FPGA Design Upgrades for the ATLAS Pixel Readout System in the Large Hadron Collider

Sanjukta Roychoudhury

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Electrical Engineering

University of Washington

2023

Committee:

Scott Hauck

Shih-Chieh Hsu

Program Authorized to Offer Degree:

Department of Electrical and Computer Engineering

University of Washington

Abstract

FPGA Design Upgrades for the ATLAS Pixel Readout System in the Large Hadron Collider

Sanjukta Roychoudhury

Chair of the Supervisory Committee:
Professor Scott Hauck
Electrical Engineering

The ATLAS Pixel Detector in the Large Hadron Collider uses a system of FPGAs in the off-detector readout system. The Read Out Driver (ROD) is a major component of the system that handles processing of front end data and sending it to the next stages of filtering and storage. This thesis starts with a background on the detector and readout system and moves on to discuss efforts in support and development of ROD firmware. The first project investigated anomalous behavior in the ROD Histogrammer. This was understood and resolved after testing in the SR1 facility, which has a mockup of the detector hardware. The second project, called Smart L1A Forwarding, was restarted in 2022 with an objective to mitigate desynchronization of data in Pixel during the challenging conditions of Run-3. Initial development and tests in SR1 and the detector have shown promising results. Further modifications have been made in an attempt to resolve issues found during testing; these modifications are being tested in the detector.

CONTENTS

CHAPTER 1: INTRODUCTION AND BACKGROUND

Projects related to firmware development for the FPGAs used in the ATLAS Pixel detector in the LHC at CERN have been described in this thesis. This section provides an introduction and background on relevant systems, starting at CERN and the LHC, then zooming in to the Pixel detector and the FPGA based readout system.

1.1 CERN

The European Organization for Nuclear Research (CERN) is an international organization that runs the largest particle physics laboratory in the world [1]. The main focus at CERN is studying fundamental particles with various particle accelerators and detectors at the facility. CERN’s headquarters is located on the French-Swiss border near Geneva, and the work there is done through collaboration with many institutions across the world. Work at CERN has led to breakthroughs in science and technology such as the discovery of the Higgs Boson and the invention of the World Wide Web.

1.2 Large Hadron Collider

The Large Hadron Collider (LHC), the world's largest and most powerful particle accelerator, is located at CERN. The LHC consists of a 27 km vacuum ring of superconducting magnets [2]. The tunnel is about 100m underground. Particles are first accelerated in smaller accelerators before being sent into the LHC, which is the final stage. Two high energy particle beams are accelerated to close to the speed of light and made to collide at four interaction points in the LHC [2]. These particle beams collide about every 25 ns. Every collision is called a bunch crossing (BC). BC is often used as a unit of measurement (25 ns). Data resulting from each bunch crossing is referred to as an event.

There are four major particle detector experiments that study different aspects of particle physics; these detectors are each installed at an interaction point where the beams collide and have been shown in Figure 1.1. The four experiments are:

- Compact Muon Solenoid (CMS)
- A Large Ion Collider Experiment (ALICE)
- Large Hadron collider beauty (LHCb)
- A Toroidal LHC Apparatus (ATLAS)

There are five more experiments located at or near the major experiments and interaction points - TOTal cross section, Elastic scattering and diffraction dissociation Measurement at the LHC

(TOTEM); LHC-forward (LHCf); Monopole and Exotics Detector At the LHC (MoEDAL); Scattering and Neutrino Detector (SND@LHC); and ForwArd Search ExpeRiment (FASER) [2].

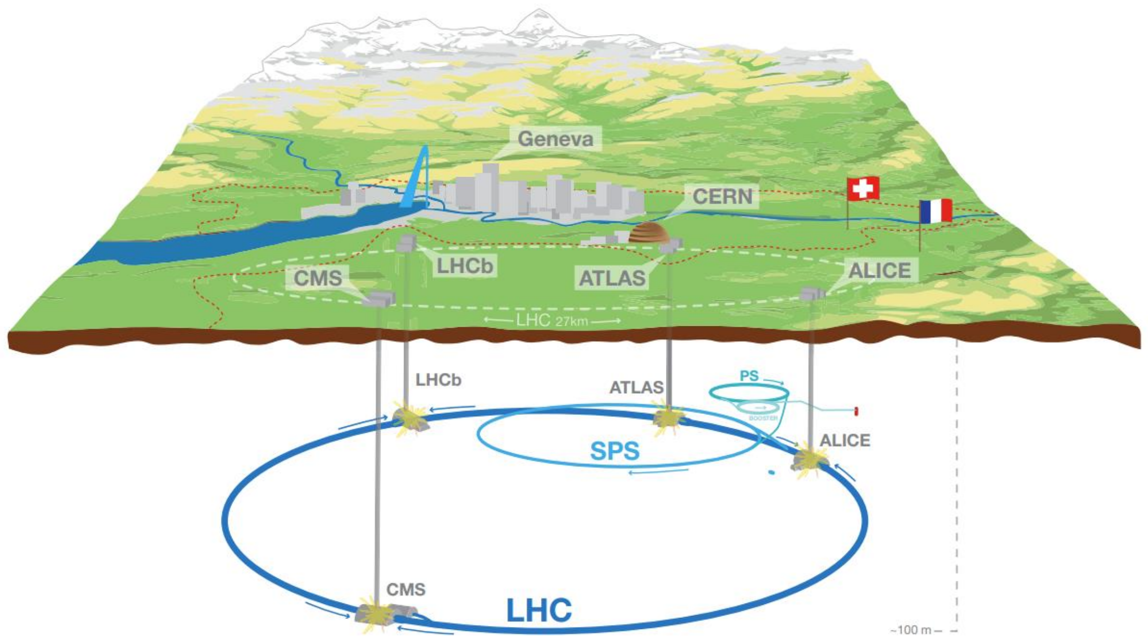


Figure 1.1: Location of the LHC and its major experiments [3].

1.3 ATLAS

ATLAS (A Toroidal LHC ApparatuS) is a general purpose particle detector that is capable of a wide range of physics, from precise measurement of the Higgs Boson to searching for physics beyond the Standard Model [4].

It is the largest experiment in the LHC, and is located in a cavern 100 m below the ground in Meyrin, Switzerland close to the main CERN site. It is 46 m long, 25 m in diameter and weighs about 7,000 tonnes. After filtering, it produces data at the rate of about 1.5 GB/s during proton-proton collisions [5].

The detector is made up of different types of sub-detectors arranged in a layered fashion around the collision point, that are meant to record path, momentum and energy of specific types of particles. The detector can be divided into four major systems (as shown in Figure 1.2):

- Inner Detector
- Calorimeters
- Muon Spectrometer
- Magnet system

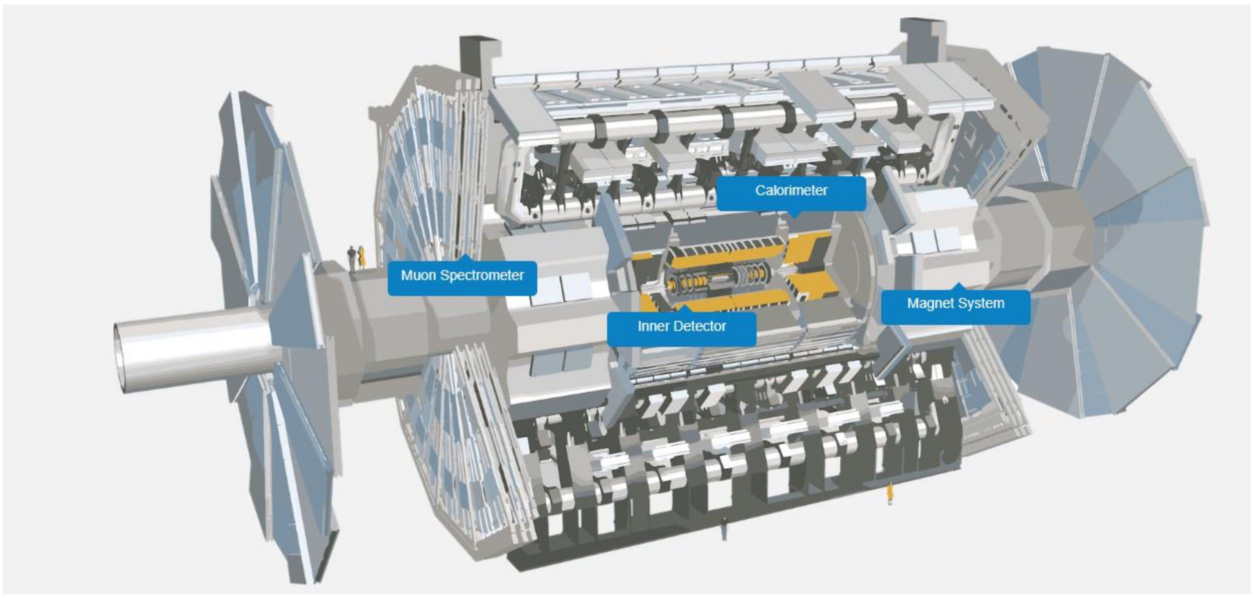


Figure 1.2: ATLAS Detector with its major components [6].

Each of these are made of multiple layers within them. The Inner Detector is the first part of ATLAS to see collisions, and measures the direction, momentum and charge of electrically charged particles produced in collisions. Calorimeters measure the energy of particles as they interact with matter and are designed to absorb particles within the detector. The muon system takes measurements related to muons that pass through the Inner Detector and Calorimeter undetected. The magnet systems bend particles so that their charge and momenta can be measured [6].

1.4 ATLAS Trigger System

ATLAS looks at millions of collisions per second, but only some of these events are likely to contain interesting data that could lead to new discoveries. ATLAS uses an event selection system called Trigger which picks events with interesting characteristics for further analysis [7]. The ATLAS Trigger uses a 2-level architecture to filter out and keep desired data.

Level-1 (L1) triggers, or Level-1 Accept (L1A), are the lowest level of triggers. These are implemented completely in hardware using FPGAs and have the highest rates of acceptance. The rate of acceptance of an event to be captured is called trigger rate. The trigger rate can be adjusted according to the requirements of the run. This adjustment comes from the tuning of acceptance criteria to increase or decrease the chance of accepting an event. These decisions are made by the Central Trigger Processor (CTP) using information from the Calorimeter and Muon Spectrometer and are sent to the Timing Trigger and Control system (TTC).

Level-1 triggers are distributed to off-detector readout electronics by the TTC Interface Module (TIM). TIM also distributes a global 40MHz clock to the detector electronics. For every trigger, unique identifiers like L1ID (Level-1 Accept ID), BCID (Bunch Crossing ID), ECRID (Event Counter Reset ID) are sent by the TIM to the Readout Drivers for data synchronization checks [8].

The data obtained after Level-1 triggers is further filtered out by High Level Triggers (HLT). This decision making is done by software on PC farms. These higher levels of trigger have more complex triggering criteria and lower trigger rates compared to Level-1 triggers [9].

1.5 Pixel Detector

The Inner Detector consists of three sub-detectors - Pixel Detector, Semiconductor Tracker (SCT) and Transition Radiation Tracker (TRT). In this thesis, the focus is on the Pixel sub-detector which is the innermost detector of the Inner Detector and hence the innermost in ATLAS. The location of the Pixel Detector in the Inner Detector has been shown in Figure 1.3.

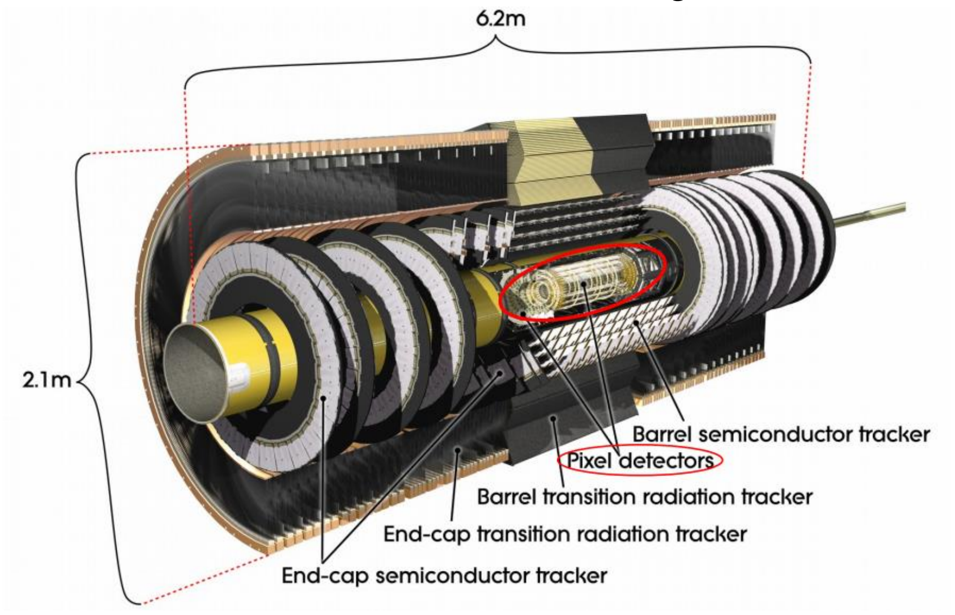

6.2m
2.1m
Barrel semiconductor tracker
Pixel detectors
Barrel transition radiation tracker
End-cap transition radiation tracker
End-cap semiconductor tracker

Figure 1.3: ATLAS Inner Detector [10].

Within the Pixel detector there is a layered structure consisting of four Barrel Layers - Layer-0 (or B-Layer), Layer-1, Layer-2, and Insertable B Layer (IBL - which was installed in 2014), as well as six end-cap disks: Disk-1, Disk-2 and Disk-3 on both sides. All of the layers except the IBL are the original layers, often referred to as the Pixel Layers. These layers have been shown in Figure 1.4.

There are over 90 million silicon-based particle detectors (silicon Pixel sensors) distributed across these layers. These sensors are attached to Front End chips (FEs) that can process sensor signals, which are then connected to the off-detector readout system. When a pixel is hit by a charged particle, it deposits charge on the analog part of the sensor, which is amplified and captured by the digital part of the sensor [10].

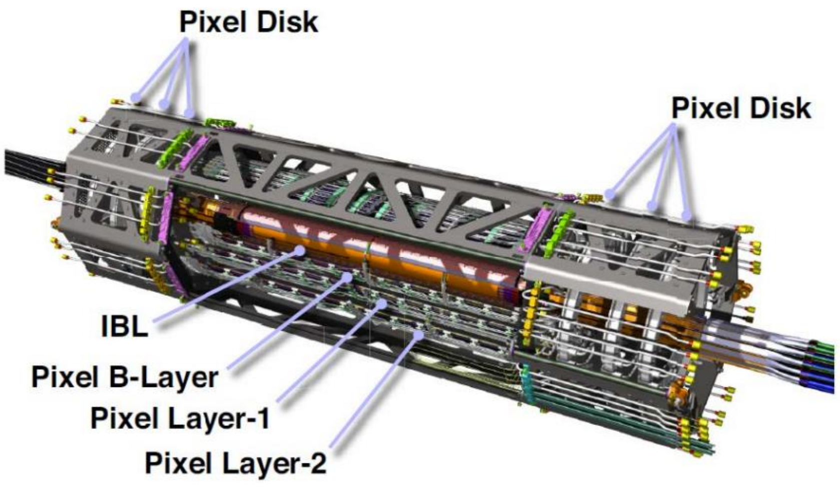


Figure 1.4: Layers in the Pixel detector [11]

1.6 Pixel Readout Chain

The readout chain in the Pixel detector sends data from sensors in response to Level-1 triggers (see section 1.4). The readout chain consists of on-detector and off-detector components. In Pixel Layers, the front end chips (FEI3) are connected to an additional component called the Module Control Chip (MCC) to make up a module. An optoboard (OB) receives electronic data from the modules and converts it to optical data which is sent to the off-detector readout system through an optical link. The module and optoboard are on-detector electronics and are hence more susceptible to the radiation caused by the particle beam. These components are shown in the on-detector portion of Figure 1.5.

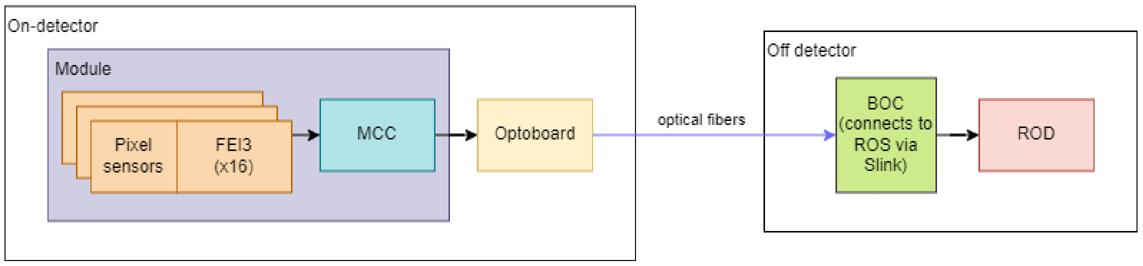


Figure 1.5: Simplified block diagram of the Pixel readout chain from one module to ROD.

The Read Out Driver (ROD) and Back of Crate (BOC) are the main components of the VME crate based off-detector readout system, and are made up of a collection of FPGAs. The TIM is also interfaced with this system to distribute triggers. Since these are located away from the detector in a separate counting room, which is underground and shielded from the collision site, they are safe from radiation. The ROD sends commands for the modules through the BOC. The BOC gets data serially from modules through the optical link and sends this data to the ROD. The ROD further processes data to send and sends it back to the BOC that can forward it to the ReadOut System (ROS) through the S-Link interface [12] for the next stages of filtering and storage. The organization of these components have been shown in Figure 1.6 for IBL.

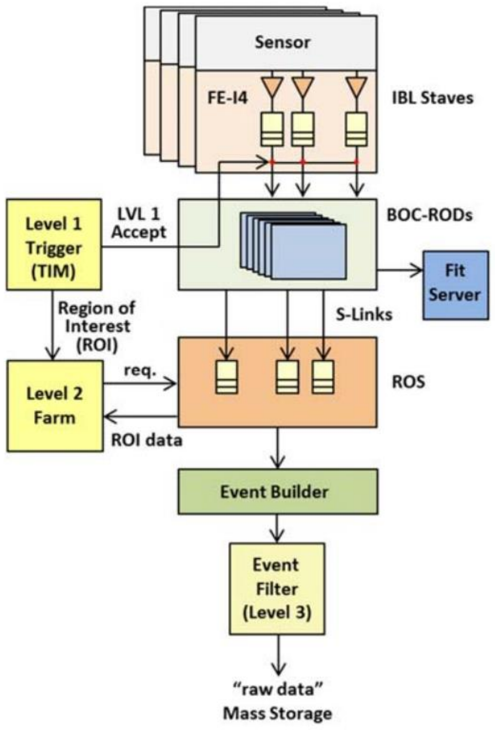


Figure 1.6: IBL section of ATLAS TDAQ [8].

An important thing to note is that the readout system is slightly different for the IBL and Pixel layers. Initially, when there were just the original three Pixel layers, a different readout called SiROD (Silicon Read Out Driver) was used [13]. When the IBL was inserted, a completely new readout system was developed. This is known as IBL ROD, shown in Figure 1.7. Later, the Pixel readout was upgraded to be similar to the IBL ROD. It has the same general structure and datapath but has a few changes in the firmware because of differences in the front end chips and modules. Therefore currently, the hardware (FPGAs and boards) for IBL and Pixel layers are the same but the firmware is slightly different.

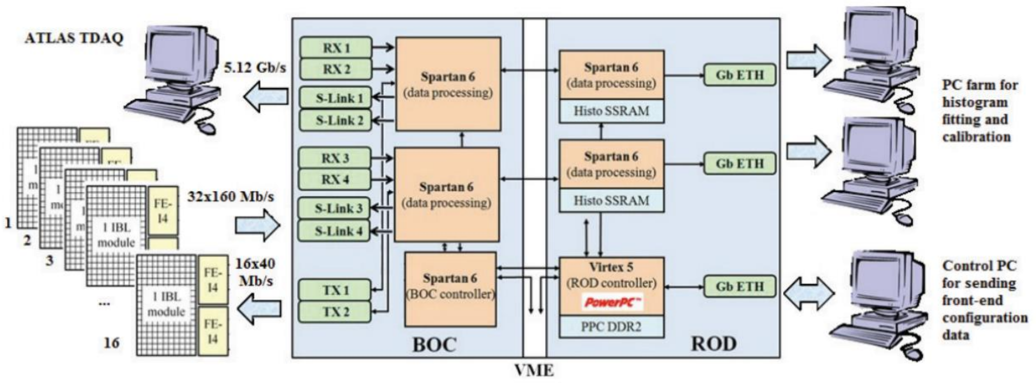

ATLAS TDAQ
5.12 Gb/s
32x160 Mb/s
16x40 Mb/s
1 IBL module
FE-I4
1
2
3
...
16
RX 1
RX 2
S-Link 1
S-Link 2
RX 3
RX 4
S-Link 3
S-Link 4
TX 1
TX 2
Spartan 6 (data processing)
Spartan 6 (data processing)
Spartan 6 (BOC controller)
BOC
VME
Spartan 6 (data processing)
Histo SSRAM
Spartan 6 (data processing)
Histo SSRAM
Virtex 5 (ROD controller)
PowerPC
PPC DDR2
ROD
Gb ETH
Gb ETH
Gb ETH
PC farm for histogram fitting and calibration
Control PC for sending front-end configuration data

Figure 1.7: IBL readout system [14]

The front ends, modules, BOC and ROD are further described below.

FEI4, FEI3 and MCC

FEI3 [15] is the front end chip for Pixel and FEI4 [16] is the front end chip for IBL. The front end chips are bump bonded to the Pixel sensors to detect particles hitting the sensor. The Front end chip has circuitry that computes the Time over Threshold (ToT), which represents the number of clock cycles (bunch crossings) that the signal was high compared to a configurable threshold. The occurrence of the signal for longer than the threshold from a particle hitting the sensor is known as a hit, and each hit has a ToT associated with it. Each FEI3 typically has a grid of 18x160 pixels, 2,880 in total, that are 50x400 μm^2 each. Each FEI4 typically has 80x336 pixels, 26,880 in total, that are 50x250 μm^2 each.

A Module Control Chip [17] is shared by 16 FEI3 chips to make up a module (see Figure 1.5). The MCC is responsible for arbitration of FE data to be sent to the link. It also handles error checking for bitflips and keeps a count of skipped triggers when the FEs cannot keep up with incoming triggers. In IBL, two FEI4 chips make up a module (see Figure 1.7), there is no MCC, and that functionality is handled by FEI4. The readout speed in IBL is 160 Mb/s. In Pixel it is 160 Mb/ for B-Layer, Layer-1 and Disk-2, and 80 Mb/s in the other layers.

BOC

The Back of Crate (BOC) board is the opto-electrical conversion board [18] that is an interface between modules and the ROD. It consists of three Xilinx Spartan-6 FPGAs: one master BOC Control FPGA (BCF) and two BOC Master FPGAs (BMF) that are slave data links. Modules are connected to the BOC via optical fibers. The BOC sends and receives information between the ROD and modules, including commands and hit data. The BOC also has an S-Link interface that sends data to the ROS.

ROD

The Read Out Driver (ROD) is responsible for event building and calibration [19]. It consists of four Xilinx FPGAs:

- Master: Virtex-5 FX70T with PowerPC (PPC) processor
- Two slaves: Spartan-6 LX150 with MicroBlaze (MB) processor
- Program Reset Manager (PRM): Spartan-6 LX45, the low-level board controller

The ROD is mainly responsible for transmitting control commands and configuration data to the front ends through the BOC and receiving data streams from the front ends through the BOC after decoding. It then detects and marks format errors, builds events and sends the events to the ROS through the BOC. It also handles histogramming during calibration mode.

The ROD Master gets trigger information from the TIM and generates commands for the front ends. It also builds event and synchronization information that is sent to the ROD Slave.

The ROD Slave handles data formatting, event building and histogramming. This datapath will be described in a bit more detail since the projects in the following chapters are related to the ROD Slave. In the Slave FPGA, the Formatter, Event Fragment Builder (EFB), Router and Histogrammer are arranged into two half slaves, with each half slave consisting of two Formatters, one EFB, one Router and one Histogrammer as shown in Figure 1.8. Each ROD Slave typically connects to 6 or 7 MCCs, if it is for Pixel Layers, or 16 FEI4s (8 modules consisting of 2 FEI4s each) for IBL. The descriptions here are based on the IBL-ROD manual [19], which also has more detailed firmware implementation information.

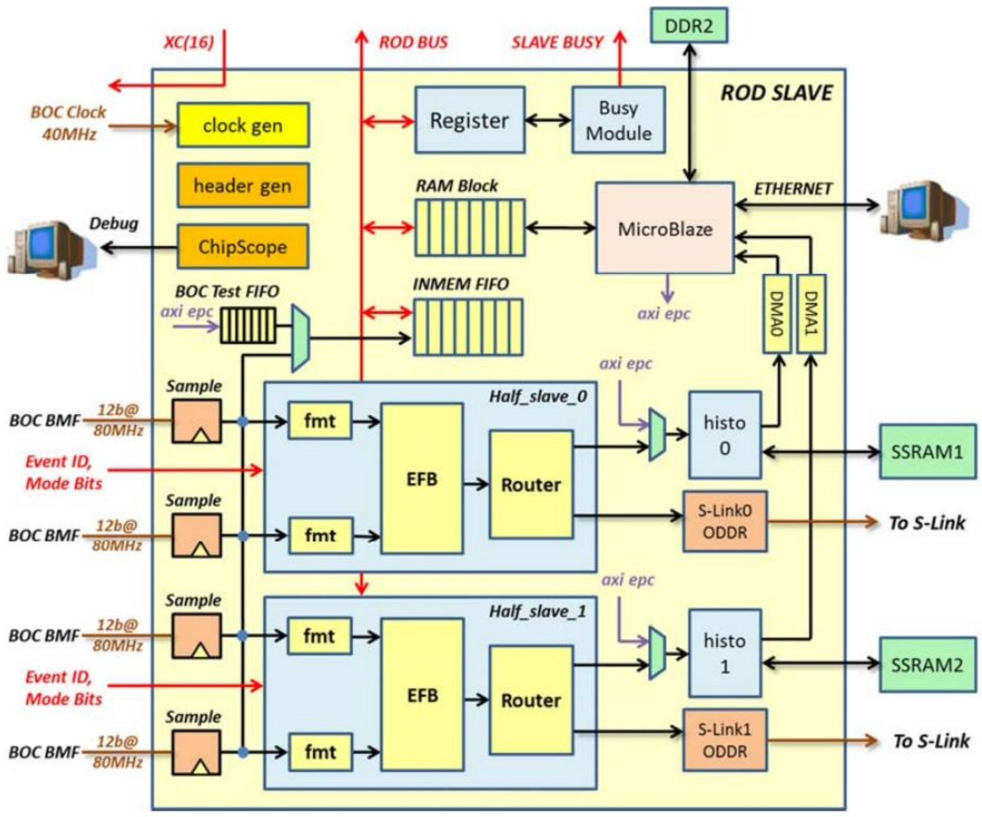


Figure 1.8: ROD Slave firmware block diagram [8]

Formatter

The Formatter is the first stage in the ROD slave datapath. It gets module data from the BOC. Each Formatter mainly consists of four link encoders (or a quad link encoder in Pixel) and a readout controller. The module data is sent to link encoders depending on the link enabled. The link encoder first decodes the data from the BOC and breaks it into header, hits, errors and trailer. The readout controller is a block in the formatter that organizes the readout of data in the enabled links to the EFB and also handles a few error scenarios where it inserts empty events if the FE does not respond. Links are enabled by a register in the ROD Slave that controls which links are active according to the configuration of the run. The formatter is also responsible for buffering module data in FIFOs and sending a busy signal to CTP (part of the trigger system) if FIFOs are full. Formatter interfaces and schematic have been shown in Figures 1.9-1.10.

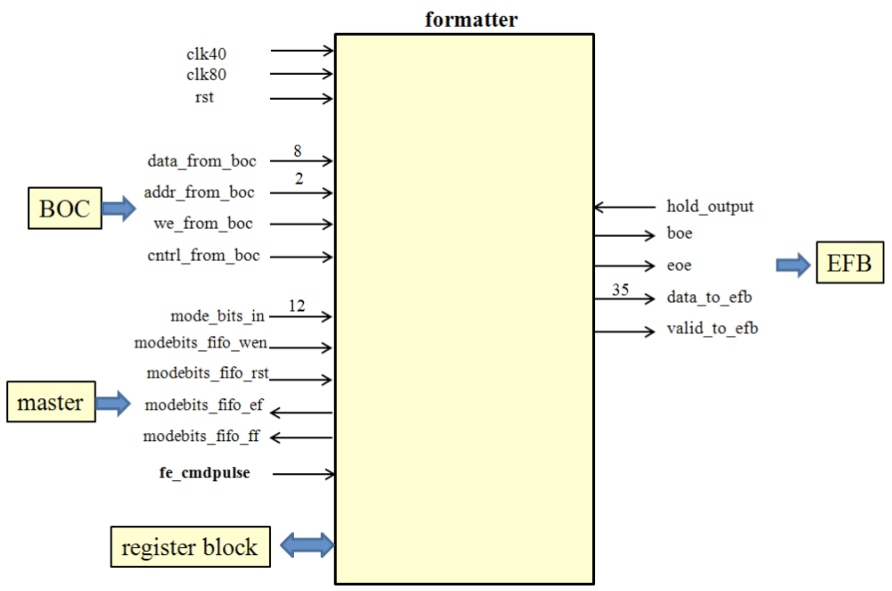


Figure 1.9: Interfaces of the Formatter [19]

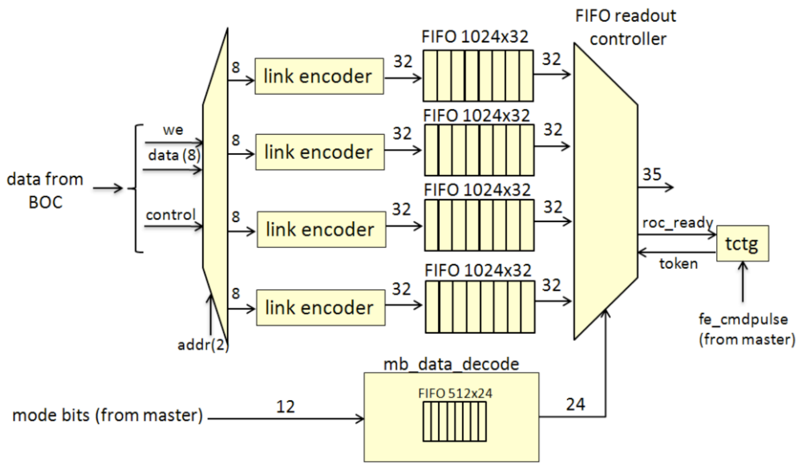


Figure 1.10: Schematic of the Formatter [19]

EFB

The Event Fragment Builder handles data checking and event fragment generation. It receives data from two Formatters (up to 8 links) and other event information from the master. It does error checking, flagging, and organizes this data into event fragments to be sent through S-Link, which has a specific format. There is an S-Link header and trailer that encapsulates all module data associated with an event. The interfaces of the EFB have been shown in Figure 1.11.

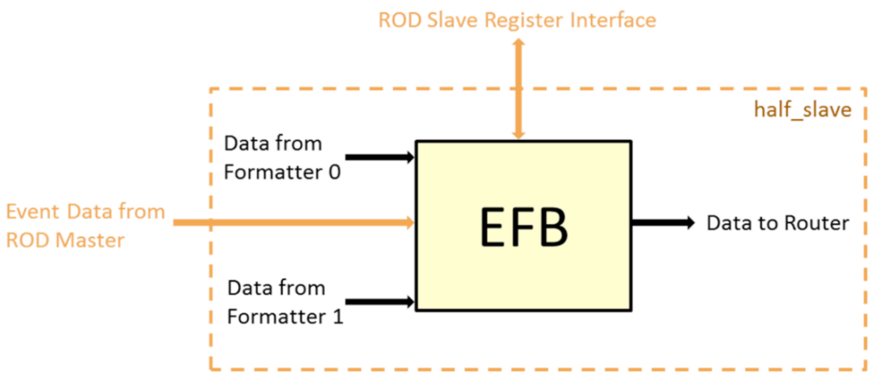


Figure 1.11: Interfaces of the EFB [19]

Router

The Router gets all events processed by the EFB and either forwards them to the ROS through S-Link (via BOC) or extracts hits for the Histogrammer. Hit data consists of row, column, chipID and ToT. Hits are routed only to the Histogrammer during calibration and to the S-Link during datataking runs. The interfaces of the Router have been shown in Figure 1.12.

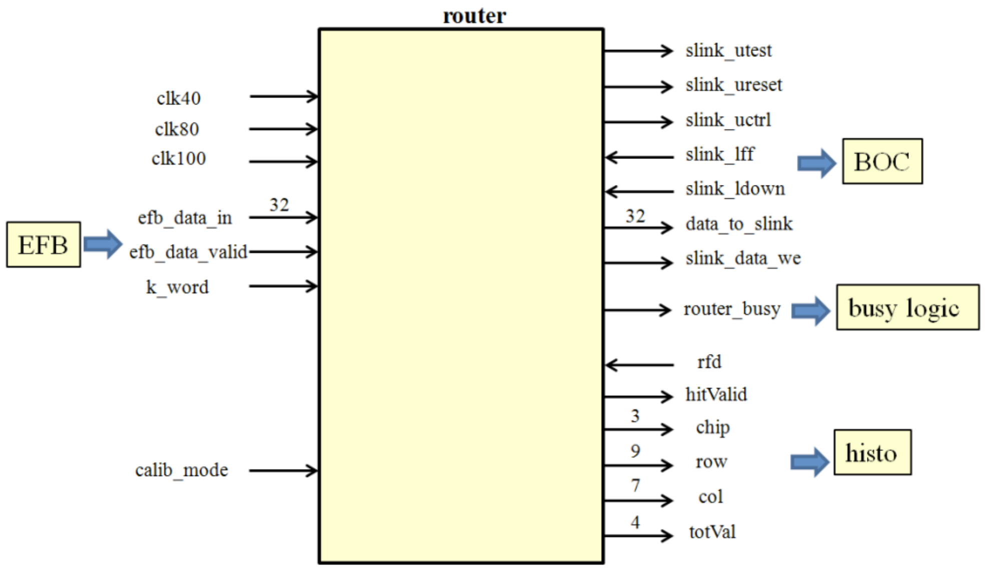


Figure 1.12: Interfaces of the Router [19]

Histogrammer

The Histogrammer block gets extracted hits from the Router and accumulates the occupancy (number of hits in a pixel), the ToT and ToT^2 values for histograms used in detector calibration and also online monitoring during datataking. The interfaces of the histogram have been shown in Figure 1.13.

Computed histograms are stored in an external SSRAM. The chip number, row and column numbers of a pixel hit are translated into an address for the SSRAM. When there is a hit, the associated memory address is read, the hit, ToT and ToT^2 are added, and written back to that address in their respective fields as shown in Figure 1.14. When the histogram is complete, the MicroBlaze controls the transfer of the results to an external DDR2RAM through DMA. This data is then transferred from the DDR2RAM to a PC farm called FitServer via ethernet for further processing. Each SSRAM word is 36 bits wide, and each DDR2RAM word is 32 bits wide. There are different readout schemes to handle this difference in word width based on the calibration test or scan being done. More details can be found in [19]. The readout schemes are:

- LONG_TOT: Reads out the complete 36 bits/pixel, generating two data words per pixel.
- SHORT_TOT: Computes the difference between expected hits (programmed) and actual hits that a pixel receives and packs this value together with $\sum$ToT and $\sum$ToT^2 into a 32-bit word.
- ONLINE_OCCUPANCY: Samples and reads out 24-bit wide occupancy values only. This is used during data-taking for online monitoring if histogramming is enabled.
- OFFLINE_OCCUPANCY: Combines the 8-bit occupancy values of 4 pixels into one 32-bit word.

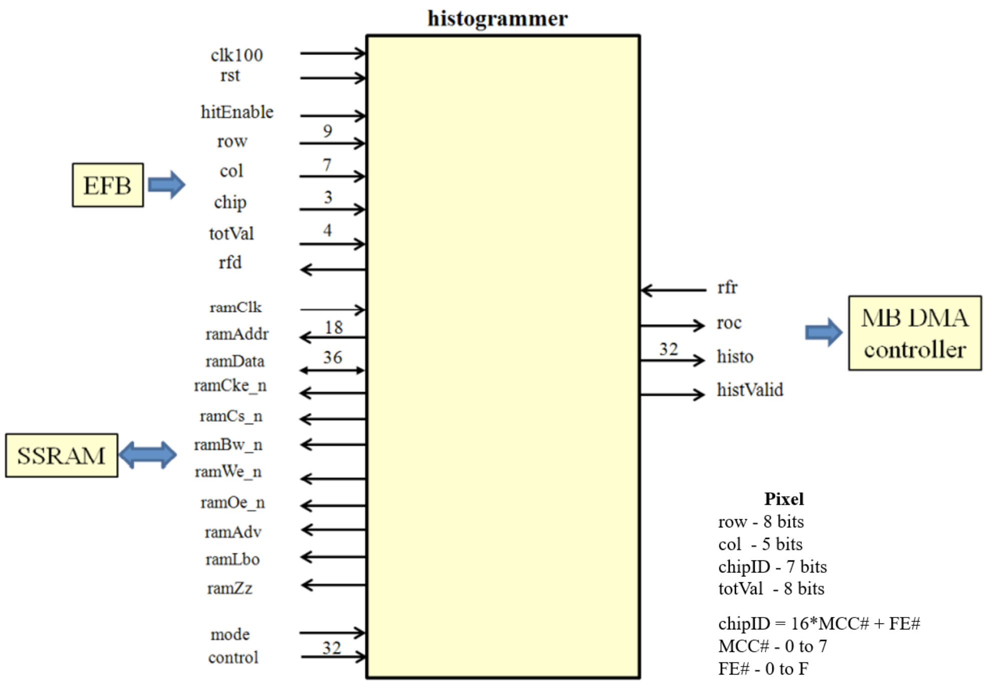


Figure 1.13: Interfaces of the Histogrammer [19]

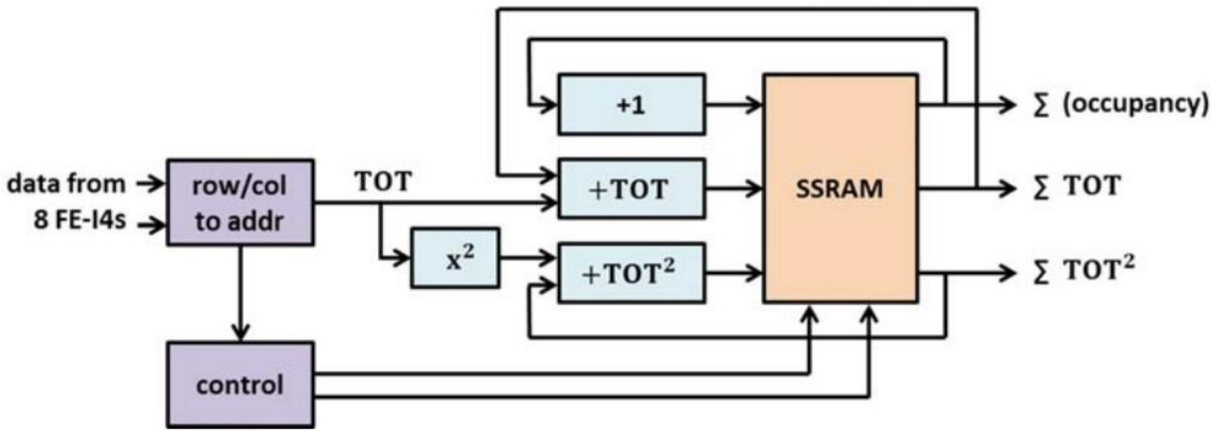


Figure 1.14: Implementation of the ROD Histogrammer [14]

Slave Register Block

The ROD Slave register block consists of many important control and monitoring registers that can be accessed by the ROD Master if configured to do so. Some registers of interest that are available here are the fei3_mode (configuration register), fmt_link enable (links enabled in formatter) and monitoring registers that have the count of total events or other specific types of events or errors. All register address information can be found in rodSlaveRegPack, rodHistoPackPix, rodHistoPack files in the firmware.

1.7 Context on ROD Firmware

Datataking and calibration are the two different modes in which the detector and ROD Slave in particular can operate.

- Datataking mode is when the detector is actively collecting data. Sensor data goes through the entire ROD slave datapath and is sent to S-Link through the BOC.

- Calibration mode is used for calibrating the system to ensure things are running correctly. There are various scans that are done for calibration. Digital and Analog scans are some of the most commonly done [20]. Digital Scan tests the functionality of the digital part of the readout chain for each pixel. Analog Scan tests the analog part of each pixel in addition to the digital part.

These modes have implications for firmware. The legacy IBL ROD Slave firmware could support both datataking and calibration. However, it was using up most of the resources in the FPGA which hindered the addition of new features in the firmware. Therefore, the ROD Slave firmware was split into datataking and calibration versions. The main difference is that the calibration version has the MicroBlaze CPU which is needed for histogramming but datataking does not. This freed up space for the addition of new features in datataking. Therefore, considering Pixel, IBL, dataking and calibration, there are four versions of ROD Slave firmware being used.

All of the official documentation for the present-day ROD-BOC system is in the context of IBL, although the same system is now used in Pixel. The changes in the firmware arise because of minor data format differences, and module arrangement, but the general flow of data is the same.

1.8 SR1

SR1 is a testing facility at ground level at CERN that houses replicas of the hardware present in the ATLAS detector, which is underground. It consists of a replica of a vertical slice of the detector. The SR1 lab enables developers to access the hardware and carry out testing at any time, regardless of whether the actual detector is actually in operation. It also allows for testing of new features before deployment in the real detector, where hardware is inaccessible and difficult to replace in case of any unexpected issues. The SR1 setup can be accessed remotely so tests can be run without physically being at CERN. This offers flexibility for collaborators working on ATLAS across the world. There are both Pixel and IBL style modules available. These are arranged in crates replicating different layers in the detector. Each crate has a few slots populated with readout boards, with each slot consisting of a ROD-BOC system. Testing can also be done using the module emulator in the BOC, which does not require switching on modules. The SR1 setup has been extensively used in the development and testing of ROD Slave firmware described in this thesis. Appendix A has more information on usage of SR1.

CHAPTER 2: ROD HISTOGRAMMER INVESTIGATION

2.1 Motivation

Histograms are used to show the number of hits (also called occupancy) in each Pixel in each front end in a module. The x-axis corresponds to the column of the front end and the y-axis represents the row. The number of hits in each pixel is represented by the color of that pixel. No color indicates that there are no hits in that pixel. For Pixel layers, each histogram shows the occupancy for 16 FEI3s in a module, and for IBL it shows occupancy for 2 FEI4s in each module as shown in Figure 2.1. The Histogrammer block in the ROD slave firmware plays an important role in the generation of these histograms as described in Section 1.6.

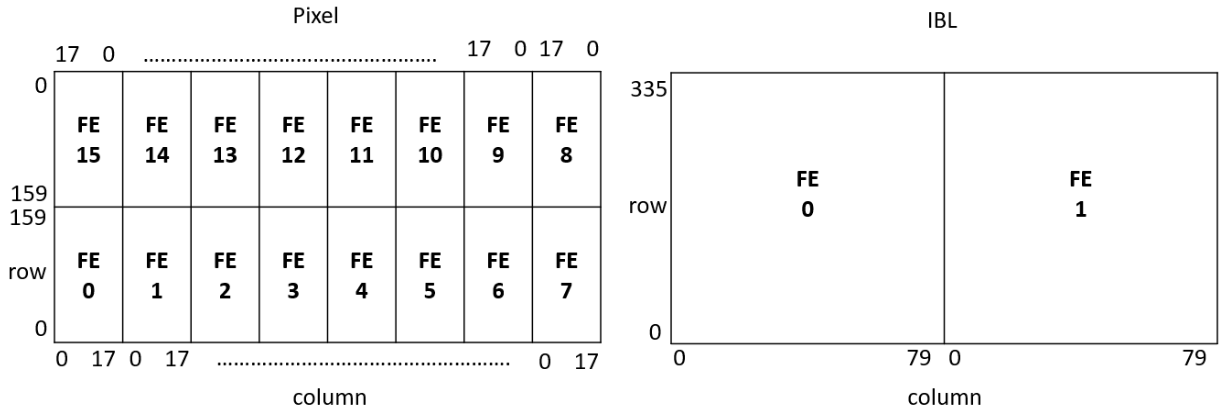


Figure 2.1: Histogram structure for Pixel and IBL showing mapping of front end pixels on the histogram

Configuring the detector to collect data optimally involves the generation of a noise mask. This mask is used to mask off pixels that are too noisy and hence may either be faulty or mistuned. This mask is generated by enabling histogramming and running the detector in datataking mode (without beam) for a certain number of events (usually ~10 million). Then at the end of the run, the online occupancy histograms are analyzed to mask pixels that have greater than or equal to 10 hits, meaning that the occupancy per event is greater than 10^6 (10 hits/10 million events). It should be noted that this process requires the running of calibration firmware in datataking mode, since the MicroBlaze processor is required for histogramming [21].

It was observed that there were erroneous hits in the noise mask histograms for multiple modules in the detector as seen in Figures 2.2 and 2.3. This was seen both in Pixel and IBL. In Figure 2.2, which is from a Pixel layer, there is a yellow block at the bottom left that has more hits than the rest of the histogram. In Figure 2.3, which is from IBL, the block on the left half that is blue, and a vertical green and a horizontal blue line have more hits than the rest of the histogram. There seemed to be a specific pattern to these hits, and it seemed unnatural that so many pixels spatially close together in almost regular shapes would be noisy in different modules. This would result in the noise mask unnecessarily masking off many pixels if not understood and resolved.

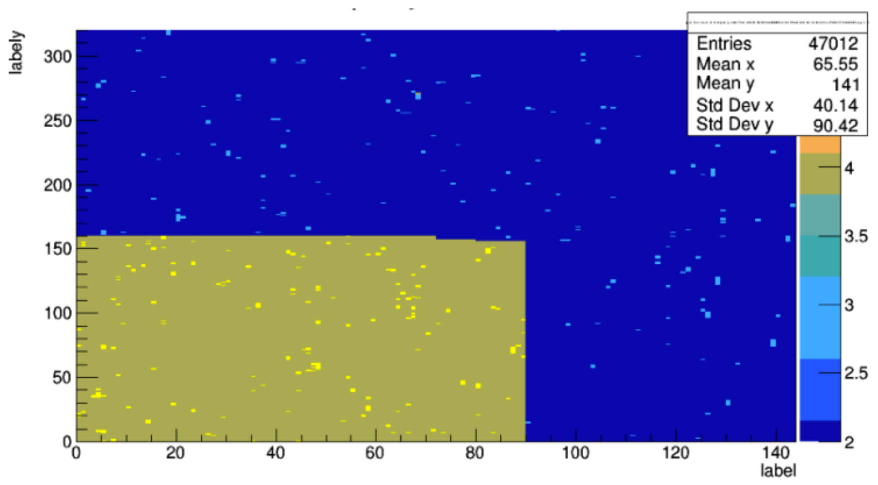


Figure 2.2: Histogram from Pixel Run 404598 [21].

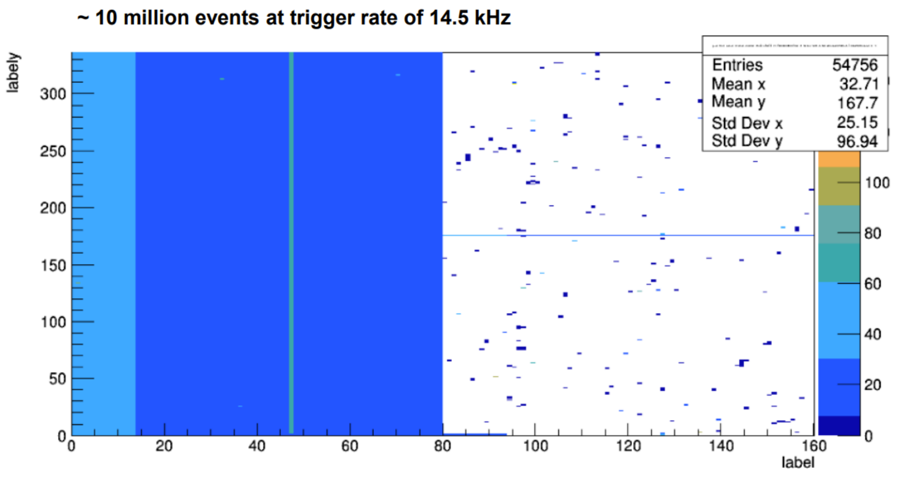


Figure 2.3: Histogram from IBL Run 400850 [21].

2.2 Initial investigation and tests

Investigation of this issue was done on the SR1 setup. The first step was to become familiar with operating SR1 for testing and viewing histograms from datataking runs. The next step was to perform high level tests by running in datataking mode on SR1 to characterize the problem and see if any initial clues could be found.

These tests were done using calibration firmware, in datataking mode with histogramming enabled and generating online occupancy graphs like in the noise mask generation procedure. These histograms represent online occupancy during a datataking run, showing the number of hits in each pixel of each FE in the module. The SIM_1A_SAFE_HISTO runconfig was used, which uses the emulator (SIM) and there are no actual hits in the system (all empty events, SAFE) with histogramming enabled (HISTO). Histogramming is off by default, so a config with HISTO enabled had to be created from SIM_1A_SAFE. Figure 2.4 shows examples of a clean histogram with no hits in this configuration, and a corrupted histogram that has a block of hits despite being in this config with no hits.

The issue does show up with SIM_1A_READY_HISTO with hits in the system as well, but doing tests without hits generates histograms that are cleaner and easier to keep track of since the corruption is very obvious.

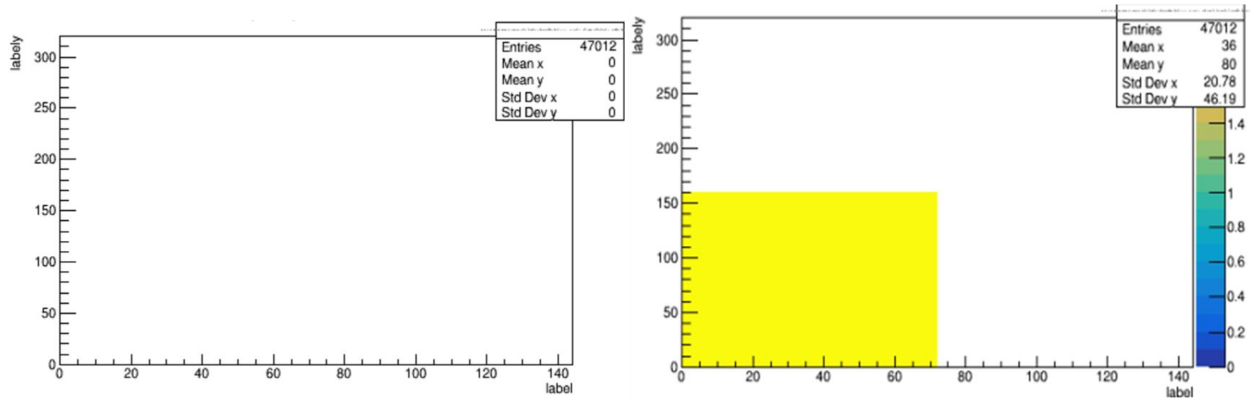


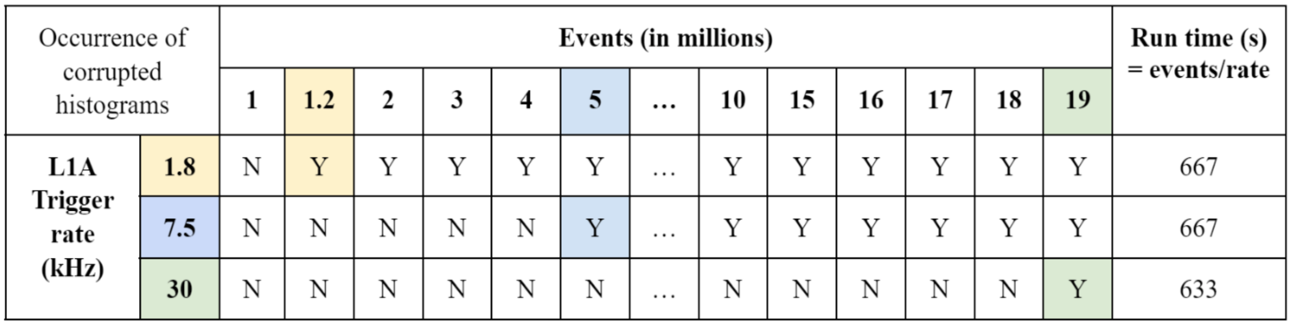
Figure 2.4: Expected histogram without hits (left) vs corrupted histogram (right).

Runs were performed with varying trigger rates and stopped after a varying number of events had passed to see if there was any dependence on these parameters. The trigger rate is the rate at which events are captured. So, if the Level-1 trigger rate is 30 kHz (which means 30k events are captured per second) and 18 million events have to be captured, datataking has to run for 18 million (events) / 30k (events/second) which comes out to be 600 seconds or 10 minutes of runtime. The process of running for a certain number of events, stopping the run and checking the histogram for corruption was done for three trigger rates, 1.8 kHz, 7.5 kHz and 30 kHz.

Corruption occurred at some point regardless of the trigger rate but at different numbers of events for each trigger rate. In Table 2.1, trigger rates, number of events passed and whether or not there was histogram corruption (Y/N) has been mapped out. The trigger rate, and first occurrence of corruption at that rate, and the number of events passed at the first occurrence of corruption have been highlighted in the same color. The runtime at which corruption appeared for the first time has then been calculated by using the highlighted numbers in the formula, run time = (number of events)/(trigger rate). Corruption continues to show up for runs with a greater number of events than the first occurrence of corruption.

A key observation was that the approximate run time at which the problem started showing up was the same. This quantity is similar (~630-670s, or ~10-11 minutes) for all trigger rates. The number is not exact because the trigger rate slightly varies throughout the run, and the number of events has been approximated. This runtime consistency was the major observation resulting from this stage of the investigation that gave some direction for the next steps to take.

Table 2.1: Histogram corruption with varying trigger rates and run time of first occurrence. (Y- corrupted histogram, N- clean histogram)

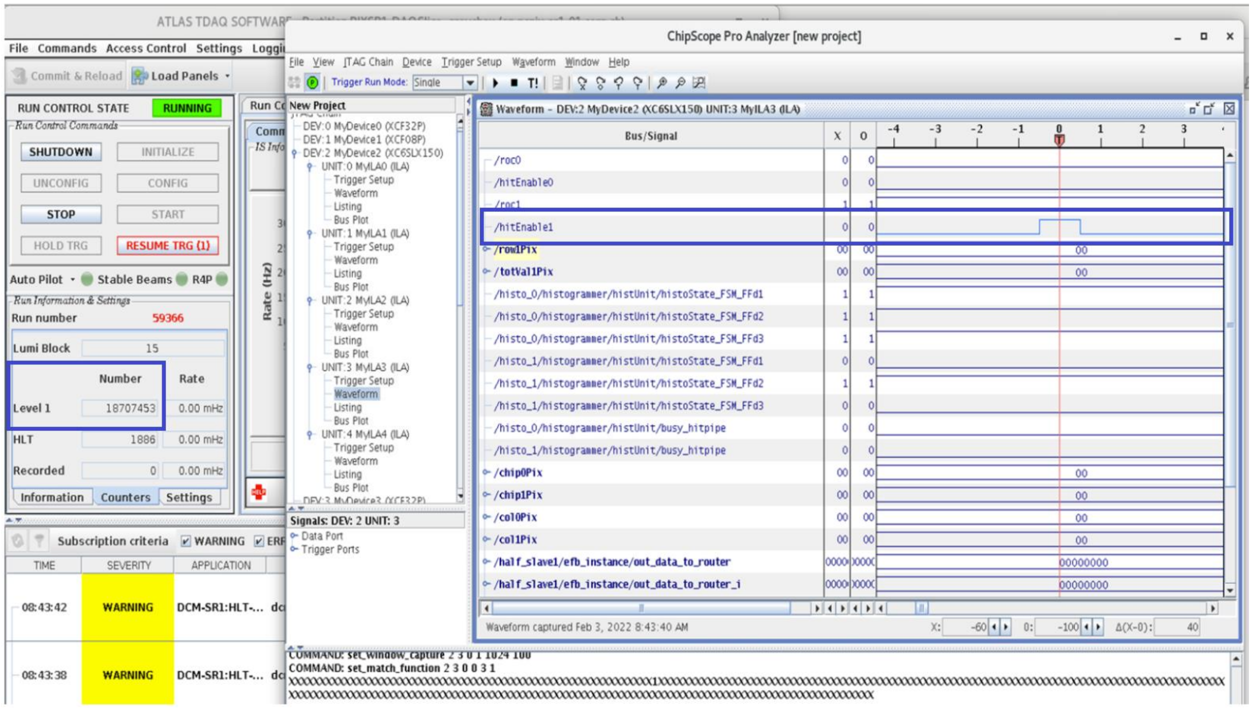


Occurrence of corrupted histograms		Events (in millions)													Run time (s) = events/rate
		1	1.2	2	3	4	5	…	10	15	16	17	18	19	
L1A Trigger rate (kHz)	1.8	N	Y	Y	Y	Y	Y	…	Y	Y	Y	Y	Y	Y	667
	7.5	N	N	N	N	N	Y	…	Y	Y	Y	Y	Y	Y	667
	30	N	N	N	N	N	N	…	N	N	N	N	N	Y	633

2.3 Further Investigation

The next step was to use the Xilinx ChipScope tool to look at signals in the FPGAs directly. ChipScope is an Integrated Logic Analyzer (ILA) that can be embedded into Xilinx FPGAs and allows the capture of signals in the hardware based on a set trigger pattern. Since the issue was showing up in the histograms, the Histogrammer module was the starting point. The objective was to figure out if the bug was within the Histogrammer or somewhere else in the path.

The inputs of the Histogrammer block were looked at during a datataking run, with the same config described earlier with no hits. The hitEnable signal indicates whether a valid hit is coming into the Histogrammer or not (histogram interfaces in Figure 1.13). During the run, the hitEnable signal was going high for the first time at around 10-11 minutes of runtime. This has been shown in Fig 2.5. The number of events passed is about 18.7 million which can be seen in the box at the left next to Level1 and under Number. The hitEnable1 signal, which is fourth from the top in the waveform, is being asserted around the same time as ~18.7 million events were reached. This indicated that the issue was not internal to the Histogrammer and was happening at an earlier stage in the datapath, since with no hits, the hitEnable signal should never be asserted. The timing lines up with the observation in the previous high-level tests, where it was seen that the first occurrence of corruption in the histogram was at around 19 million events at 30kHz.



RUN CONTROL STATE: RUNNING

Run Information & Settings		
Run number	59366	
Lumi Block	15	
	Number	Rate
Level 1	18707453	0.00 mHz
HLT	1886	0.00 mHz
Recorded	0	0.00 mHz

TIME	SEVERITY	APPLICATION
08:43:42	WARNING	DCM-SR1:HLT-...
08:43:38	WARNING	DCM-SR1:HLT-...

Waveform - DEV:2 MyDevice2 (XC6SLX150) UNIT:3 MyILA3 (ILA)

Bus/Signal	X	O
/roc0	0	0
/hitEnable0	0	0
/roc1	1	1
/hitEnable1	0	0
/row1Pix	00	00
/totVal1Pix	00	00
/histo_0/histogrammer/histUnit/histoState_FSM_FFd1	1	1
/histo_0/histogrammer/histUnit/histoState_FSM_FFd2	1	1
/histo_0/histogrammer/histUnit/histoState_FSM_FFd3	1	1
/histo_1/histogrammer/histUnit/histoState_FSM_FFd1	0	0
/histo_1/histogrammer/histUnit/histoState_FSM_FFd2	1	1
/histo_1/histogrammer/histUnit/histoState_FSM_FFd3	0	0
/histo_0/histogrammer/histUnit/busy_hitpipe	0	0
/histo_1/histogrammer/histUnit/busy_hitpipe	0	0
/chip0Pix	00	00
/chip1Pix	00	00
/col0Pix	00	00
/col1Pix	00	00
/half_slave1/efb_instance/out_data_to_router	0000	0000
/half_slave1/efb_instance/out_data_to_router_i	0000	0000

Waveform captured Feb 3, 2022 8:43:40 AM

```
COMMAND: set_window_capture 2 3 0 1 1024 100
COMMAND: set_match_function 2 3 0 0 3 1
```

Figure 2.5: Screenshot of TDAQ software window showing the number of events passed (~18.7 million) and ChipScope waveform of the hitEnable being asserted for the first time in a run with no hits (trigger rate = 30 kHz).

It was also observed that there was an incrementing pattern in the hit data in a specific order. So, for the first 160 hits, the row would go from 0x0 to 0x9F (159, max value of row in a FEI3) with everything else staying at 0. Then after all row values were cycled through once the column value would increment to 0x1 and stay at 0x1 until the row would go from 0 to 0x9F again with everything else being 0, and then column increased to 0x2 and cycled through row values until column value reached its maximum, 0x11. The MCC# (higher 4 bits of chipID) increments after all column values have been cycled through once. Similarly, this pattern is continued for ToT and FE# (lower 4 bits of chipID). Therefore, it goes in the sequence of row, column, MCC#, ToT, FE# with each field incrementing by one after the previous field cycles through all possible values. Consecutive hits have incrementing row values until the maximum is reached when it starts again from 0. The top waveform in Figure 2.6 shows how the row is increasing with consecutive hits. The bottom waveform in Figure 2.6, shows a snippet of consecutive hits where ChipScope was set to capture only if row = 0x9F and col = 0x11. It can be seen that ToT is 5 and chipPix (chipID) goes from 0x0F,0x1F, 0x2F, 0x3F and so on until 0x7F, which means that MCC# is increasing and FE# is at F. Then ToT becomes 6 and chipPix cycles through 0x0F to 0x7F again. Then ToT becomes 0 and the chip ID starts from 0x10,0x20...until 0x70 indicating that the FE# changed from F to 0.

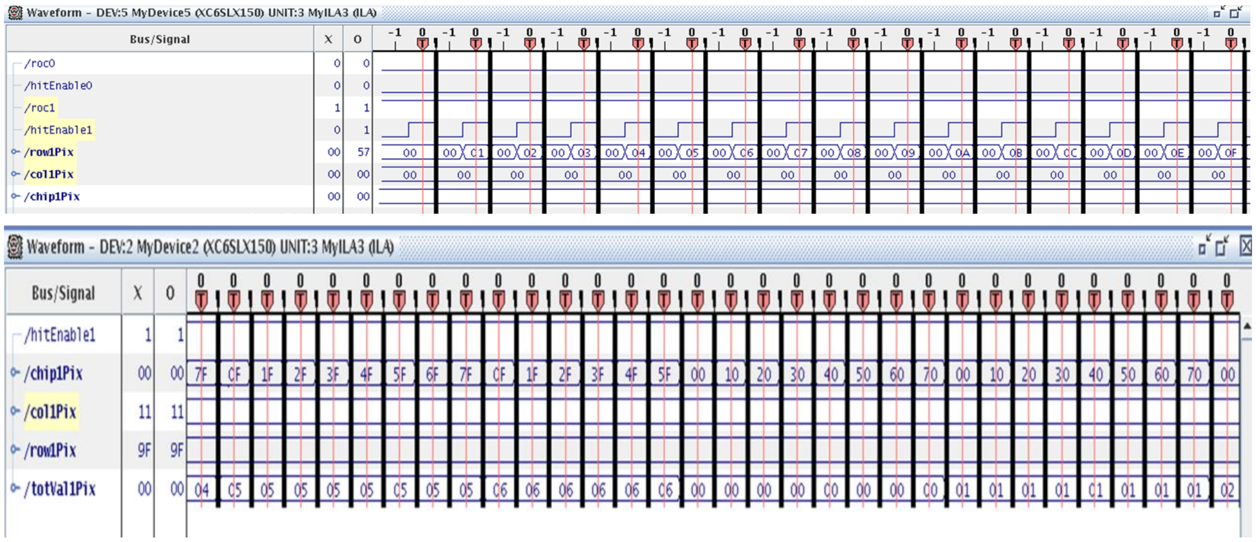


Figure 2.6: ChipScope waveforms showing the incrementing pattern. Top- triggering for consecutive hits, shows row increasing. Bottom- triggering for consecutive hits with fixed row and col, shows chipID and ToT increasing.

The next step was to look at the EFB and Router, which provide the inputs to the Histogrammer. The EFB sends header/trailer and module hit data to the Router in 32-bit words. The Router extracts only hit data out of all the data it receives for the Histogrammer. Upon doing more tests using ChipScope to see the data flowing through the EFB around the time the hitEnable is asserted, it was seen that there were no actual hits, but only S-Link header/trailers that were made up of data coming from the master. This was operating as expected and is shown in Figure 2.7 where the green boxes indicate that there was no hit data coming from modules and the red box shows where event information was coming from the master.

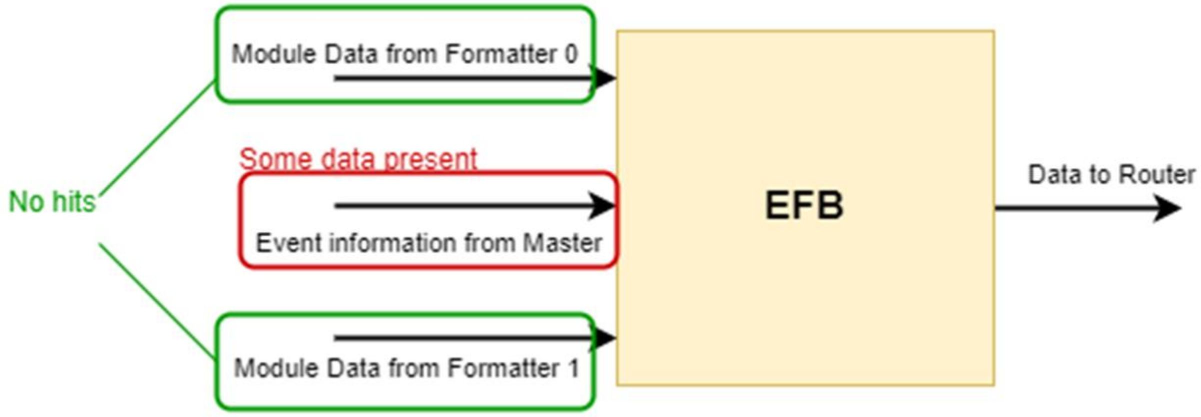

Module Data from Formatter 0
No hits
Some data present
Event information from Master
Module Data from Formatter 1
EFB
Data to Router

Figure 2.7: Representation of data flow observed in the EFB.

Figure 2.8 shows ChipScope waveforms with no hit data coming from the modules around the time the Router starts giving out hits, which are the second and third signals from the bottom called data_out showing all zeros and ev_header_data_out that consists of event information that are non-zero which is being sent to out_data_to_router, the data finally going into the Router.

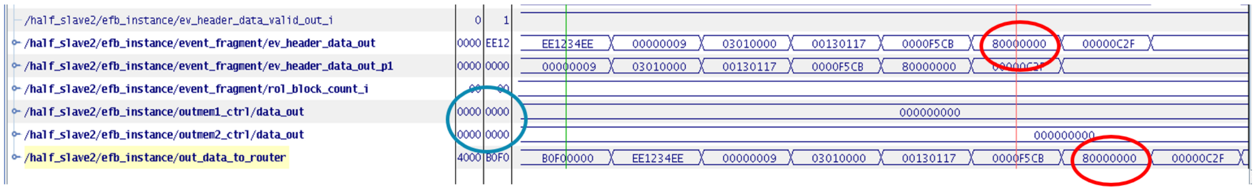
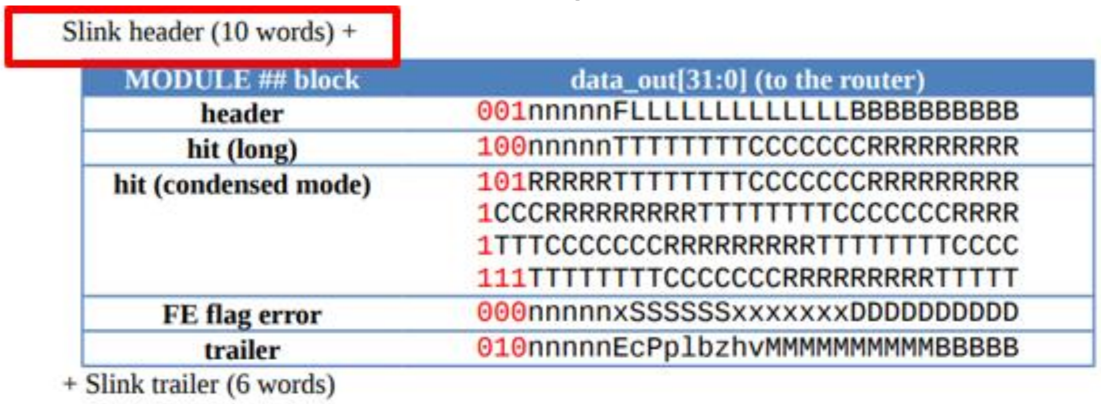

/half_slave2/efb_instance/ev_header_data_valid_out_i
/half_slave2/efb_instance/event_fragment/ev_header_data_out
/half_slave2/efb_instance/event_fragment/ev_header_data_out_p1
/half_slave2/efb_instance/event_fragment/rol_block_count_i
/half_slave2/efb_instance/outmem1_ctrl/data_out
/half_slave2/efb_instance/outmem2_ctrl/data_out
/half_slave2/efb_instance/out_data_to_router
EE1234EE
00000009
03010000
00130117
0000F5CB
80000000
00000C2F
B0F00000

Figure 2.8: ChipScope waveform showing EFB data going into the router. data_out- data from formatters, ev_header_data_out - master event data.

The Router gets data from the EFB in the format shown in Figure 2.9- S-Link header (event information), module data (module header, hit, module trailer) and S-Link trailer (event information) in 32-bit words. The Router checks for a specific bit pattern corresponding to a hit to determine whether that should be sent to the Histogrammer or not.



Slink header (10 words) +

MODULE ## block	data_out[31:0] (to the router)
header	001nnnnnFLLLLLLLLLLLLLBBBBBBBBBB
hit (long)	100nnnnnTTTTTTTTCCCCCCCRRRRRRRRR
hit (condensed mode)	101RRRRRTTTTTTTTCCCCCCCRRRRRRRRR
	1CCCRRRRRRRRRTTTTTTTTCCCCCCCRRRR
	1TTTCCCCCCCRRRRRRRRRTTTTTTTTCCCC
	111TTTTTTTTCCCCCCCRRRRRRRRRTTTTT
FE flag error	000nnnnnxSSSSSSxxxxxxxDDDDDDDDDD
trailer	010nnnnnEcPplbzhvMMMMMMMMMMBBBBB

+ Slink trailer (6 words)

n: link number	F: FeI4B flag bit
L: L1ID	B: BCID
T: hit ToT	C: hit column
R: row column	S: service code
D: service code counter	E: readout timeout error bit
c: condensed mode	P: link masked by PPC
p: preamble/header error	l/b: L1ID/BCID error
z: trailer timeout error	h: header/trailer limit error
v: row/column error	M: skipped trigger counter

nnnnn: slave ID (1b) – efb ID (1b) – formatter ID (1b) – link ID (2b)

Figure 2.9: Router input data format (IBL) [19].

The Router logic (specifically the hit extractor block shown in Figure 2.10) checks if the Router input is a hit by looking at whether bits 31-29 are ‘100’ (see data format in Figure 2.8, line called hit (long) has 100 highlighted in red) and if row and col values are in valid range. If that condition is met it extracts fields of hit data (row, col, ToT, chip) and adds a valid bit (which becomes the hitEnable signal) for the Histogrammer.

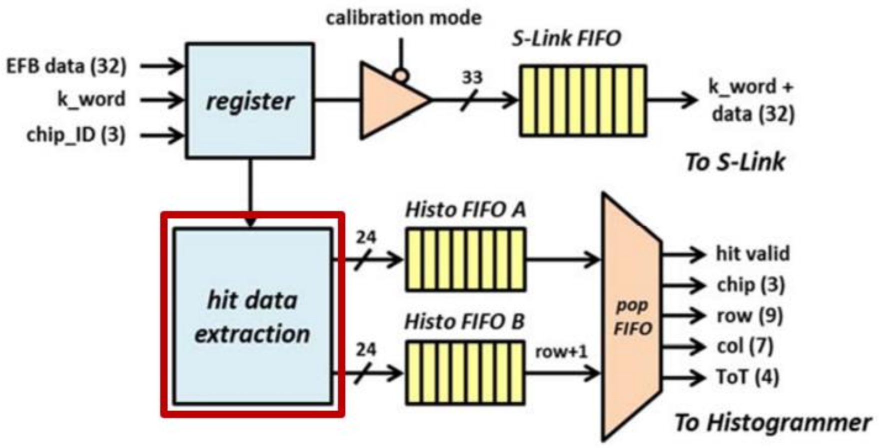
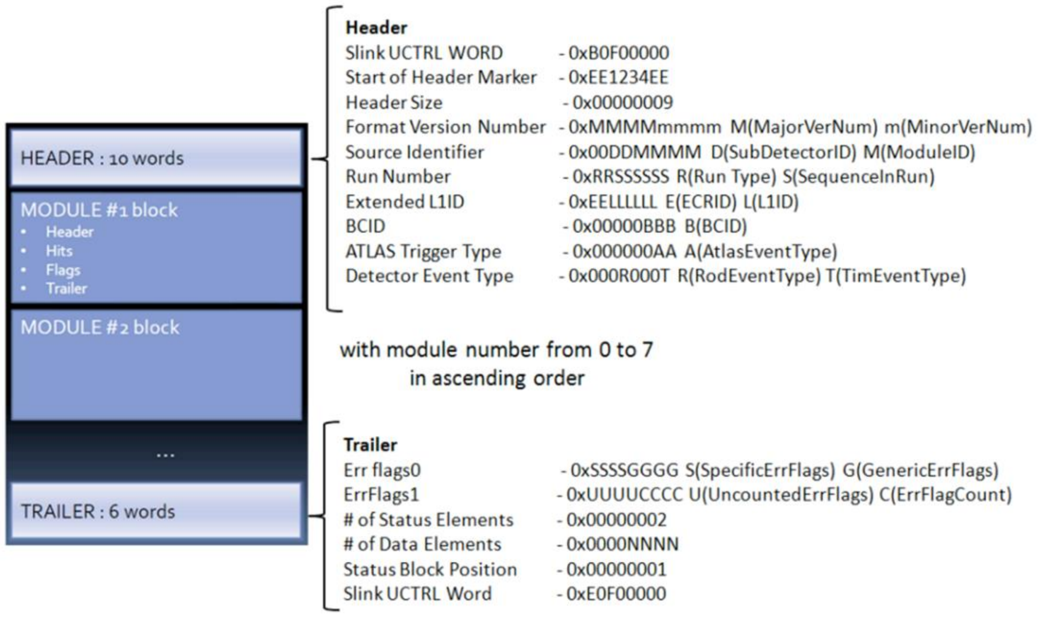


Figure 2.10: Schematic of router. Hit data extractor highlighted. Adapted from [19].

It was seen that hits were indeed coming out of the Router after about 10 minutes of runtime. This starts to make sense considering the Router logic and looking at incoming header data closely. The Extended L1ID word in the S-Link header is of particular interest, which is word 7 under Header in Figure 2.11. The Extended L1ID word consists of the ECRID and L1ID.

- ECR - Event Counter Reset done every ~5s to reset the L1ID counters. ECRID counts how many such resets are done.
- L1ID - Level1 ID is a count of the number of L1A trigger requests. This resets on every ECR.



Header

Slink UCTRL WORD	- 0xB0F00000
Start of Header Marker	- 0xEE1234EE
Header Size	- 0x00000009
Format Version Number	- 0xMMMMmmmm M(MajorVerNum) m(MinorVerNum)
Source Identifier	- 0x00DDMMMM D(SubDetectorID) M(ModuleID)
Run Number	- 0xRRSSSSSS R(Run Type) S(SequenceInRun)
Extended L1ID	- 0xEELLLLLL E(ECRID) L(L1ID)
BCID	- 0x00000BBB B(BCID)
ATLAS Trigger Type	- 0x000000AA A(AtlasEventType)
Detector Event Type	- 0x000R000T R(RodEventType) T(TimEventType)

Trailer

Err flags0	- 0xSSSSGGGG S(SpecificErrFlags) G(GenericErrFlags)
ErrFlags1	- 0xUUUUCCCC U(UncountedErrFlags) C(ErrFlagCount)
# of Status Elements	- 0x00000002
# of Data Elements	- 0x0000NNNN
Status Block Position	- 0x00000001
Slink UCTRL Word	- 0xE0F00000

Figure 2.11: Event Fragment format. The S-Link header consists of 10 words and word 7 in the Extended L1ID [19].

The Router checks for bits 31-29 of the input regardless of whether it is a S-Link header/trailer or module data (see Figure 2.9). Notice the structure of header word 7 consisting of ECRID and L1ID in Figure 2.11. It was observed that when ECRID=0x80 (128 in decimal), the Router accepts the header word as a hit and sends it through to the Histogrammer because input bits 31-29 are equal to ‘100’. This is shown in Figure 2.12 where the Router input data is the second signal from the bottom with 0x80000000 circled in red and hitEnable of the Histogrammer, second signal from the top being asserted also circled.

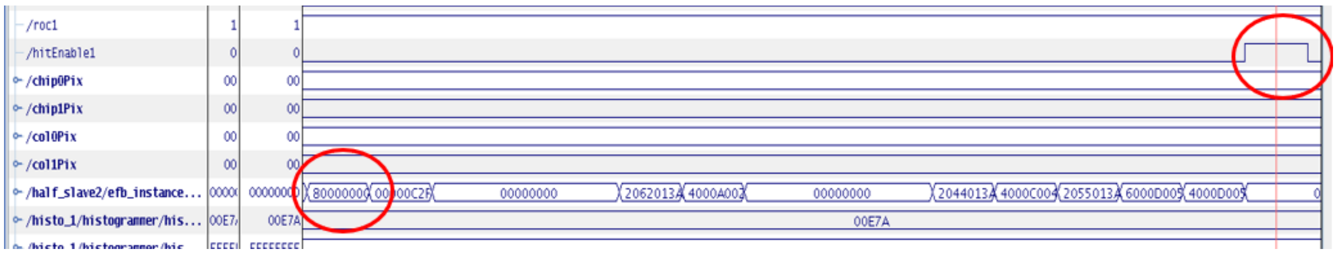
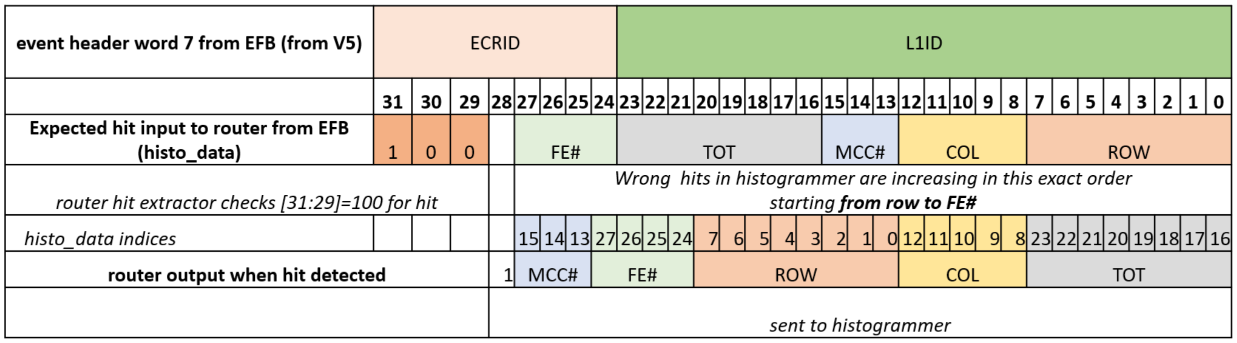

/roc1 1 1
/hitEnable1 0 0
/chip0Pix 00 00
/chip1Pix 00 00
/col0Pix 00 00
/col1Pix 00 00
/half_slave2/efb_instance... 0000 0000000 80000000 00 00C2F 00000000 2062013A 4000A002 00000000 2044013A 4000C004 2055013A 6000D005 4000D005 0
/histo_1/histogrammer/his... 00E7 00E7A 00E7A
FFFFI FFFFFFFF

Figure 2.12: hitEnable1 asserted after S-Link header data = 0x80000000 matches the Router check condition of input bits 31-29 = 100

It appears that the Router was misinterpreting certain header data values as a hit. The incrementing pattern described earlier makes sense because the extended L1ID word that is being misinterpreted as a hit is all counter data. So, values were cycling through exactly in the order of bits of the counters increasing. The mapping of the ECRID and L1ID S-Link header word to the Router input and fields of a hit has been shown in Figure 2.13. It can be seen that fields of a hit map onto the counter data exactly in the order in which they were cycling through, starting from row to FE#. The rearrangement of the data done by the Router to output to the Histogrammer is also shown.



event header word 7 from EFB (from V5)	ECRID								L1ID																							
	31	**30**	**29**	**28**	**27**	**26**	**25**	**24**	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Expected hit input to router from EFB (histo_data)	1	0	0		FE#				TOT								MCC#			COL					ROW							
router hit extractor checks [31:29]=100 for hit					*Wrong hits in histogrammer are increasing in this exact order starting* ***from row to FE#***																											
histo_data indices					15	14	13	27	26	25	24	7	6	5	4	3	2	1	0	12	11	10	9	8	23	22	21	20	19	18	17	16
router output when hit detected				1	MCC#			FE#				ROW								COL					TOT							
				sent to histogrammer																												

Figure 2.13: Mapping of extended L1ID word from S-Link header to router input format and its processing into a hit for the histogrammer.

The ECR period is a fixed value that is independent of trigger rate which explains why the time at which hits show up is independent of trigger rate. At the time of these tests, the ECR period was about 5s, so the ECRID reached 128 after about 128x5=640s of running. This is in line with the runtime dependence seen in initial tests.

2.4 ECR test

To further confirm the fact that the wrong hits were appearing due to the ECR counter, a final test was done by changing the ECR period to 10 s instead of 5 s in SR1. This would mean that the hits show up at ~1200s or 20 minutes instead of ~10 minutes. This is exactly what was observed, which confirmed the ECR dependence.

2.5 Solution implemented

A solution to this bug was to add a signal (am_i_dataword) to distinguish between module data words and header/trailer words from EFB to the Router and use this signal as a check in the Router to differentiate between actual hit data and other data. This was added in the gen_fragment block of the EFB where the S-Link header, module data and trailer are read out in order to generate event fragments. The Router then forwards hits to the Histogrammer only if am_i_dataword is asserted. This change has been implemented for both Pixel and IBL and has been tested in SR1 in both datataking (with histogramming enabled) and calibration modes. Histograms generated before and after applying the fix have been shown in Figure 2.14. It can be seen that after applying the fix, the histogram is empty, as expected, with no hits. The source code is in Appendix B.

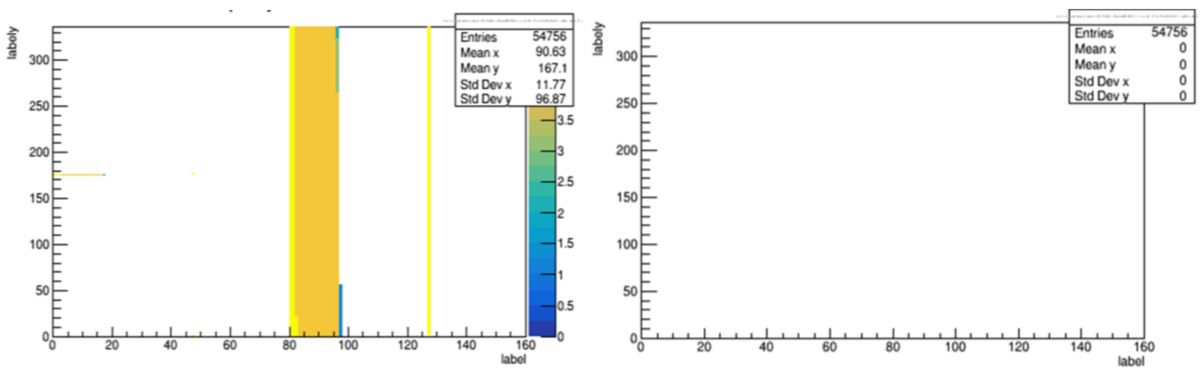


Figure 2.14: IBL histograms. Left- corrupted histogram before the fix. Right- clean histogram after applying the fix in firmware. Structure of the histogram is different because of the different number of FE's per module (2) compared to Pixel (16).

2.6 Conclusion

The Pixel Operations team tested the firmware with the fix in Summer 2022 and verified that calibration works correctly after this modification. The misinterpretation of header data as a hit (at ECR=128) also helped explain issues seen in the S-Link merger being designed by developers in the Pixel team. The S-Link merger combines data coming out of two Routers into one datastream to be sent in the S-Link via the BOC. Therefore, the issue in the histograms has been investigated and resolved with a fix that has been tested.

CHAPTER 3: SMART L1A FORWARDING

3.1 Motivation

In Run-3, the LHC has been delivering a higher luminosity than ever before, and high trigger rates (~100 kHz) are expected. This means that Pixel front end chips are facing challenging conditions. Front-end processing of this many hits may not be able to respond to all triggers in time, so these triggers are skipped in the MCC. If these skipped events are not accounted for correctly, there is a misalignment, or desynchronization induced in the data stream [22].

The number of triggers that have been skipped is sent with module data from the MCC to the ROD, so that the ROD can compensate by inserting dummy event data. However, discrepancies in the skipped trigger counts were observed in Run-2, which is suspected to be due to a malfunction in the MCC. In the challenging conditions during Run-3, where skipped triggers are more likely, this malfunction leads to desynchronization which lasts until the next Event Counter Reset (ECR). ECR is an ATLAS wide resynchronization signal, occurring every few seconds, which could correspond to several hundred thousand missed events between ECRs and hence millions of missed events over an entire datataking run.

Our Smart L1A Forwarding algorithm aims to get around this bug in the MCC through a fix in the ROD Slave firmware. Initial development of this algorithm was done by Nico Giangiacomi (INFN Bologna) in 2018-2020. This project was resumed in 2022. This section gives background on the algorithm. We then discuss my integration, testing and the additional modifications that were made to deploy this improvement to the actual ATLAS Pixel hardware.

3.2 Getting data from the Front End to the ROD

A trigger is the decision to capture an interesting event. For each event of interest, a trigger is sent from the Trigger system to the ROD. The ROD then sends a trigger command to the MCC (via the BOC). When the MCC gets this trigger command it tells the FE to capture the event. The MCC receives the corresponding hit data from the front-ends and writes it into an MCC FIFO buffer called the event buffer. This event buffer can ideally store up to 16 events. Events are read out of the buffer when the MCC sends module data to the ROD (via the BOC) for further processing.

Each FE keeps track of identifiers like Level1ID (L1ID) and Bunch Crossing ID (BCID) for every trigger and sends it to the ROD with hit data as shown in Figure 3.1. The EFB in the ROD slave gets event data from the ROD master which consists of another set of L1ID and BCID information from the TIM that comes with each trigger. A block in the EFB called bc_l1_check compares these two sets of L1IDs and BCIDs and sets the corresponding flags in the EFB output data stream in the module data header if there is a mismatch. This mismatch indicates desynchronization in the data stream because the FE's IDs and the ROD's IDs are misaligned. Events that are desynchronized are considered invalid data. There are cases (which will be discussed in following sections) when the FE cannot provide data in response to a trigger and the ROD has to insert dummy events to maintain synchronization.

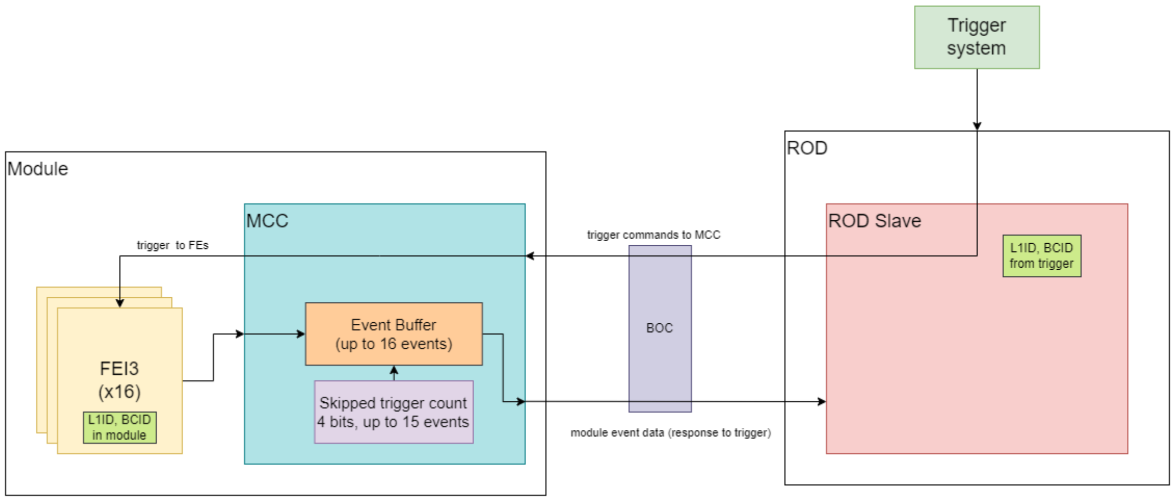


Figure 3.1: Simplified block diagram of how a trigger is sent from the ROD to a module.

Skipped trigger counter in MCC

It is possible that the MCC's event buffer fills up to its maximum capacity of 16 events, but the MCC still gets a trigger command. In this case, there is no space in the event buffer to store that data and this trigger has to be skipped. To handle such cases, there is a skipped trigger counter in the MCC that keeps count of the number of triggers skipped because of this lack of space in the buffer. This quantity is sent to the ROD so that empty events can be inserted to maintain synchronization. If there are skipped events, this information is sent with the last event that was in the buffer before the skips happened. This counter can keep count of up to 15 skipped events. The block diagram of the process of a trigger going through the ROD to the FE, and the FE sending data in response to the trigger is shown in Figure 3.1.

3.3 Problem with MCC's skipped trigger count

It has been seen that the skipped trigger count sent from the MCC to the ROD is sometimes incorrect. Because of this, the ROD ends up inserting the wrong number of dummy events in the data stream, resulting in desynchronization. This desynchronization lasts until the next Event Counter Reset, which happens every ~5s. At higher trigger rates, the chances of skipped triggers occurring are higher, since the buffer is filling faster and is not being emptied out quickly enough to make space for more events, and so this kind of desynchronization becomes more of a problem.

Since events are captured at trigger rates of up to 100 kHz, even a few seconds of desynchronization results in the loss of a significant amount of data.

3.4 Smart L1A Forwarding Algorithm

Since the MCC is an ASIC that cannot be modified, the issue of wrong skipped trigger count values cannot be solved by directly changing anything in the MCC, and a different solution is required. The Smart L1A Forwarding algorithm [23] is a mechanism in the ROD Slave FPGA developed to mitigate desynchronization happening because of the wrong skip trigger counts from the MCC. There is extra logic in the ROD Slave to allow more control over when to forward LIA triggers to modules.

The ROD now keeps track of triggers sent and event data received back as a response to the triggers from each module. When the difference between sent triggers and received events (i.e. the triggers in flight value) exceeds a defined pending trigger threshold value, the ROD stops sending triggers to that module, and only starts sending triggers again when it receives more responses and the number of triggers in flight no longer exceeds the threshold. When triggers are inhibited in this way, as in not actually sent to the module, the ROD inserts dummy events as a response to these triggers being inhibited, to maintain synchronization in the data stream. Figure 3.2 shows where the SmartL1A algorithm lies in the process of sending a trigger and receiving module data.

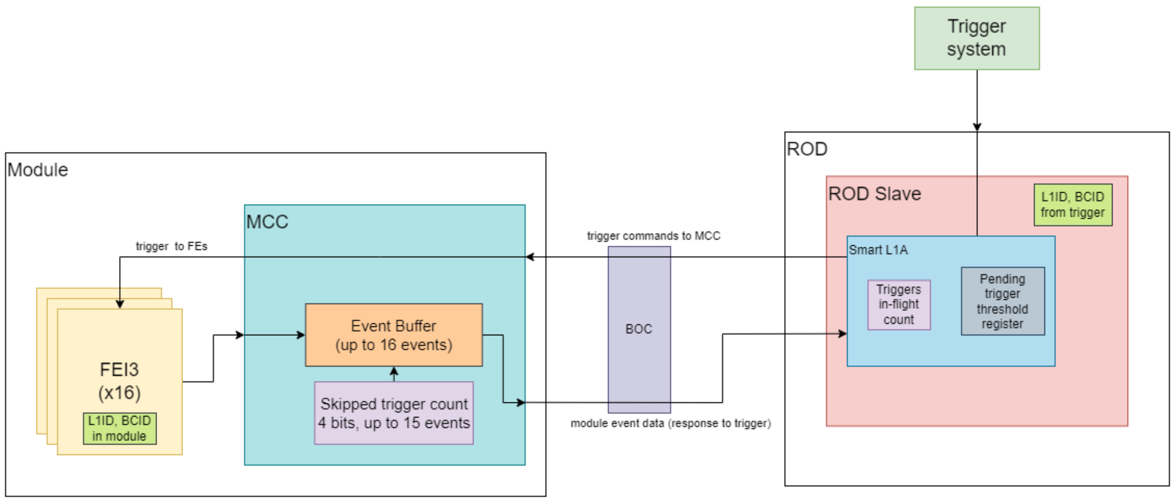


Figure 3.2: Simplified block diagram of how a trigger is sent from the ROD to a module including the SmartL1A algorithm.

3.5 Inefficient events in Pixel

Any empty events inserted by the ROD or desynchronized events (L1ID or BCID mismatch) are considered inefficient events. These events are considered invalid data that cannot be used for physics studies. The inefficiency being discussed here is at the Pixel module level, involving invalid data from the ROD in Pixel. This is different from ATLAS-wide general efficiency metrics which are at a higher level. Our goal is to minimize the number of events that are inefficient in Pixel. The different types of inefficient events are shown in Figure 3.3. Empty events are any dummy events inserted by the ROD, and desynch is inefficiency caused by systematic problems. Important event data formats associated with inefficient events are described in Appendix C5, while the complete data format can be found in [25].

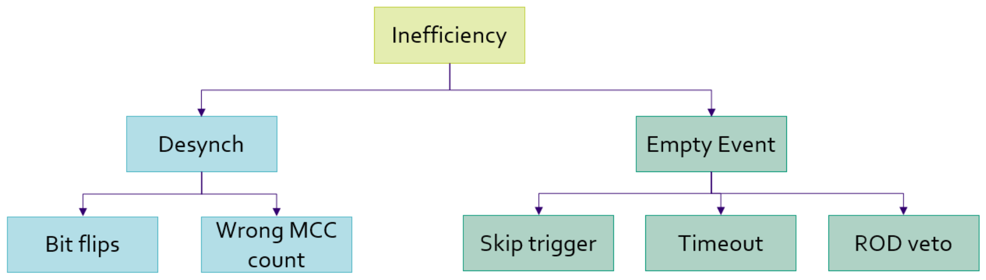


Figure 3.3: Classification inefficiencies in Pixel.

Empty events inserted

1. Skipped Trigger: Empty events that are inserted by the ROD using the skipped trigger count from the MCC.
2. ROD Vetoed: Empty events by the SmartL1A algorithm in the ROD when the number of triggers that are in-flight exceeds the threshold, and therefore triggers are inhibited.
3. Module Timeout: This type of event is inserted when the module does not reply to a trigger in time (the time limit is defined in a ROD slave register).

Desynchronization

1. Mismatch in L1ID or BCID because of anything else (such as bitflips).
2. Misalignment, and hence mismatch in L1ID and BCID induced because of the wrong skipped trigger count from the MCC.

All of these inefficiencies (both empty events and systematic desynchronization) are meant to be flagged as desynchronization (L1ID/BCID mismatch) by setting bits 26(L1ID) or 25(BCID) of the header when module data is formatted and built into event fragments in the EFB in the ROD slave. There is no other way to distinguish between good and bad events during offline data analysis. This flag gives an indication of any inefficient event that is invalid and cannot be used for physics. The type of empty event or desynchronization is not considered during physics data analysis, and everything is counted as total desynchronization. However, monitoring the types of inefficient events is important for testing this algorithm and this has been tracked using additional counters in the ROD firmware.

3.6 Initial implementation

The SmartL1A algorithm's initial implementation in firmware was done by Nico Giangiacomi. Some modifications were made by me in the monitoring counters. Further modifications made after testing have been discussed in Section 3.10.

Mechanism

The SmartL1A forwarding mechanism was implemented in firmware by keeping track of triggers sent and events received in every formatter link that is getting data from a module. The pending trigger threshold (6 bits) is set in the pendTrigThrReg register at address 0x814 (8 bits) in the ROD Slave register block. The default value of this register is 0xFF, indicating that the mechanism is switched off.

A trigger FIFO at the top level of each ROD Slave keeps track of whether a trigger was inhibited or sent to modules.

In the quad link formatter, the difference between the number of sent triggers and received events, which are the triggers in-flight, is stored in tr_tbprocessed (triggers to be processed). Sent events are tracked using the trigger FIFO at the top level and received events are obtained by checking data coming in through the formatter links from the BOC for the module trailer. The skipped trigger count from the module header was also factored into the calculation of received events. Each formatter link has a signal called modPendingStatus. If the in-flight triggers (tr_tbprocesssed value) in a link exceeds (not equals) the threshold, the modPendingStatus signal of that link is set to 0 else it is set to 1. Changes also had to be made in the fifo_readout block in the formatter to account for this new type of empty event and insert it whenever a trigger has been inhibited to maintain synchronization. This block also handles adding L1ID corrections to the module data since if triggers are inhibited, the module's L1ID data will be misaligned from the L1ID in the TIM.

At the top level the decision on whether or not an incoming trigger will be sent to the module is taken based on the modPendingStatus signal from the formatters. If the modPendingStatus for a link is 0, then the trigger will be inhibited in the corresponding serial line to the module.

The number of pending triggers was added in bits [9:4] of the event trailer for monitoring and debugging. An overview of the firmware implementation of the SmartL1A mechanism has been shown in Figure 3.4.

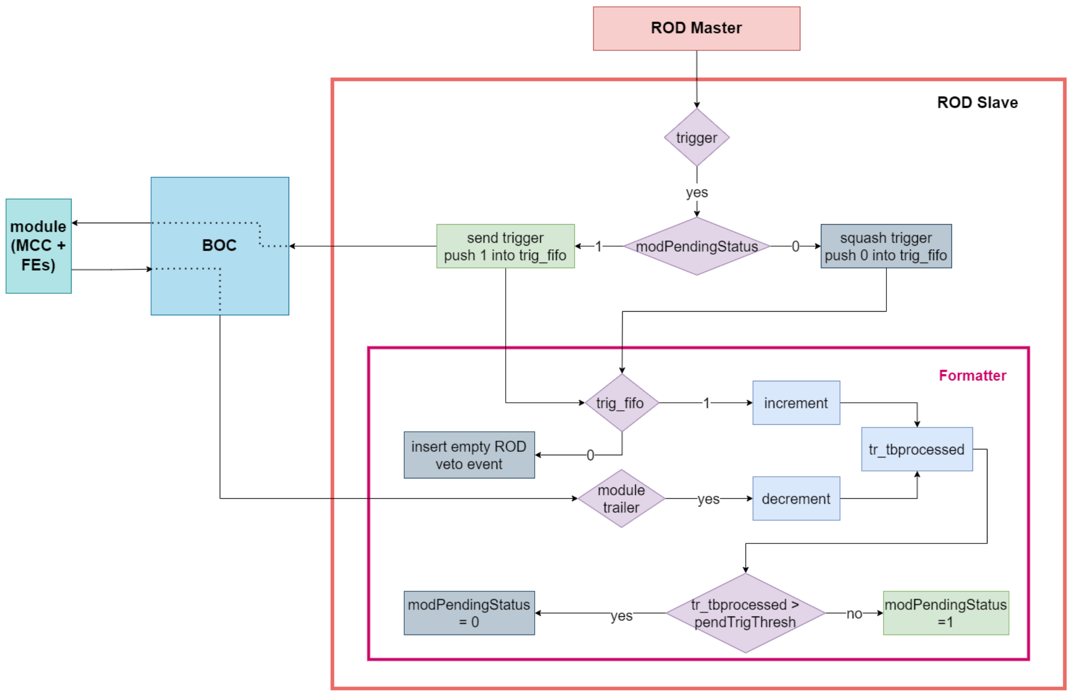


Figure 3.4: Overview of the SmartL1A forwarding algorithm firmware implementation. This process happens independently in each formatter link.

Counters and monitoring registers

There is a module in the firmware (Desynch_monitor) that keeps track of the total number of events and various inefficiencies. A few changes were made here to help keep track of different types of inefficiency. Existing registers are configured to be read by monitoring systems in the detector, whereas newly added registers were only used for development in SR1 at the time of writing.

- Total Inefficiency (any kind of inefficiency as described above): existing registers
- ROD veto: HTLimit regs are being used to store these.
- Skipped triggers: new registers added
- Timeout: existing registers
- Global event counter: The original implementation only has a formatter event counter. This would not count any ROD inserted events. An additional global event counter has been added to keep track of all events. S-Link merged data had to be used for this to be accurate for the 160Mb/s readout implementation.
- tr_tbprocessed of each link was being sent to new registers for debugging. This does not have to be used in actual monitoring.

Adding this implementation into current datataking firmware

The following changes were made to integrate the algorithm originally developed by Nico into the current datataking firmware:

- Extra core (.xco file) for top level trigger_fifo
- Extra cores for 2 types of counters- 24-bit (for occupancy in formatter) and 19-bit (desynch_monitor) versions
- Updates in data format and insertion of dummy events – formatter (data_format, fifo readout)
- Quad link formatter – computation of tr_tbprocessed
- Modified counter widths in occupancy_monitor and desynch_monitor
- Added new registers and addresses in ibl_slv_reg, rodSlaveRegPack
- Top level – identifying and squashing triggers if the pending triggers exceeds the threshold
- Desynch monitor- modified monitoring counters, added global event counter

3.7 Testing infrastructure in SR1

After porting the original implementation into the current datataking firmware, testing was done by varying parameters such as trigger rate and pending trigger threshold to verify the functionality of the mechanism and identify optimal parameters.

The testing of this algorithm required hits coming in through the MCC. Modules in SR1 had to be used since the emulator which is in the BOC generates ideal module data that would not have wrong skip count values. A high number of hits was also required to replicate challenging conditions. Since SR1 does not have a particle beam, this could only be achieved by switching on modules and detecting hits because of noise.

To induce as much noise as possible in the sensors in the modules, the modules were configured in a specific way. LV, the low voltage required to power modules, was switched on. HV, the high voltage for sensor biasing, was switched off. The preamps, which are preamplifiers in the front end that shape the input current pulse from the sensors, were switched on. In this configuration we generated a significant amount of noisy, random events, to test the Smart L1 algorithm.

There are other parameters that could be configured in the testing infrastructure. Deadtime is a parameter in the trigger system that ensures a time between triggers so that the system can catch up. Simple deadtime is the deadtime (in BC) that has to be present between two triggers. Complex deadtime is based on the leaky bucket model, with X as the size of the bucket (L1A triggers) and R being the time (in BC) to leak 1 L1A. More details can be found in [9].

Ultimately, the tests at highest trigger rate (100 kHz) were the most important since the issue of excessive desynchronization because of the MCC bug arises only at high trigger rates. Most of the testing infrastructure was based off of Nico's initial tests. Modifications were made to take into account new registers and visualization of collected data.

The monitoring registers were read and dumped in a file every second for 10-20 minutes of datataking. Counters are read using the Occupancy_Desynch_countersRead command. ROD Slave registers had to be written on the fly during datataking runs to change pendingTrigThreshReg configurations. This was done using the writePendTrigVme command. See Appendix C2.

The counters are set up to get reset on every read. This is part of a mechanism to stop the counters from incrementing while the read is in progress so that all values are from the same instant in time. Counters were originally 32 bits wide in the original datataking firmware and the SmartL1A implementation by Nico, but smaller widths of 19 (for event counters) and 24 (for occupancy counters) were chosen such that enough events could be safely captured in between reads and resets without wasting FPGA resources.

Quick Status (QS) is a monitoring system running in the PowerPC in the ROD Master that regularly reads and resets monitoring registers. This is used for monitoring in the detector with additional infrastructure but SR1 presently does not have a way to record and visualize these values in a similar way. However, the reading and resetting of registers by QS in the background would interfere with the reading of the registers done independently by Occupancy_Desynch_countersRead during tests. Therefore, a special runconfig was created with QS disabled – NORM_1A_80_QS_OFF.

The events and inefficiencies over time are extracted from the dump files by another script and given out in a format ready for processing such as calculation of cumulative events and percentage of inefficiency and visualization. A Python notebook was used to create graphs and calculate inefficiency metrics in percentages from the extracted data for each run and module. See Appendix C4.

3.8 Test in SR1 with modules

The initial implementation ported into the current datataking version was tested in SR1 using modules at a readout speed of 80 Mb/s in various crates and slots. Tests were done and data was collected using the infrastructure and steps described above.

To begin with, tests were done on just one module in SR1 to replicate Nico's tests [24] and refine the testing infrastructure. The graphs in Figure 3.5 show the cumulative number of skipped triggers, ROD veto by the SmartL1A mechanism, timeout, all inefficient events and total events for different thresholds at a trigger rate of 100 kHz. Looking at the graph with the mechanism off and threshold set to 20, it can be seen that whenever skipped triggers are introduced into the system in the graphs with threshold the desynchronization shoots up. In the graph with threshold set to 15, there are no skipped triggers. The overall inefficiency is the lowest at a threshold of 15. A pending trigger threshold value of 15 would mean that 16 triggers actually have to be sent to get the number of triggers in flight to be greater than the threshold of 15 and then following trigger (17th trigger) would be inhibited. Figure 3.6 shows a plot of the total inefficiency against the pending trigger threshold where a threshold of 15 has the lowest inefficiency compared to thresholds ranging from 8 to 20 and the mechanism switched off.

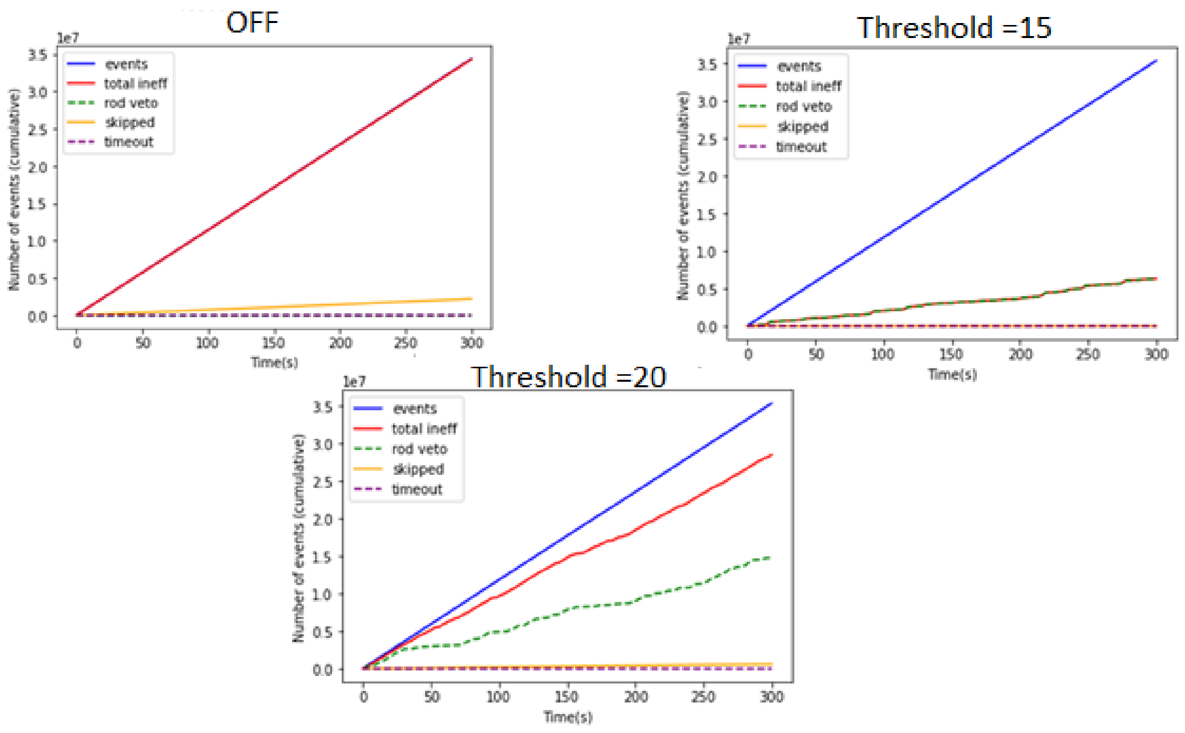


Figure 3.5: Graphs of different events (cumulative) vs time for the mechanism not enabled (top left), threshold of 15 (top right), threshold of 20 (bottom).

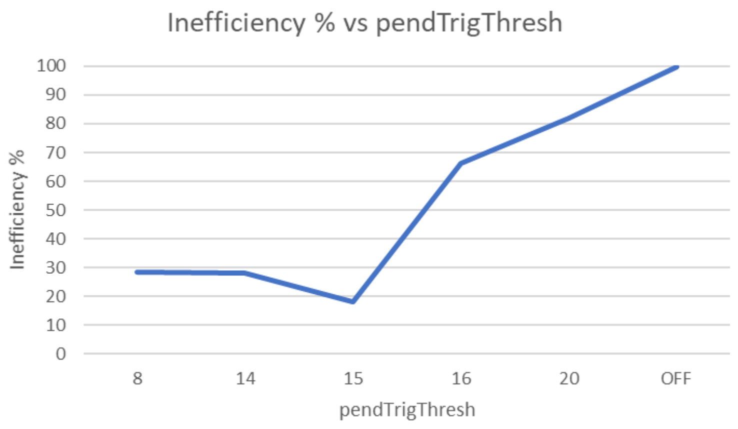
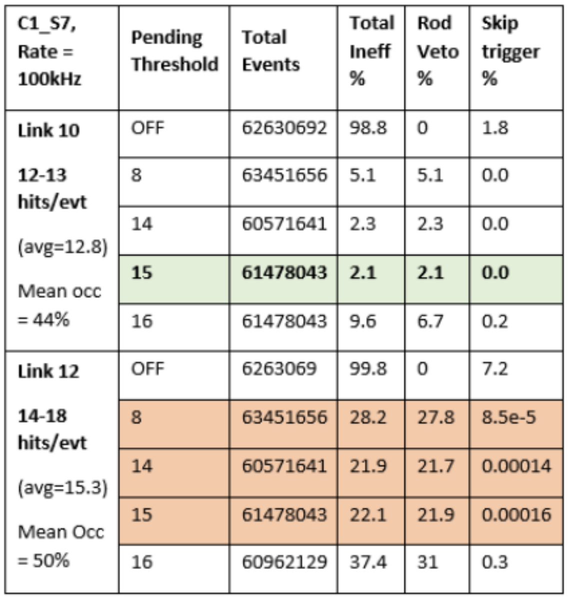


Figure 3.6: Overall inefficiency% vs threshold from an initial 1 module test.

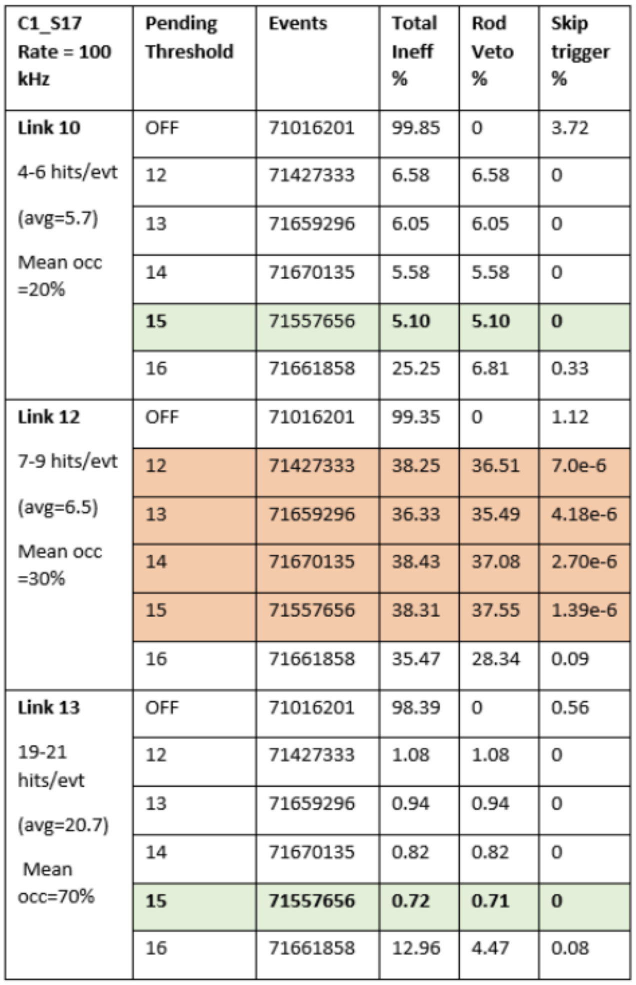
Further tests were performed to see if this behavior holds up in other crates and modules in SR1. The simple deadtime was set to be 5 and complex deadtime was set to be 15/400. Tables 3.1-3.3 show percentages of inefficiency with different thresholds that were found during tests. Information on hits per event and link usage (percentage of bandwidth usage of link from MCC to BOC) has also been included. Link numbers here correspond to Formatter links getting data from an MCC/module each.

Table 3.1: Crate 1 slot 7



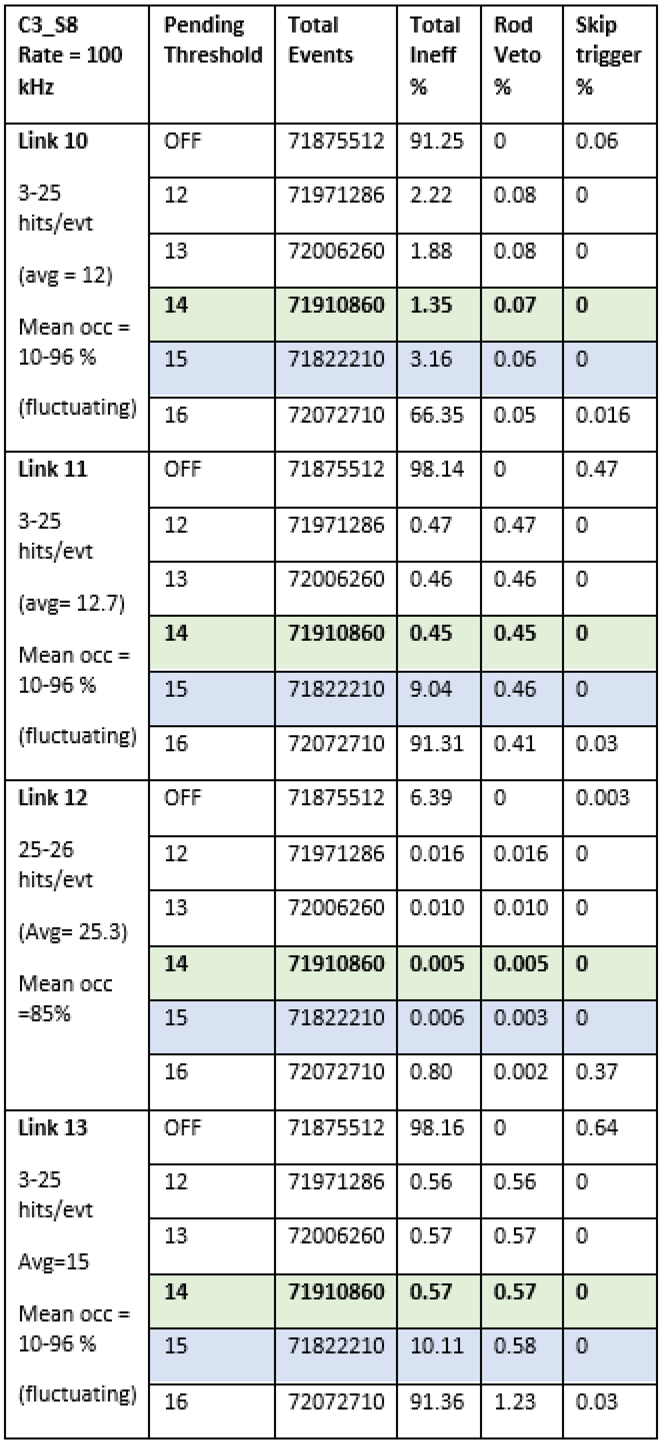
C1_S7, Rate = 100kHz	**Pending Threshold**	**Total Events**	**Total Ineff %**	**Rod Veto %**	**Skip trigger %**
Link 10 **12-13 hits/evt** (avg=12.8) Mean occ = 44%	OFF	62630692	98.8	0	1.8
	8	63451656	5.1	5.1	0.0
	14	60571641	2.3	2.3	0.0
	15	**61478043**	**2.1**	**2.1**	**0.0**
	16	61478043	9.6	6.7	0.2
Link 12 **14-18 hits/evt** (avg=15.3) Mean Occ = 50%	OFF	6263069	99.8	0	7.2
	8	63451656	28.2	27.8	8.5e-5
	14	60571641	21.9	21.7	0.00014
	15	61478043	22.1	21.9	0.00016
	16	60962129	37.4	31	0.3

Table 3.2: Crate 1 slot 17



C1_S17 Rate = 100 kHz	**Pending Threshold**	**Events**	**Total Ineff %**	**Rod Veto %**	**Skip trigger %**
Link 10 4-6 hits/evt (avg=5.7) Mean occ =20%	OFF	71016201	99.85	0	3.72
	12	71427333	6.58	6.58	0
	13	71659296	6.05	6.05	0
	14	71670135	5.58	5.58	0
	15	71557656	**5.10**	**5.10**	**0**
	16	71661858	25.25	6.81	0.33
Link 12 7-9 hits/evt (avg=6.5) Mean occ =30%	OFF	71016201	99.35	0	1.12
	12	71427333	38.25	36.51	7.0e-6
	13	71659296	36.33	35.49	4.18e-6
	14	71670135	38.43	37.08	2.70e-6
	15	71557656	38.31	37.55	1.39e-6
	16	71661858	35.47	28.34	0.09
Link 13 19-21 hits/evt (avg=20.7) Mean occ=70%	OFF	71016201	98.39	0	0.56
	12	71427333	1.08	1.08	0
	13	71659296	0.94	0.94	0
	14	71670135	0.82	0.82	0
	15	**71557656**	**0.72**	**0.71**	**0**
	16	71661858	12.96	4.47	0.08

Table 3.3: Crate 3 slot 8



C3_S8 Rate = 100 kHz	Pending Threshold	Total Events	Total Ineff %	Rod Veto %	Skip trigger %
Link 10 3-25 hits/evt (avg = 12) Mean occ = 10-96 % (fluctuating)	OFF	71875512	91.25	0	0.06
	12	71971286	2.22	0.08	0
	13	72006260	1.88	0.08	0
	14	**71910860**	**1.35**	**0.07**	**0**
	15	71822210	3.16	0.06	0
	16	72072710	66.35	0.05	0.016
Link 11 3-25 hits/evt (avg= 12.7) Mean occ = 10-96 % (fluctuating)	OFF	71875512	98.14	0	0.47
	12	71971286	0.47	0.47	0
	13	72006260	0.46	0.46	0
	14	**71910860**	**0.45**	**0.45**	**0**
	15	71822210	9.04	0.46	0
	16	72072710	91.31	0.41	0.03
Link 12 25-26 hits/evt (Avg= 25.3) Mean occ =85%	OFF	71875512	6.39	0	0.003
	12	71971286	0.016	0.016	0
	13	72006260	0.010	0.010	0
	14	**71910860**	**0.005**	**0.005**	**0**
	15	71822210	0.006	0.003	0
	16	72072710	0.80	0.002	0.37
Link 13 3-25 hits/evt Avg=15 Mean occ = 10-96 % (fluctuating)	OFF	71875512	98.16	0	0.64
	12	71971286	0.56	0.56	0
	13	72006260	0.57	0.57	0
	14	**71910860**	**0.57**	**0.57**	**0**
	15	71822210	10.11	0.58	0
	16	72072710	91.36	1.23	0.03

During these tests, it was seen that some modules in crate-1 had skipped triggers showing up regardless of the threshold (highlighted in orange in Tables 3.1-3.2), which is suspected to be due to bitflips or a fault in the modules. The results in modules in crate-1 (without erroneous skipped triggers) show a good decrease in inefficiency with the mechanism. The best efficiency is achieved at a threshold of 15 (highlighted in green), meaning that 16 triggers are sent and allowed to be in-flight before triggers are inhibited by SmartL1A, and there are no skipped triggers in the system. This is likely because the MCC's event buffer can hold 16 events without skipping triggers. At a threshold of 16 (resulting in 17 triggers in flight), skipped triggers come into play because there is not enough space in the MCC buffer. The MCC sends inaccurate skipped trigger values and desynchronization increases significantly. A threshold less than 15, 14 for example would not use the MCC event buffer to its full potential and would result in dummy events being inserted when that could have been real data from front ends since there is space in the buffer. This is not optimal and results in a slight increase in inefficiency because the dummy events are invalid data. Therefore, to get the best out of the system, the MCC buffer should be fully utilized without letting skipped triggers occur, which is happening at a threshold of 15.

There was anomalous behavior in crate-3 slot-8 where the best performance was at 14 and desynchronization significantly increased at a threshold of 15 but no skipped triggers were seen in the system at or below 15. Values at 14 and 15 respectively have been highlighted in green and blue in Table 3.3. It is unlikely that it is something to do with the algorithm since it shows up only in this crate, and an overall systematic problem would show up everywhere. It is possible that this crate might have been too noisy or faulty, since these tests are only dependent on the noise in the test environment.

In the worst-case scenario, using 14 as the threshold is also reasonable since the difference in inefficiency between 15 and 14 is marginal compared to the overall improvement in mitigating desynchronization. No significant effect on inefficiency was seen by varying deadtime parameters in SR1.

The final decision on viability and parameters has to be taken based on tests in the detector. These tests were done in SR1 and relied on noise so the performance in the conditions of the real detector is unknown.

3.9 Tests in the ATLAS Detector – 2022

A ROD in Disk-1 (D1_S15), which runs at a readout speed of 80 Mb/s was loaded with the SmartLlA forwarding firmware to test the algorithm in the detector by the Pixel Operations team in November 2022.

No additional testing infrastructure was required in the detector since there is already a Pixel dashboard [26] used to record and visualize metrics tracked by QS. This allowed real-time monitoring of the total desynchronization and ROD-veto events (going into HTLimit registers at the time of testing) which were of primary interest for this test.

A decrease in desynchronization (total inefficiency) was observed compared to having the mechanism off, and threshold of 15 was the best performing as observed in SR1. The graphs shown here are from the Pixel Dashboard, plotting the percentage of desynchronization (total inefficiency) against time. As shown in Figure 3.7a, desynchronization immediately went up when the threshold was set to 16, and the zoomed-in graph in Figure 3.7b shows that at 15 there is the lowest desynchronization. A precise decrease in desynchronization cannot be calculated with the limited data available at the time of writing. However, a rough estimate can be made by considering the desynchronization without the mechanism to be about 0.5% (Figure 3.8) and 0.015% with the threshold being 15 (Figure 3.7b), resulting in an inefficiency% reduction of at least 30 times. It should be noted that these values can vary greatly depending on the conditions of the run.

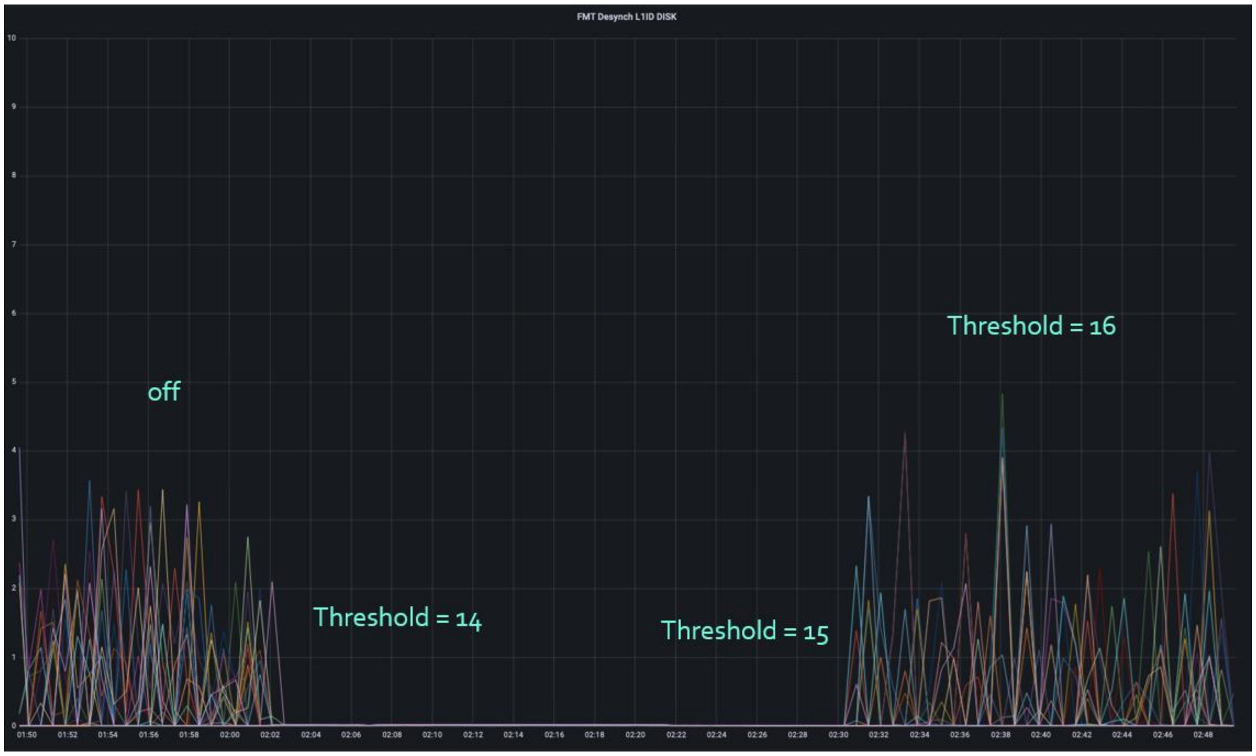


Figure 3.7a: Desynchronization when switching on SmartL1A forwarding with threshold =14, 15 and 16.

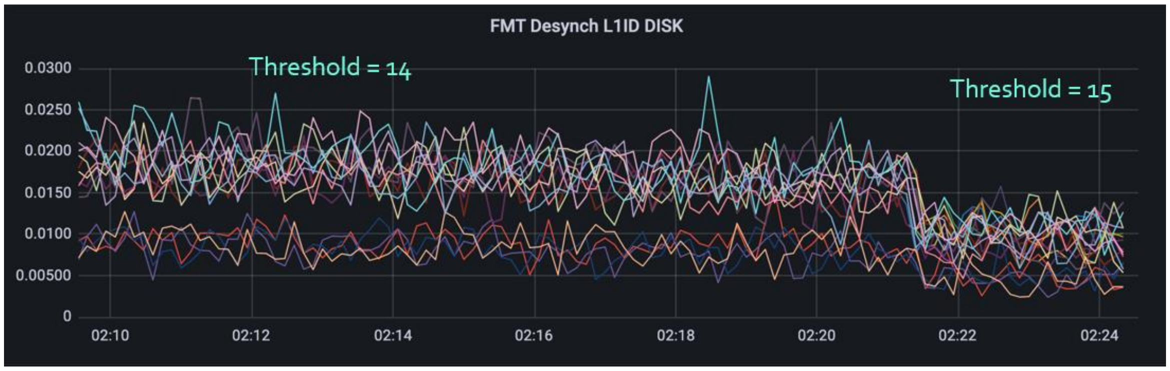


Figure 3.7b: Zoomed into desynch when switching from threshold of 14 to 15.

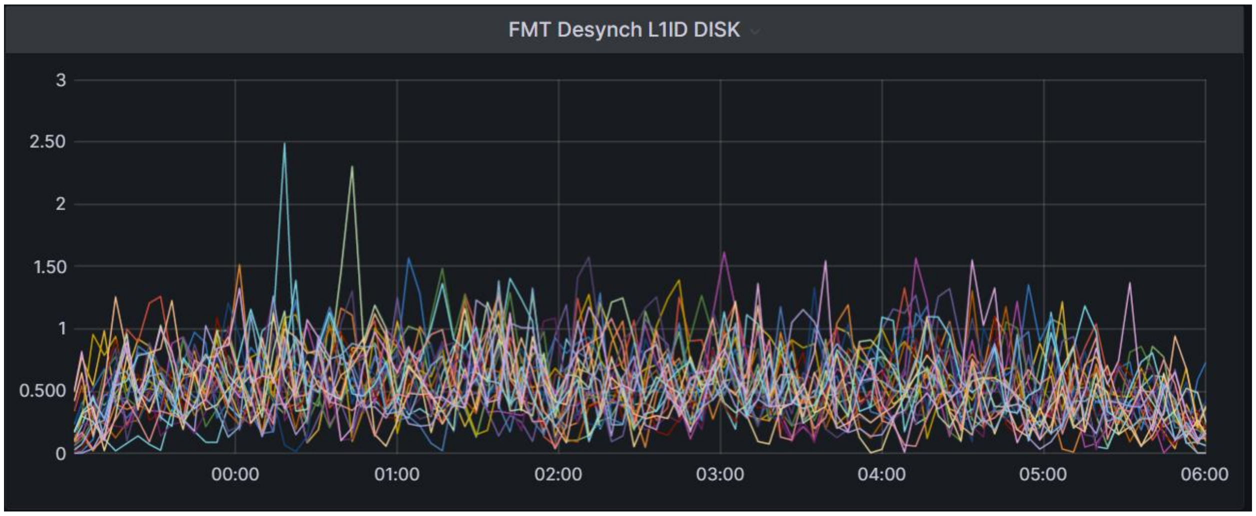


Figure 3.8: Desynch in D1_S15 without the mechanism for reference.

However, there were issues in 2 out of 3 runs where some modules (two at a time) got desynchronized and never recovered. The number of triggers to be processed was suspected to be stuck at a high number. The link usage for the two modules dropped to zero and all triggers were being ROD vetoed. This was happening with thresholds of 14 and 15. Two modules went into desynchronization at the same time which was expected considering link sharing in Disk-1 (will be described in the next section). In Figure 3.9, it can be seen that the desynch value shot up randomly, although no changes in the settings were made at this point. The desynchronization percentage is extremely high in the plot because of the method used by the monitoring system to calculate the percentage of desynchronization. It divides the number of desynchronized events only by the number of events coming from modules. It does not count ROD inserted dummy events. In this case the events from the formatter were very low because triggers were all being inhibited so modules were not sending data. All the events were dummy ROD veto events inserted by the ROD due to the SmartL1A mechanism. So, when the monitoring system counts all the ROD dummy events as desynchronization and divides that by the very low number of module events, the desynchronization percentage appears to be much greater than 100%.

Another run that was about 6 hours long went without any problems. No further testing was possible in the detector in 2022 due to the Year End Technical Shutdown (YETS). This behavior could not be replicated in SR1. The lower number of functional modules attached to RODs in SR1 compared to the real detector is a possible limitation.

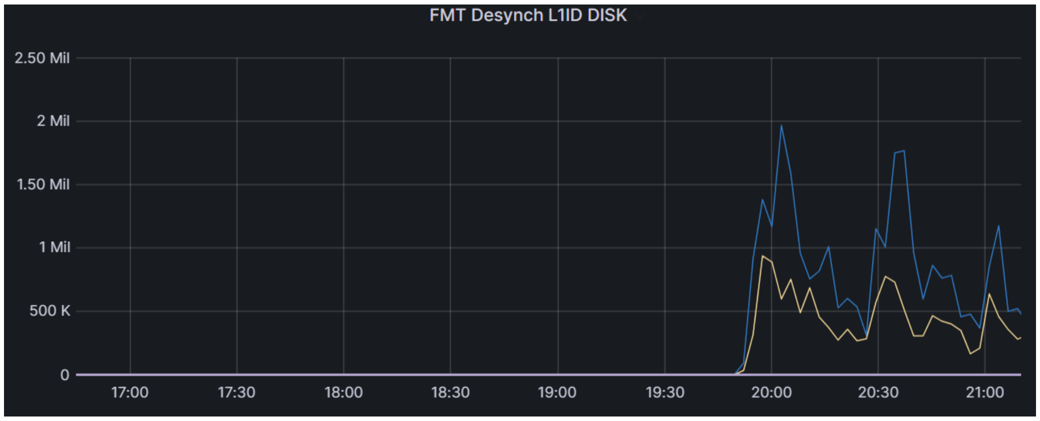


Figure 3.9: Problem during testing in the ATLAS detector - Pixel Dashboard showing two modules that got completely desynchronized.

3.10 Further modifications done

The modifications done after the initial implementation by Nico can be split into two parts. One is the implementation of the algorithm that is compatible with the 160 Mb/s readout speed. The second is an attempt to prevent the issues seen during testing in the detector in 2022. The source code can be found in Appendix C3.

160 Mb/s implementation

The initial implementation only worked for the 80 Mb/s readout speed. In the 80 Mb/s readout mode, there are two formatter links mapped to one serial link between the ROD and BC. This is because Layer-2, Disk-1 and Disk-3, (which can only run at 80 Mb/s) have more modules (MCCs) connected than the 8 links available. So, in some cases two MCCs have to share a serial link, which is handled by the BOC. In the SmartL1A forwarding mechanism, the trigger command ends up being inhibited for both modules sharing a serial link because there is no way to send triggers to them selectively.

For the 160 Mb/s mode the module to serial line mapping is different. Numerically, an even serial link i corresponds to formatter link 2*i, and odd links correspond to 2*i-1. It is a one to one mapping since there is no link sharing in layers that operate on 160 Mb/s. Table 3.4 shows the mapping of serial lines to Formatter links for both readout speeds.

The SmartL1A forwarding firmware has been modified to take this difference into account. The mapping changes based on the readout mode being used, which is defined by a ROD slave register. This has been verified to be working functionally in SR1 both using emulator and real modules.

Table 3.4: Serial link vs Formatter link mapping for both readout rates.

Serial Link	**Formatter Link - 80 Mb/s**	**Formatter Link - 160 Mb/s**
xc(0)	0,8	0
xc(1)	1,9	1
xc(2)	2, 10	4
xc(3)	3, 11	5
xc(4)	4, 12	8
xc(5)	5, 13	9
xc(6)	6, 14	12
xc(7)	7, 15	13

Potential fix for issues seen during tests in the detector

When the issue was seen during testing in the detector, it was suspected that the number of triggers to be processed (tr_tbprocessed) was stuck at a very high value. This could have been caused by an unexpected decrement that caused an underflow, or wraparound of the count, making it a large number. In an attempt to prevent this, a check has been put in place to ensure tr_tbprocessed never goes negative. The first step is to detect whether the tr_tbprocessed number was going negative. If it does, the idea is to not accept that value and do something else instead. That could have been either of these:

- Set the value to zero.
- Do not change the value.
- Update the value but ignore decrement.

Forcing the value to zero may result in there potentially being pending triggers in the system that would be ignored. Not changing the value would lose out on recording the increment of pending triggers which is not ideal. The best option would be to only increment but ignore the decrement so it cannot go negative. This condition should not happen since in a module a trigger is always sent first (increment) and then a reply is received (decrement) but this would be a safety mechanism against possible corrupted data due to something like bitflips. This has been tested in SR1 to verify that the basic functionality of the algorithm has been maintained.

Status registers to flag when the negative condition is met were added to enable further debugging of the issue in the detector if needed.

Additionally, the current implementation takes skipped triggers into consideration in the decrement. Now that it has been seen that the best performing configuration of threshold set to 15 should not have the possibility of having skipped triggers in the system, the removal of that condition would make the algorithm cleaner, more robust and less prone to errors due to potential corruption in the skipped count field. This has not been implemented at the time of writing since more testing is required, but it can be explored in the future.

3.11 Tests in the ATLAS Detector - 2023

The Pixel Operations team has resumed testing since datataking restarted after the shutdown. The firmware with the modifications to prevent underflow and implementation for 160 Mb/s was tested in the detector in a few RODs in Disk-1 (80 Mb/s) and Disk-2 (160 Mb/s). The problems that came up in the 2022 tests did not occur and things are running smoothly at the time of writing. Figure 3.10 is a plot of desynchronized events in each module in each ROD in the B-Layer during a run that indicates that RODs with the mechanism have much lower desynchronized events (the blue/white stripes) than the others. However, more testing is required in other layers and in more RODs at the same time to ensure stability and determine the overall effect of this algorithm on mitigating desynchronization in challenging datataking conditions.

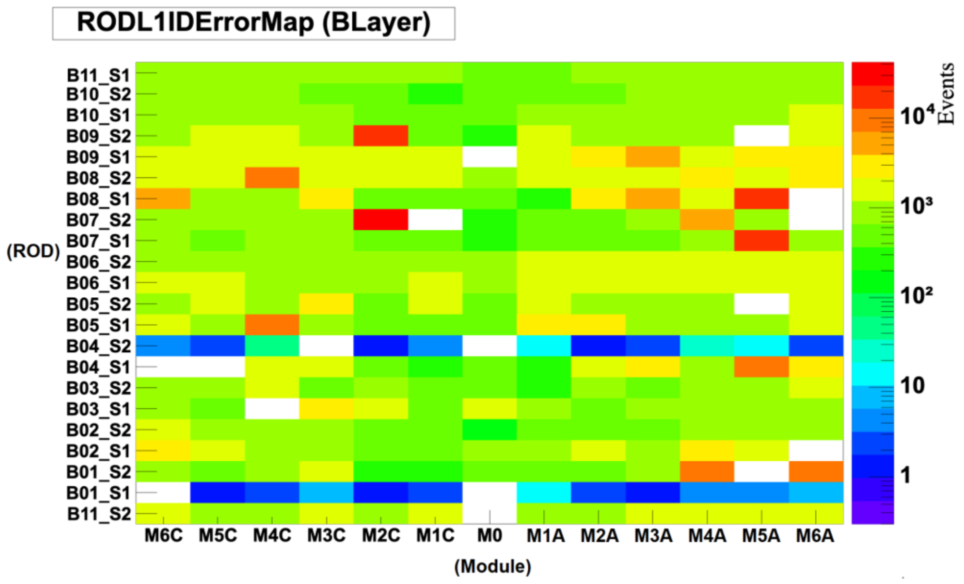


Figure 3.10: Online monitoring plot of desynch in B-Layer Modules during a run (Courtesy: Marcello Bindi). RODs B04_S2 and B01_S1 with the SmartL1A feature have a much lower number of desynched events.

3.12 Conclusion

Initial tests of the SmartL1A forwarding mechanism in the detector have shown promising results with regard to a decrease in inefficiency in the system. However, there were issues that could not be replicated in SR1, and further modifications have been made in an attempt to mitigate it. The firmware has also been updated with an implementation of the algorithm for a readout speed of 160 Mb/s. The firmware with these modifications is being tested in the detector, and no major issues have been encountered at the time of writing. Future work would entail extending the testing of the updated implementation to more RODs to check for stability and possibly implementing the changes suggested to improve robustness of the mechanism.

REFERENCES

[1] CERN Website, https://home.cern/about
[2] LHC Website, https://home.cern/science/accelerators/large-hadron-collider
[3] CERN Image Gallery, https://home.cern/resources/image/cern/views-cern-images-gallery
[4] CERN ATLAS Website, https://home.cern/science/experiments/atlas
[5] "LHC The guide", https://cds.cern.ch/record/2809109/files/CERN-Brochure-2021-004-Eng.pdf
[6] ATLAS Website- Detector & Technology, https://atlas.cern/Discover/Detector
[7] ATLAS Website - Trigger and Data Acquisition System, https://atlas.cern/Discover/Detector/Trigger-DAQ
[8] "Readout Driver Firmware Development for the ATLAS Insertable B-Layer", Shaw-Pin Chen, https://people.ece.uw.edu/hauck/publications/BingThesis.pdf
[9] "The ATLAS Level-1 Central Trigger", Mark Stockton https://cds.cern.ch/record/1300249/files/ATL-DAQ-PROC-2010-036.pdf
[10] "Insertable B-Layer Technical Design Report", https://cds.cern.ch/record/1291633/files/ATLAS-TDR-019.pdf
[11] "The Upgraded Pixel Detector of the ATLAS Experiment for Run-2 at the LHC", Mario P. Giordani, https://cds.cern.ch/record/2231012/files/ATL-INDET-PROC-2016-009.pdf
[12] CERN S-LINK Website, http://hsi.web.cern.ch/atlas/rolink/specsheet/specification.html
[13] "ATLAS Silicon ReadOut Driver (ROD) Users Manual", http://www-eng.lbl.gov/~jmjoseph/Atlas-SiROD/Manuals/usersManual-v164.pdf
[14] "Design of the ATLAS IBL Readout System", Allesandro Polini et. al., https://www.sciencedirect.com/science/article/pii/S1875389212019190
[15] "ATLAS Pixel FE Chip: Description and Failure Summary", Kevin Einsweiler, https://twiki.cern.ch/twiki/pub/Sandbox/LauraJeantySandbox/ATLASPixelFEChip_v3_0.pdf
[16] "The FE-I4B Integrated Circuit Guide", https://indico.cern.ch/event/261840/contributions/1594374/attachments/462649/641213/FE-I4B_V2.3.pdf
[17] "MCC: the Module Controller Chip for the ATLAS Pixel Detector", Roberto Beccherle et. al., https://cds.cern.ch/record/685344/files/indet-2002-002.pdf
[18] "IBL BOC Design and Firmware Manual", Marius Wensing et. al., https://indico.cern.ch/event/261840/contributions/1594374/attachments/462648/641211/IBL_BOC_Design_and_Firmware_Manual.pdf
[19] "IBL ROD BOC Manual Developer Version", Shaw-Pin Chen et. al., https://twiki.cern.ch/twiki/pub/Sandbox/LauraJeantySandbox/iblRodBocManual_v1.2.3.pdf
[20] ATLAS Pixel GitLab Wiki - Calibration Procedure, https://gitlab.cern.ch/atlas-pixel/daq/atlaspixeldaq/-/wikis/Calibration%20procedure#calibration-procedure

[21] “Observation of ROD histogrammer issues on noise mask (slide deck)”, Marcello Bindi, https://indico.cern.ch/event/1093108/contributions/4597061/attachments/2338003/3985629/NoiseMaskHistogrammer.pdf
[22] “SmartL1A forwarding validation and tests”, Pixel Operations, https://github.com/uw-acme/acme-lab-documentation/blob/main/lhc/Pixel_IBL/Smart_L1A/task_description/SmartL1IDTests_v1.pdf
[23] “ATLAS Pixel Detector and readout upgrades for the improved LHC performance”, Nico Giangiacomi, https://cds.cern.ch/record/2684079/files/CERN-THESIS-2018-437.pdf
[24] “Firmware Updates (slide deck)”, Nico Giangiacomi, https://indico.cern.ch/event/803422/contributions/3340933/attachments/1895044/3126212/Firware_Update.pdf
[25] “ROD Firmware Error Reporting”, Nico Giangiacomi et. al., https://github.com/uw-acme/acme-lab-documentation/blob/main/lhc/Pixel_IBL/DataFormat/RODByteStreamErrors.pdf
[26] Pixel Dashboard Website, https://atlasop.cern.ch/tdaq/pbeastDashboard/d/000000025/pixel-dashboard?orgId=1

ACKNOWLEDGEMENT

This thesis is based upon work supported by the National Science Foundation under Grant No. PHY-2110963, US ATLAS Maintenance and Operation funding and subsistence support from the CERN ATLAS Pixel operation.

First and foremost, I would like to thank my advisors, Prof. Scott Hauck and Prof. Shih-Chieh Hsu, for being amazing mentors throughout my time at UW. I have learnt a lot from them, ranging from technical aspects of digital design, about the LHC, managing a project and much more. I am incredibly grateful for the opportunity to work on these projects which would not have been possible without my advisors.

Special thanks Marcello Bindi, Martin Kocian and Tobias Bisanz for their guidance during my time working with Pixel. They have been instrumental in helping me understand the base of my work and the infrastructure used. I would also like to thank the rest of the Pixel Operations Team, including Kerstin Lantzsch, Vinicius Franco Lima, Andrea Sciandra, James Philip Iddon and Chris Scheulen for their help throughout, and making me feel at home during my time at CERN. My time at CERN is one of my most cherished memories.

I would like to thank Gabriele Balbi for getting me up to speed on ROD firmware development. I would also like to express my gratitude to Nico Giangiacomi for designing the Smart L1A Forwarding mechanism and generously explaining his work so I could restart the project.

Thank you to Geoff Jones for his insights on FPGA development and VHDL, and Timon Heim for his wealth of knowledge on ATLAS. They have been great resources during my time at ACME Lab. I would also like to thank all of my ACME colleagues for their inputs and support.

Finally, I would like to thank my family and friends for their encouragement without which I could not have completed my graduate degree.

APPENDIX

APPENDIX A: SR1 TUTORIALS

During my time working with the SR1 setup, I created notes and tutorials on how to use the setup and the features associated with firmware. These can be found in the ACME documentation repo: https://github.com/uw-acme/acme-lab-documentation/tree/main/lhc/Pixel_IBL/SR1_Tutorials

APPENDIX B: HISTOGRAMMER

RODSlave Firmware repo, branch- Sanjukta_calibration_final: https://gitlab.cern.ch/atlas-pixel/daq/pixelrod_firmware/RodSlave/-/blob/Sanjukta_calibration_final/Firmware/Pixel/src/calibration/router/router_top.vhd#L385

APPENDIX C: SMART L1A FORWARDING

1. Nico Giangiacomi's slides during initial development

https://github.com/uw-acme/acme-lab-documentation/tree/main/lhc/Pixel_IBL/Smart_L1A/related_material

2. DAQ Software repo (atlaspixeldaq)

Added writePendtrighthreshregVme and modified Occupancy_Desynch_counters_Read files.

Branch- Sanjukta_SmartL1: https://gitlab.cern.ch/atlas-pixel/daq/atlaspixeldaq/-/tree/Sanjukta_SmartL1/RodDaq/IblUtils/Vme/src
Also note changes in this file for compiling: https://gitlab.cern.ch/atlas-pixel/daq/atlaspixeldaq/-/blob/Sanjukta_SmartL1/RodDaq/cmake_tdaq.txt#L288

Copying over these changes into a local setup with the updated TDAQ version is recommended rather than cloning this branch directly.

3. ROD Slave Firmware

Current implementation, Branch- Sanjukta_Smart_L1_forwarding_160:

- Repo- https://gitlab.cern.ch/atlas-pixel/daq/pixelrod_firmware/RodSlave/-/tree/Sanjukta_Smart_L1_forwarding_160/Firmware/Pixel
- Gitlab pipeline - https://gitlab.cern.ch/atlas-pixel/daq/pixelrod_firmware/RodSlave/-/pipelines/5357699

Nico's original implementation, Branch- Firmware_SmartL1IDAlgorithm

- Repo – https://gitlab.cern.ch/atlas-pixel/daq/pixelrod_firmware/RodSlave/-/tree/Firmware_SmartL1IDAlgorithm/Firmware/Pixel

4. Testing repos

- SmartL1_testing – https://gitlab.cern.ch/sroychou/SmartL1_testing/-/tree/master. Should be cloned in an lxplus machine. Run scripts from the sbc of the corresponding crate.
- Python notebook for plotting – https://gitlab.cern.ch/sroychou/plotting. Plots cumulative events and computes percentages of inefficiency from the extracted dump file.

5. Useful data formats

Important fields that can help identify these types of events have been described below.

Skipped trigger event trailer data format:

Bits	**31 - 28**	**27 - 24**	**23 - 20**	**19 - 16**	**15 - 12**	**11 - 8**	**7 - 4**	**3 - 0**
Trailer	4	0	0	link	A	C	C	A

This is slightly different from the trailer format in the original datataking firmware. The link number was added for clarity during the initial development of the SmartL1A forwarding algorithm.

ROD vetoed SmartL1A event header and trailer data format:

Bits	**31 - 28**	**27 - 24**	**23 - 20**	**19 - 16**	**15 - 12**	**11 - 8**	**7 - 4**	**3 - 0**
Header	2	1	X	X	B	A	A	D
Trailer	4	0	8	0	link	B	A	D

Timeout event trailer data format:

Bits	**31 - 28**	**27 - 24**	**23 - 20**	**19 - 16**	**15 - 12**	**11 - 8**	**7 - 4**	**3 - 0**
Trailer	4	1	0	0	link	B	A	D