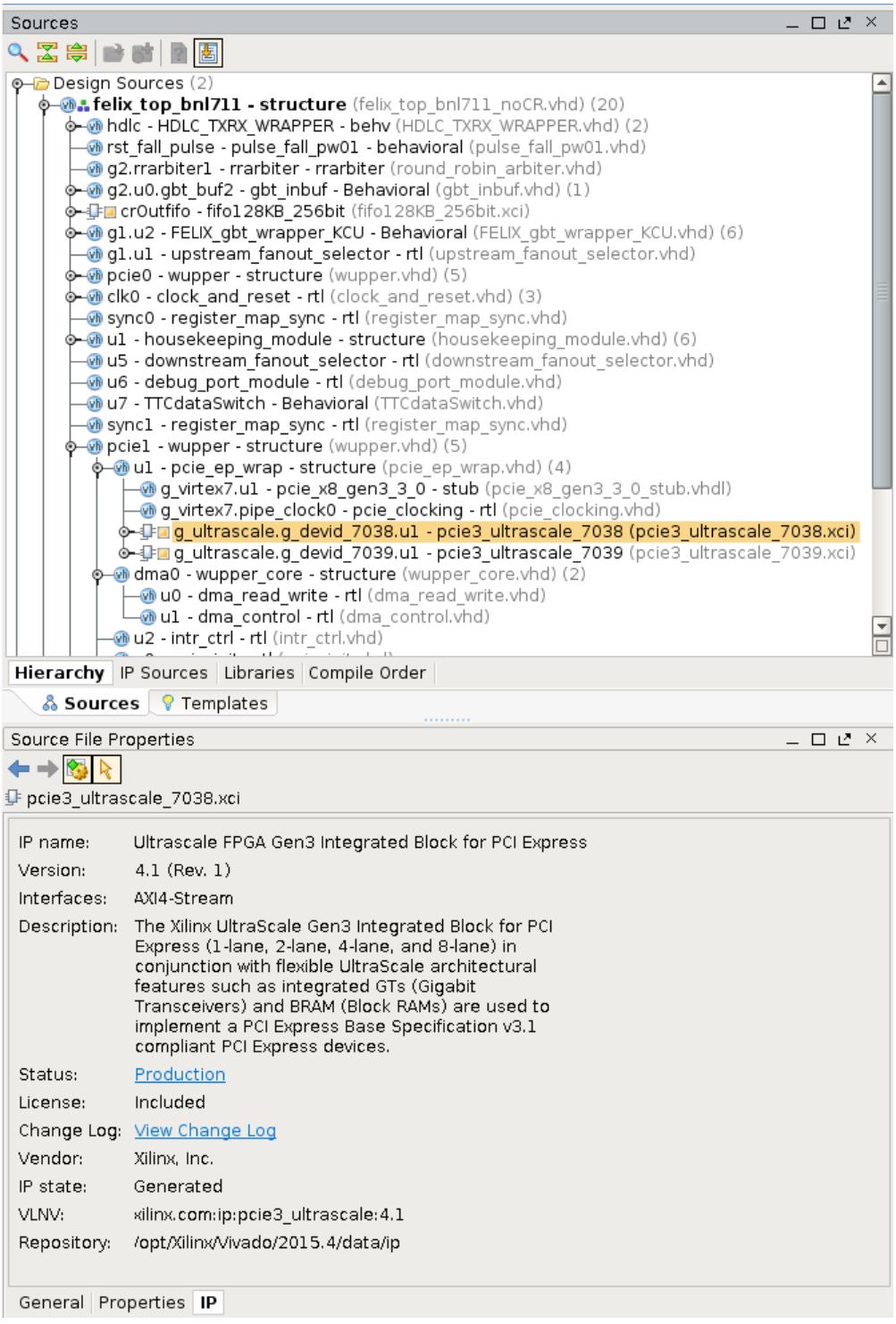


Figure 122: Ultrascale FPGA Gen3 Integrated Block for PXI Express (4.1)

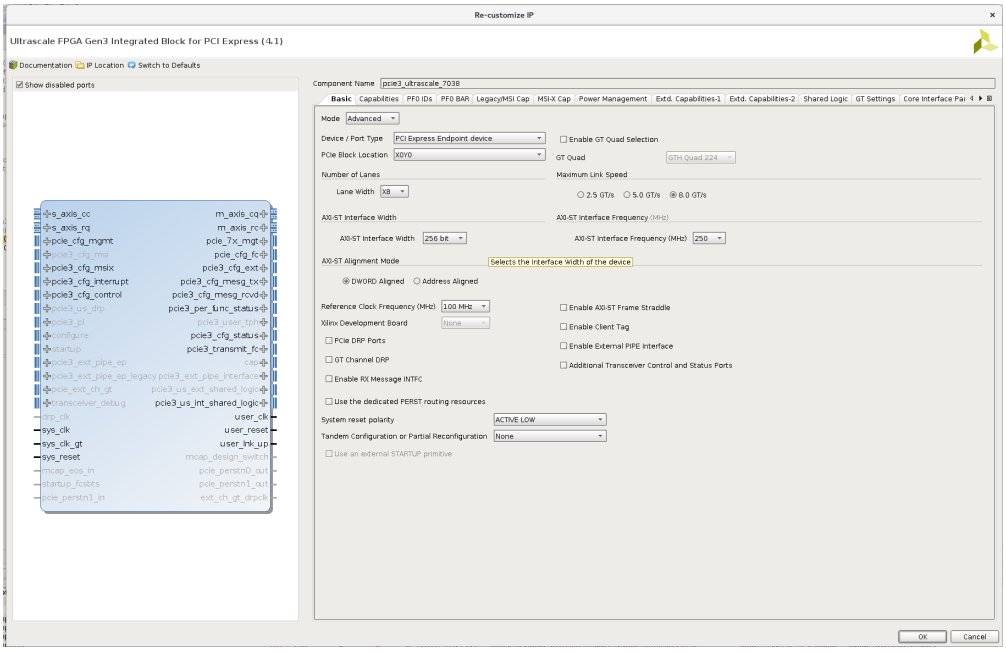


Figure 123: Ultrascale FPGA Gen3 Integrated Block for PXI Express (4.1)

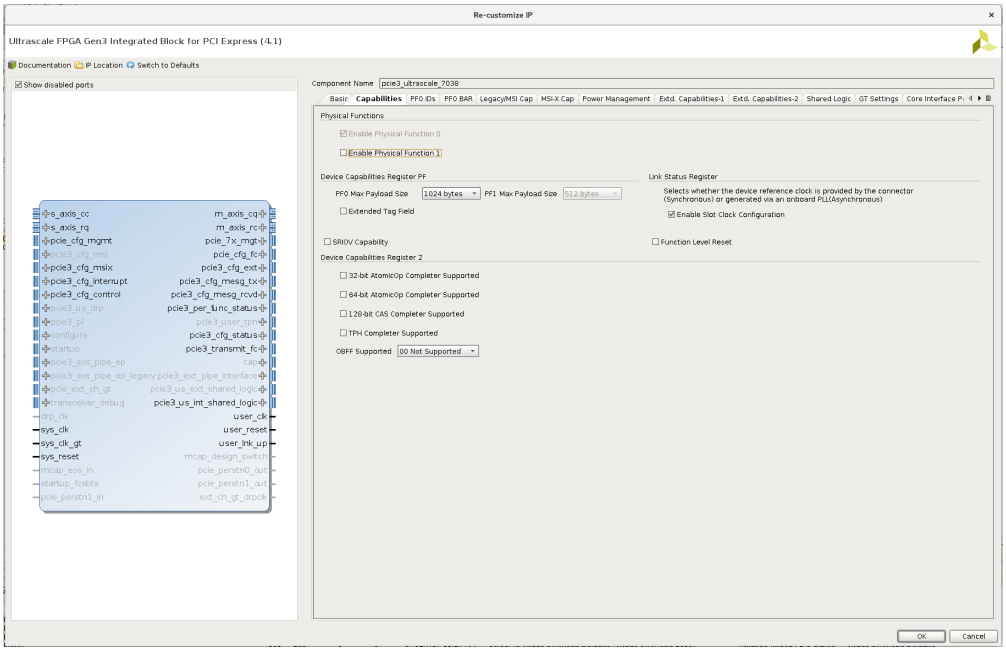


Figure 124: Ultrascale FPGA Gen3 Integrated Block for PXI Express (4.1)

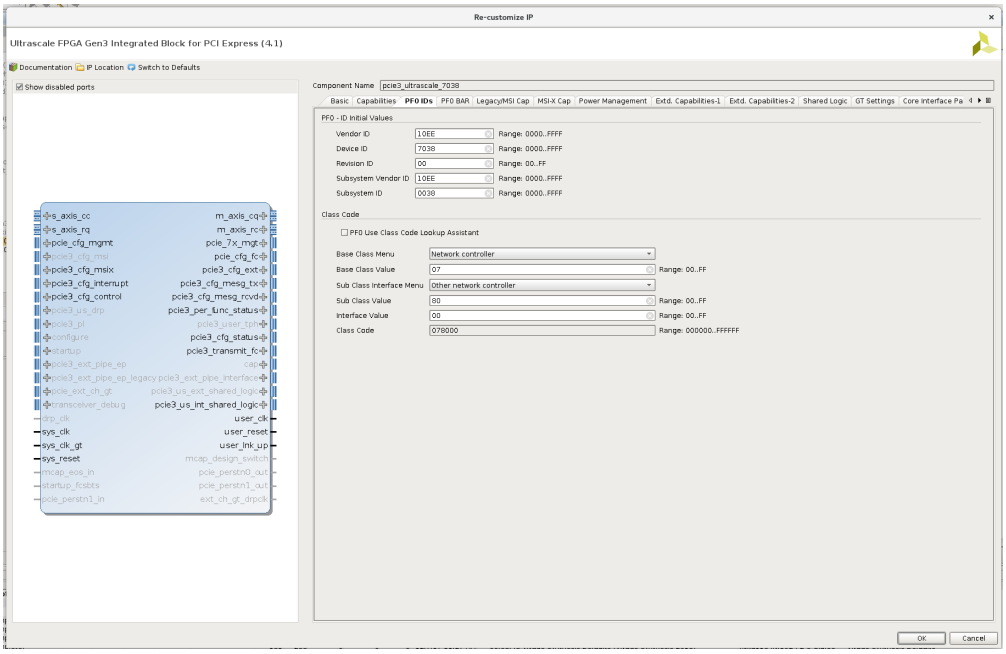


Figure 125: Ultrascale FPGA Gen3 Integrated Block for PXI Express (4.1)

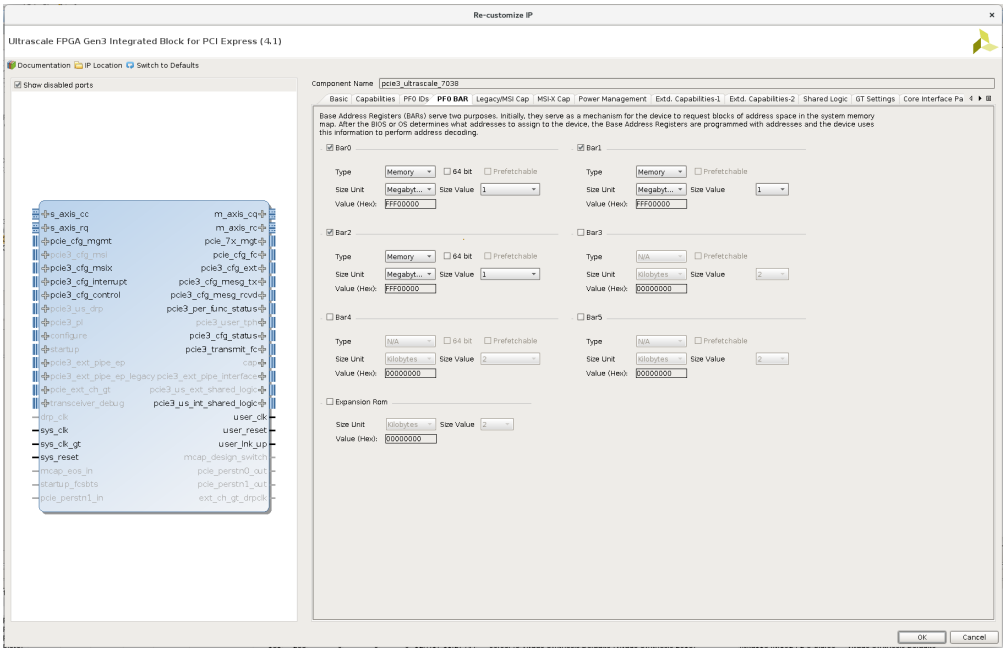


Figure 126: Ultrascale FPGA Gen3 Integrated Block for PXI Express (4.1)

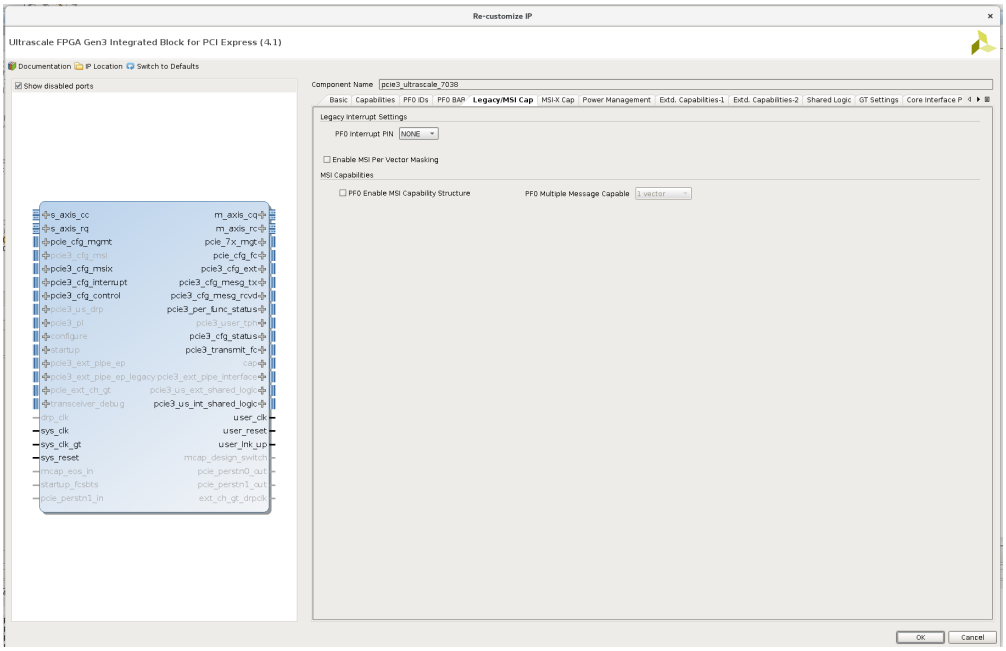


Figure 127: Ultrascale FPGA Gen3 Integrated Block for PXI Express (4.1)

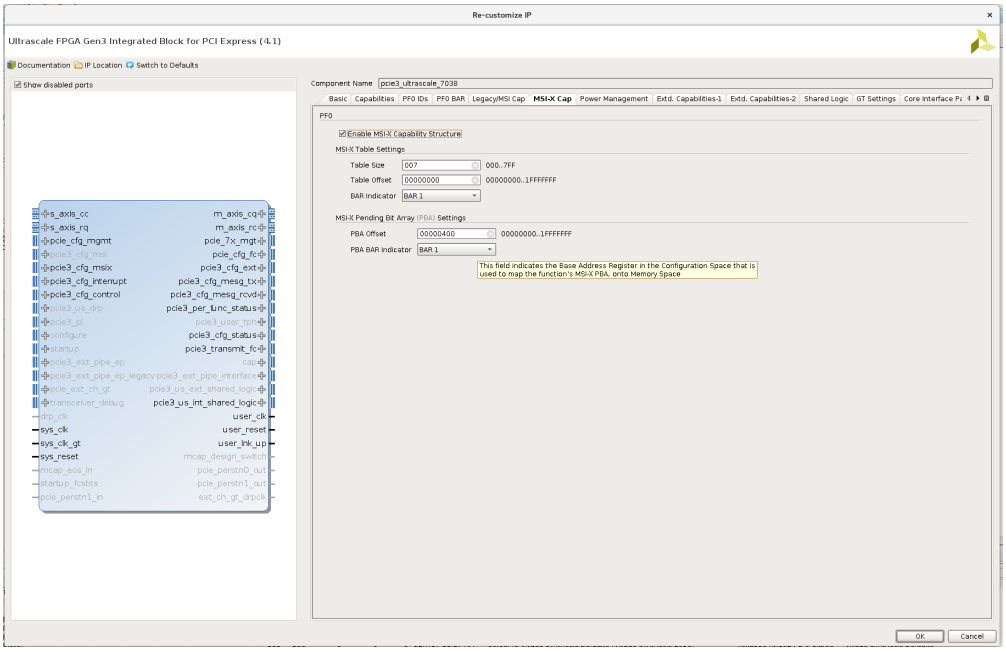


Figure 128: Ultrascale FPGA Gen3 Integrated Block for PXI Express (4.1)

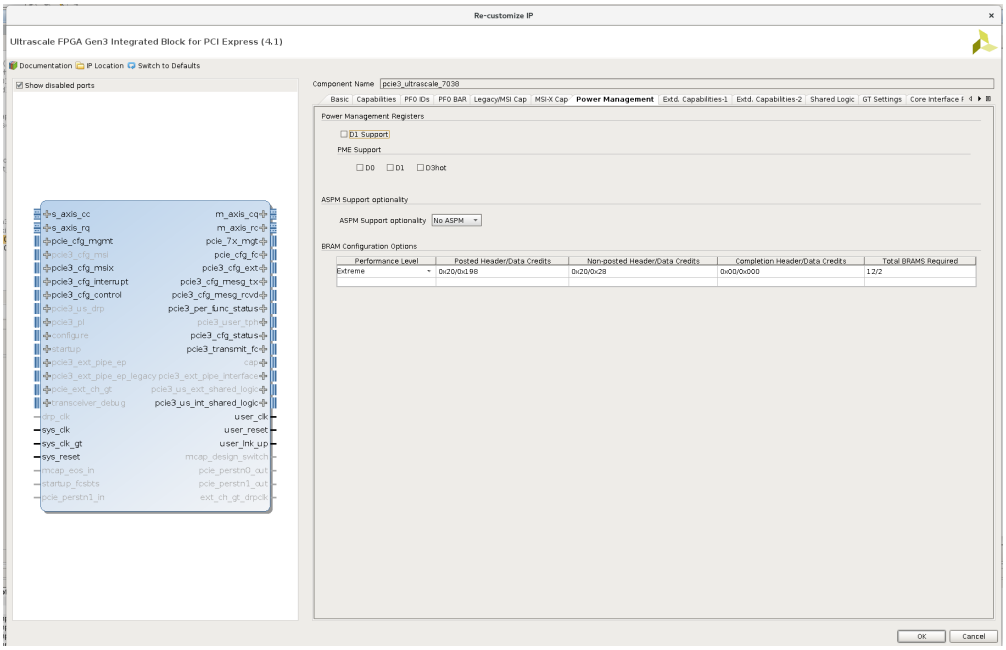


Figure 129: Ultrascale FPGA Gen3 Integrated Block for PXI Express (4.1)

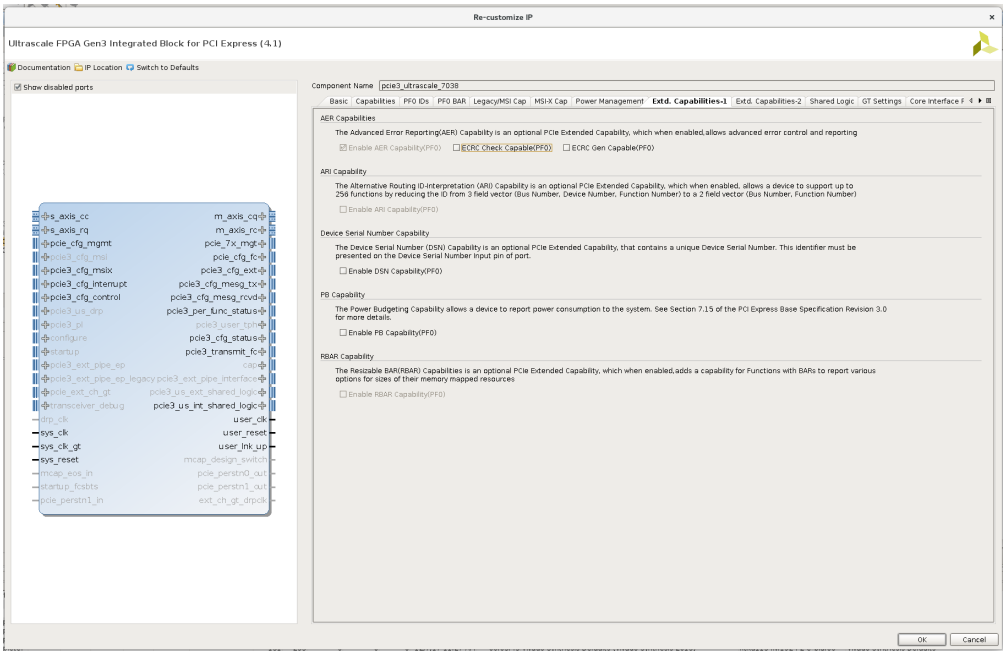


Figure 130: Ultrascale FPGA Gen3 Integrated Block for PXI Express (4.1)

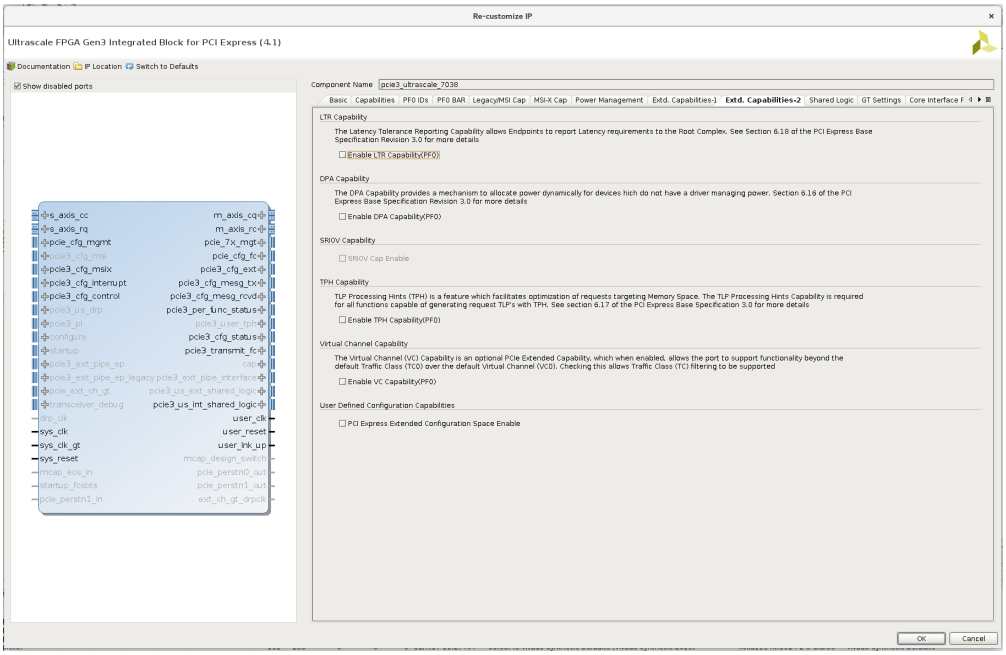


Figure 131: Ultrascale FPGA Gen3 Integrated Block for PXI Express (4.1)

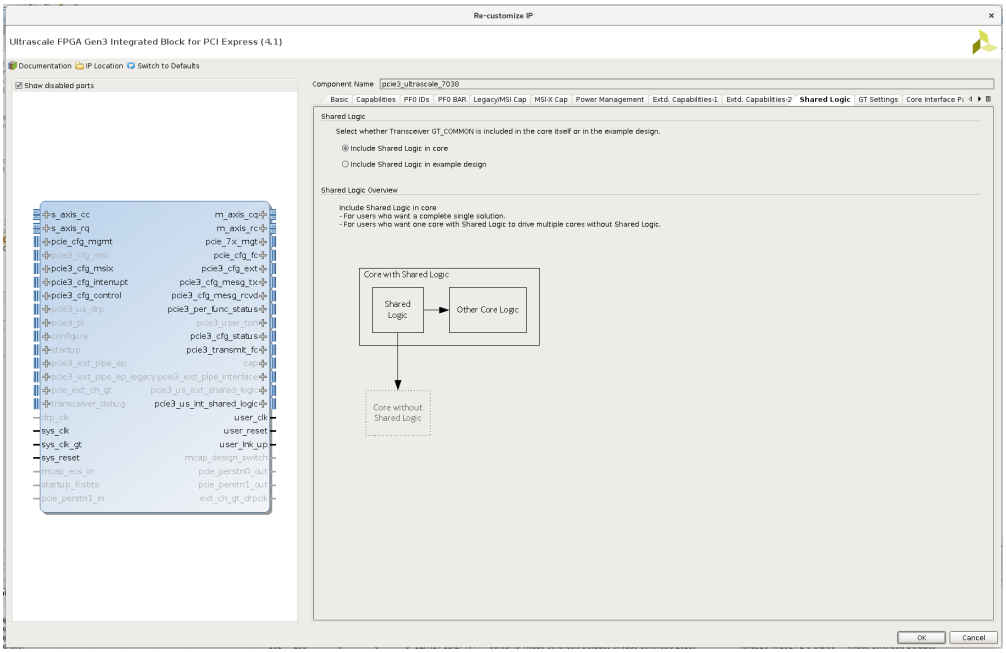


Figure 132: Ultrascale FPGA Gen3 Integrated Block for PXI Express (4.1)

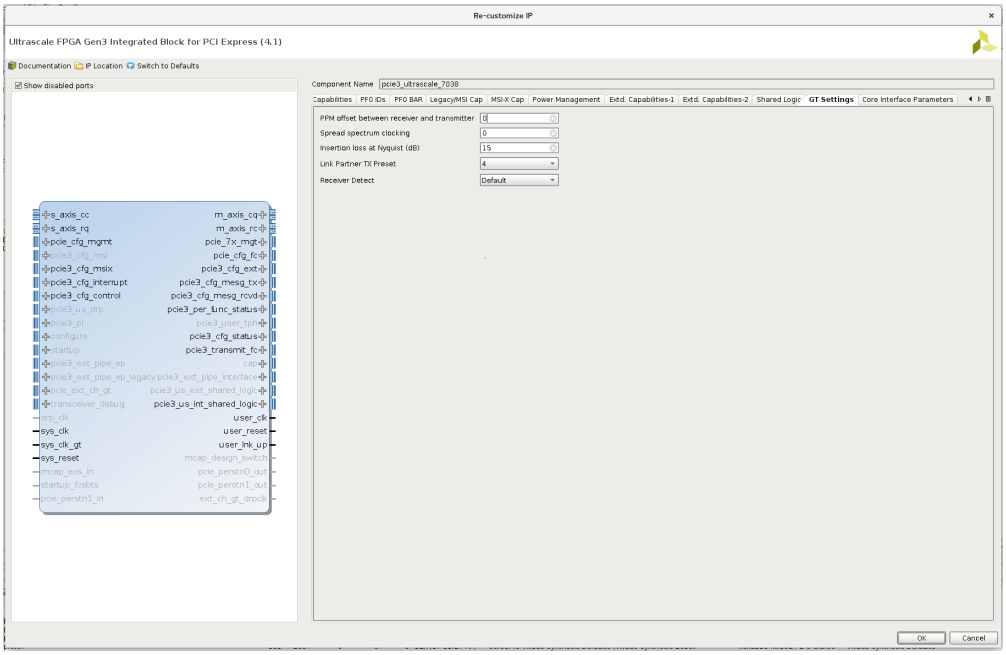


Figure 133: Ultrascale FPGA Gen3 Integrated Block for PXI Express (4.1)

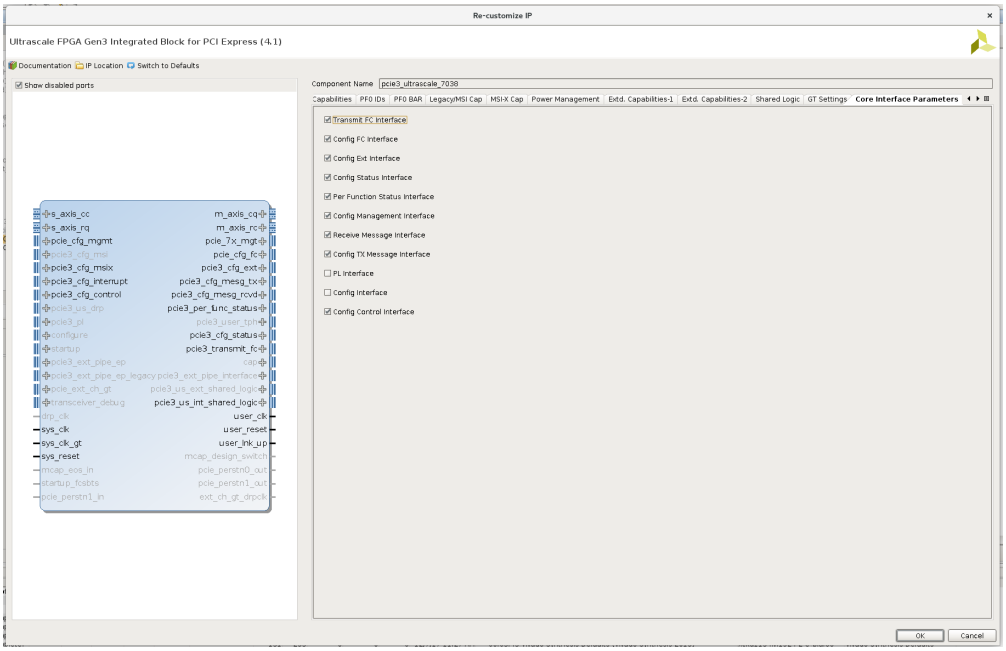


Figure 134: Ultrascale FPGA Gen3 Integrated Block for PXI Express (4.1)

D References

ALPIDE:

1. ALPIDE manual: http://sunba2.ba.infn.it/MOSAIC/ALICE-ITS/Documents/ALPIDE-operations-manual-version-0_3.pdf
2. overview from PRR: https://indico.cern.ch/event/576906/contributions/2367510/attachments/1374383/2086216/ALPIDE-PRR-Chip-Overview.pdf
3. frontend from PRR: https://indico.cern.ch/event/576906/contributions/2376460/attachments/1374411/2093285/ALPIDE_PRR_FrontEnd.pdf
4. Felix Reidt thesis (ALPIDE analog frontend, section 5.5.1, pp 60–62): http://cds.cern.ch/record/2151986/files/
5. Miljenko Suljic thesis (ALPIDE charge collection): https://cds.cern.ch/record/2303618/files/
6. PDG guide on particle interactions with matter (fluctuations in energy loss for thin detectors, section 34.2.9, pp 12–): http://pdg.lbl.gov/2017/reviews/rpp2017-rev-passage-particles-matter.pdf
7. PDG guide on detectors (silicon detectors, section 35.7, pp 48–): http://pdg.lbl.gov/2017/reviews/rpp2017-rev-particle-detectors-accel.pdf
8. subthreshold/weak inversion: https://ieeexplore.ieee.org/abstract/document/5357240/, https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-012-microelectronic-devices-and-circuits-fall-2009/lecture-notes/MIT6_012F09_lec12.pdf
9. ALPIDE frontend paper: http://iopscience.iop.org/article/10.1088/1748-0221/11/02/C02042
10. other papers/presentations:
    (a) https://www.sciencedirect.com/science/article/pii/S0168900215011122
    (b) https://indico.cern.ch/event/666016/contributions/2722251/attachments/1523408/2380925/20170914-ALPIDE-FoCal-Study-Aglieri.pdf
11. ALPIDE carrier boards (we use pALPIDE carrier V3): https://twiki.cern.ch/twiki/bin/viewauth/ALICE/ITS-WP5
    (a) schematic: https://twiki.cern.ch/twiki/pub/ALICE/ITS-WP5/palpide3_carrierv3_sch.pdf
    (b) layout: https://twiki.cern.ch/twiki/pub/ALICE/ITS-WP5/palpide3_carrierv3_pcb.pdf
    (c) Gerber files: https://twiki.cern.ch/twiki/pub/ALICE/ITS-WP5/palpide3_carrierv3_gerber.zip
    (d) wirebonding diagram for ALPIDE on a pALPIDE carrier V3: https://twiki.cern.ch/twiki/pub/ALICE/ITS-WP5/AlpideOnCarrierV3_bonding.pdf

12. adapters for ALPIDE carriers: https://twiki.cern.ch/twiki/bin/view/ALICE/ALPIDE-adaptor-boards
    (a) schematic: https://twiki.cern.ch/twiki/pub/ALICE/ALPIDE-adaptor-boards/ru_alpide3_adaptorv2_sch.pdf
    (b) layout: https://twiki.cern.ch/twiki/pub/ALICE/ALPIDE-adaptor-boards/ru_alpide3_adaptorv2_pcb.pdf
    (c) Gerber files: https://twiki.cern.ch/twiki/pub/ALICE/ALPIDE-adaptor-boards/ru_alpide3_adaptorv2_gerber.zip

RU:

1. overview from PRR: https://indico.cern.ch/event/698929/contributions/2866597/attachments/1624265/2602067/WP10_PRR_RU_Overview_v1.pdf
2. firmware overview from PRR: https://indico.cern.ch/event/698929/contributions/2866596/attachments/1620721/2596687/WP10_PRR_RU_Firmware_v0.pdf
3. auxFPGA from PRR: https://indico.cern.ch/event/698929/contributions/2928340/attachments/1625560/2603137/20180413-PRR_PA3_alme.pdf
4. control interfaces from PRR: https://indico.cern.ch/event/698929/contributions/2866586/attachments/1622853/2603570/WP10_PRR_RU_Interfaces_v2.pdf
5. scrubbing testing from PRR: https://indico.cern.ch/event/698929/contributions/2928334/attachments/1622851/2600928/201804_WP10_PRR_FPGA.pdf
6. TWiki page, RUv1: https://twiki.cern.ch/twiki/bin/view/ALICE/ITS_WP10_RUV1
    (a) RUv1.0 schematic, refdes, manual (zipped): https://twiki.cern.ch/twiki/pub/ALICE/ITS_WP10_RUV1/RUv1.zip
    (b) RUv1.1 schematic, refdes, manual (zipped): https://twiki.cern.ch/twiki/pub/ALICE/ITS_WP10_RUV1/RUv1_1.zip
    (c) power mezzanine schematic: https://twiki.cern.ch/twiki/pub/ALICE/ITS_WP10_RUV1/powerMezzanine.pdf
    (d) ALICE transition board schematic: https://twiki.cern.ch/twiki/pub/ALICE/ITS_WP10_RUV1/transitionboard_full.pdf
7. TWiki page, RUv2: https://twiki.cern.ch/twiki/bin/view/ALICE/ITS_WP10_RUV2
    (a) schematic: https://twiki.cern.ch/twiki/pub/ALICE/ITS_WP10_RUV2/RUv2schematic.pdf
8. TMR:
    (a) TMR techniques from NASA (referenced by auxFPGA slides): https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20170004736.pdf
    (b) Johan Alme thesis (predecessor design for scrubbing FPGA): http://cds.cern.ch/record/1141616/files/

(c) BYU thesis (scrubbing overview): https://scholarsarchive.byu.edu/cgi/viewcontent.cgi?referer=&httpsredir=1&article=6703&context=etd

(d) Xilinx XAPP197 (TMR techniques): https://www.xilinx.com/support/documentation/application_notes/xapp197.pdf

9. GBTx manual: https://espace.cern.ch/GBT-Project/GBTX/Manuals/gbtxManual.pdf
10. GBT-SCA overview paper: http://cds.cern.ch/record/2158969/files/
11. VTRx spec: https://espace.cern.ch/GBT-Project/VLDB/Manuals/VTRx_Spec_v2.1.pdf
12. Xilinx docs: Kintex Ultrascale, GT (UG576), selectIO (UG571)
13. Si5316 jitter cleaner data sheet: https://www.silabs.com/documents/public/data-sheets/Si5316.pdf
14. auxFPGA web page: https://twiki.cern.ch/twiki/bin/viewauth/ALICE/AuxFPGA

FELIX:

1. FELIX web page: https://atlas-project-felix.web.cern.ch/atlas-project-felix/
2. FELIX documents: https://atlas-project-felix.web.cern.ch/atlas-project-felix/docs/, https://atlas-project-felix.web.cern.ch/atlas-project-felix/user/documentation.html

   (a) FELIX user manual: https://atlas-project-felix.web.cern.ch/atlas-project-felix/user/docs/FelixUserManual.pdf, https://gitlab.cern.ch/atlas-tdaq-felix/documents/blob/master/UserManual/FelixUserManual.pdf

   (b) FELIX hardware manual (v2.0): https://atlas-project-felix.web.cern.ch/atlas-project-felix/user/docs/BNL-711_V2P0_manual.pdf

   (c) FELIX tech spec: https://atlas-project-felix.web.cern.ch/atlas-project-felix/user/docs/FELIX_TechSpecs_DR03-2016.pdf

   (d) Wupper manual: https://atlas-project-felix.web.cern.ch/atlas-project-felix/user/docs/wupper.pdf, https://gitlab.cern.ch/atlas-tdaq-felix/documents/blob/master/Wupper/wupper.pdf

3. Timing mezzanine schematic: https://gitlab.cern.ch/atlas-tdaq-felix/hardware/tree/master/BNL-KMC
4. FELIX schematics: https://gitlab.cern.ch/atlas-tdaq-felix/hardware/tree/master/BNL-711
5. GBT-FPGA website: https://espace.cern.ch/GBT-Project/GBT-FPGA/default.aspx
6. GBT wrapper manual (Kai Chen's fork of GBT-FPGA): https://atlas-project-felix.web.cern.ch/atlas-project-felix/user/docs/FELIX_GBT_MANUAL_V2.pdf
7. Xilinx PG156 (PCIe endpoint)

8.

AXI documents, API docs, Linux references, Xilinx docs (DMA, PCIe)

[1] B Abelev et al. *Technical Design Report for the Upgrade of the ALICE Inner Tracking System*. Tech. rep. CERN-LHCC-2013-024. ALICE-TDR-017. Nov. 2013. URL: https://cds.cern.ch/record/1625842.

[2] ALPIDE development team ALICE ITS. "ALPIDE Operations Manual". 2016.