VERIFICATION AND ENHANCEMENT OF THE AUTORECOVERY BLOCK AND STAGE-1 BEAM TEST INTEGRATION FOR CMS HGCAL DETECTOR

by

Berkay Yazıcı

B.S., Electrical and Electronics Engineering, MEF University, 2022

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Electrical and Electronics Engineering
Boğaziçi University

2025

ACKNOWLEDGEMENTS

This work was carried out within the scope of the TENMAK project titled "Design of Electronic Systems in the HGCAL Detector, Study of SUEP and the HF Online RADDAM System Improvement" and the TÜBİTAK 2232-118C223 project, "AdvancedTCA Development for High-Energy Physics and Other Applications". I would like to express my sincere gratitude to my supervisors, Assist. Prof. Faik Başkaya and Assist. Prof. Bora Akgün, for their invaluable guidance, encouragement, and support throughout this study. I am also grateful to Mehmet Alp Şarkışla and Rauf Ozan Akpınar for their collaboration, support, and friendship during our time at CERN, Geneva.

ABSTRACT

VERIFICATION AND ENHANCEMENT OF THE AUTORECOVERY BLOCK AND STAGE-1 BEAM TEST INTEGRATION FOR CMS HGCAL DETECTOR

The Compact Muon Solenoid (CMS) detector at the Large Hadron Collider (LHC) at The European Organization for Nuclear Research (CERN) is undergoing a Phase-2 upgrade, driven by the need to ensure reliable performance under harsh radiation levels and to handle the large amount of data due to a greater number of proton-proton interactions, both brought by the increased luminosity of the High-Luminosity LHC (HL-LHC). The back-end (BE) electronics and data acquisition (DAQ) firmware are critical components of this upgrade. The BE electronics consists of field-programmable gate arrays (FPGAs), and the DAQ system is fully implemented on these programmable devices. This architecture allows for high-speed, flexible, and configurable processing of the large data streams generated during detector operation. During beam crossings, front-end (FE) application-specific integrated circuits (ASIC) that read data from on-detector sensors can become misaligned or fall out of synchronization. The Autorecovery Block is a dedicated firmware module designed to monitor error conditions and generate fast command signals to re-tune the FE electronics when necessary, thereby improving data integrity and system resilience. This thesis presents the verification, enhancement, and integration of the Autorecovery Block within the existing firmware framework. Comprehensive hardware tests were conducted using software emulators to validate its functionality. The results demonstrate that the Autorecovery Block can correctly generate the expected signals under the tested conditions, providing confidence in its readiness for future integration.

ÖZET

CMS HGCAL DEDEKTÖRÜ İÇİN AUTORECOVERY BLOĞU'NUN DOĞRULANMASI VE GELİŞTİRİLMESİ İLE AŞAMA 1 DEMET TESTİNE ENTEGRASYONU

CERN'deki Büyük Hadron Çarpıştırıcısı'nda (LHC) CMS dedektörü, artan ışınım seviyelerine karşı güvenilir performansı sürdürmek ve HL-LHC ile artan parlaklığın getirdiği daha fazla proton-proton çarpışması nedeniyle oluşan büyük veri hacmini işlemek için gerekli olan Faz-2 yükseltmesinden geçmektedir. Bu yükseltmede arka-uç elektroniği ve veri toplama donanım yazılımı kritik bileşenlerdir. Arka-uç elektroniği, alan programlanabilir kapı dizileri (FPGA) üzerine inşa edilmiştir ve veri toplama sistemi bu FPGA'ler üzerinde gerçekleştirilmiştir. Bu mimari, dedektör çalışması sırasında üretilen yüksek hacimli veri akışlarını yüksek hızda, esnek ve yapılandırılabilir bir şekilde işleme imkânı sunar. Işın geçişleri sırasında, dedektör üzerindeki sensörlerden veri okuyan ön-uç uygulamaya özel tümleşik devreler (ASIC), senkronizasyondan sapabilir veya uyumsuz hale gelebilir. Autorecovery Bloğu, bu tür hata koşullarını izlemek ve gerektiğinde ön-uç elektroniğini yeniden ayarlamak için hızlı komut sinyalleri üreten özel bir donanım yazılımı modülüdür; böylece veri bütünlüğü ve sistem dayanıklılığı iyileştirilir. Bu tezde, Autorecovery Bloğu'nun doğrulaması, iyileştirilmesi ve mevcut donanım yazılımı çerçevesine entegrasyonu sunulmaktadır. Yazılım emülatörleri kullanılarak gerçekleştirilen kapsamlı donanım testleri, blokun işlevselliğini doğrulamıştır. Elde edilen sonuçlar, Autorecovery Bloğu'nun test edilen koşullar altında beklenen sinyalleri doğru şekilde üretebildiğini ve gelecekteki entegrasyon için hazır olduğunu göstermektedir.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . iii
ABSTRACT . . . iv
ÖZET . . . v
LIST OF FIGURES . . . x
LIST OF TABLES . . . xvii
LIST OF SYMBOLS . . . xviii
LIST OF ACRONYMS/ABBREVIATIONS . . . xix
1. INTRODUCTION . . . 1
2. CMS DETECTOR AND UPGRADE FOR HL-LHC . . . 4
2.1. Large Hadron Collider . . . 4
2.1.1. Design and Physical Principles . . . 4
2.1.1.1. Injector Chain and Proton Acceleration . . . 5
2.1.1.2. Magnets and Beam Dynamics . . . 6
2.1.1.3. RF Cavities . . . 7
2.2. Overview of the Compact Muon Solenoid Detector . . . 8
2.2.1. Coordinate System . . . 9
2.2.2. Tracker . . . 9
2.2.3. Electromagnetic Calorimeter . . . 10
2.2.4. Hadron Calorimeter . . . 11
2.2.5. Magnet . . . 12
2.2.6. Muon System . . . 13
2.3. HGCAL Upgrade . . . 14
2.3.1. HL-LHC and Motivation for HGCAL . . . 14
2.3.2. HGCAL Overview . . . 15
2.3.3. Active Elements . . . 17
2.3.3.1. Silicon Sensors . . . 17
2.3.3.2. Silicon Modules . . . 17
2.3.3.3. Plastic Scintillators and Scintillator Tile-modules . . . 18

2.3.4. Electronics . . . 19
2.3.4.1. Front-end Electronics . . . 19
2.3.4.2. Back-end Electronics . . . 21
3. HGCAL BACK-END DATA ACQUISITION SYSTEM . . . 22
3.1. Uplink Data Path (Front-End to Back-End) . . . 22
3.2. Fast Control Commands and Autorecovery Block . . . 23
3.3. Capture Block . . . 25
3.3.1. Packet Detection . . . 26
3.3.2. Main Buffer . . . 27
3.3.3. Packet Matching and Output Buffer . . . 28
3.3.4. The Event Builder State Machine . . . 29
4. BACK-END FIRMWARE TOOLS AND FRAMEWORKS . . . 33
4.1. Serenity Board . . . 33
4.2. IPbus Protocol . . . 36
4.2.1. IPbus Suite . . . 37
4.3. EMP Framework . . . 38
4.4. Firmware Build and Test Workflow . . . 39
5. AUTORECOVERY BLOCK AND BEHAVIORAL VERIFICATION . . . 41
5.1. Autorecovery Block Architecture . . . 41
5.1.1. Thresholds, Priorities, and Fast Commands . . . 41
5.1.2. Declaration File . . . 43
5.1.3. Outline of Internal Processes . . . 43
5.1.3.1. Arbiter . . . 43
5.1.3.2. Router Submodule . . . 46
5.2. Configurable Routing Map . . . 49
5.2.1. Implementation of Routing Control Registers . . . 49
5.3. Cocotb-Based Verification Environment . . . 51
5.3.1. Corner Cases . . . 52
5.3.2. Verification Structure . . . 53
5.3.3. Test Results . . . 55
5.3.4. Coverage Results . . . 59

6. TESTING THE UPDATED AUTORECOVERY WITH EMP ON SERENITY 63
6.1. EMP Payload Integration 63
6.1.1. IPbus Register Mapping and Packaging Project 67
6.1.2. Creating lpGBT Input File for Autorecovery Block Test 70
6.1.3. Register Read/Writes 71
6.1.4. Test Workflow 72
6.1.5. Test Results 73
6.2. FIFO Upgrade 76
6.2.1. Implementation 76
6.2.1.1. FIFO Read Logic 78
6.2.1.2. Input Sampling Logic 79
6.2.2. Vivado Testbench 81
6.2.3. EMP Integration 82
6.2.3.1. IPbus Register Mapping 83
6.2.4. FIFO Read Scripts 84
6.2.5. Test Procedure and Results 84
7. STAGE-1 BEAM TEST INTEGRATION 89
7.1. Beam Test Platform Overview and TCDS2 Emulation Software 89
7.2. Integrating the Enhanced Autorecovery Block 90
7.2.1. Firmware Merge with IPbus 90
7.2.2. New Address Tables 92
7.3. Packet Structures 93
7.4. LpGBT Input Generation and Firmware Tests with TCDS2 Emulator . 93
7.4.1. Good Packets 96
7.4.2. Event-ID Mismatch 101
7.4.3. Bunch Crossing Mismatch 104
7.4.4. Timeout 108
8. CONCLUSION 112
REFERENCES 113
APPENDIX A: SUPPLEMENTAL VHDL CODES 117
A.1. Event Builder State Machine VHDL Code 117

A.2. Autorecovery Declaration VHDL Code 127
APPENDIX B: PYTHON SCRIPTS 132
B.1. Routing Map Control and Flag Error Counter Read Script 132
B.2. FIFO Data Capture Start Script 138
B.3. FIFO Read to CSV Script 139

LIST OF FIGURES

Figure 2.1. CERN's Accelerator Complex. 5

Figure 2.2. LHC Injection Chain. 6

Figure 2.3. LHC RF System. 7

Figure 2.4. CMS detector cutaway diagram. 8

Figure 2.5. CMS coordinate system. [10]. 9

Figure 2.6. CMS silicon pixel detector. 10

Figure 2.7. Layout of the CMS electromagnetic calorimeter. 11

Figure 2.8. A cross section view of the CMS HCAL. 12

Figure 2.9. The layout of one quarter of the CMS muon system. 13

Figure 2.10. Timeline of LHC and HL-LHC operations and upgrades. 15

Figure 2.11. Longitudinal cross section of HGCAL endcap calorimeter. 16

Figure 2.12. Layers of silicon-based HGCAL modules. 18

Figure 2.13. CMS HGCAL FE chain. 21

Figure 3.1. High level BE DAQ FPGA firmware blocks. 23

Figure 3.2. A simplified diagram of the fast control command blocks. 24

Figure 3.3. Fast Command Table. 24

Figure 3.4. Capture Block architecture showing the interface with lpGBT input, ECON-D deserialization, packet detection, event building, and output buffering mechanisms. 25

Figure 3.5. Format of an ECON-D packet. 27

Figure 3.6. Capture Block event header format highlighting two primary header fields: the L1A Timestamp (combining the Bunch Crossing (BC), Event Counter (EC), and Orbit Counter (OC)) and the 3-bit ECON-D Packet Status fields. 29

Figure 3.7. FSM diagram of the Event Builder. 30

Figure 4.1. Connectivity of the Serenity board. 35

Figure 5.1. Generic interface of the Autorecovery entity. 42

Figure 5.2. Connection of routing map signals and IPbus control registers. . . 51

Figure 5.3. Cocotb interface. 52

Figure 5.4. Waveform demonstrating the priority arbitration between ECON-D 2 and ECON-D 3 routed to FCC2. Despite the `timeout` flag having higher priority over `no_idle`, ECON-D 2's fast command `0xA9` is seen first due to its ECON-D-level priority. 56

Figure 5.5. Simulation waveform showing flag-level prioritization in `ECON-D 1`. Both `econd_rr` and `timeout` flags reach their respective thresholds simultaneously, yet the fast command corresponding to the higher-priority `econd_rr` (`0xD1`) is issued. 57

Figure 5.6. Waveform showing the effect of key control signals on the Autorecovery Block logic. Assertion of `disable_flags_i` halts fast command generation for selected `ECON-D` channels; global `reset` clears counters and outputs; `is_capture_block_valid_array_i` suppresses counting for invalid Capture Blocks. 58

Figure 5.7. Transcript window output. Assertions compare output fast commands against expected values. 59

Figure 5.8. Code coverage summary from Questa SIM for the Autorecovery Block. All statements, branches, and conditions were exercised during simulation, resulting in 100% total coverage. 60

Figure 5.9. FEC report generated by Questa SIM for the Autorecovery Block. Each boolean sub-expression within compound conditions was fully exercised, confirming exhaustive evaluation. 61

Figure 5.10. Functional coverage summary of the Autorecovery Block. All user-defined coverpoint bins were exercised at least once during the simulation. 62

Figure 6.1. Autorecovery Block IPbus register address table. 69

Figure 6.2. Example of an lpGBT input test file. The file consists of 1024 frames emulating detector signals with control and data fields. . . 71

Figure 6.3. FCC test results following Autorecovery Block reset at Frame 4. The diagram displays channel outputs, associated ECON-D modules, and their counter thresholds. Channel 5 behavior illustrates output prioritization among ECON-D 6 and ECON-D 7, with ECON-D 5 disabled. The valid bit and fast command structure are shown alongside real-time error flag status. 75

Figure 6.4. Configuration and interface of the FIFO module used in the design. 77

Figure 6.5. Write enable signal generated in the Router submodule. 77

Figure 6.6. Connection between the Router submodule and the FIFO. The fast command array is fed into the FIFO's data input, while the valid signal controls the FIFO's write enable. This ensures that only valid fast commands are written into the FIFO at the moment they are formed. 78

Figure 6.7. FIFO read process. 79

Figure 6.8. Input sampling logic implemented to control FIFO loading. 80

Figure 6.9. Simulation results verifying the functionality of the Autorecovery Block with the integrated FIFO. 81

Figure 6.10. IPbus register mapping for the FIFO-enhanced Autorecovery Block. 82

Figure 6.11. Updated IPbus address map used in the Autorecovery Block. The figure consolidates the complete register layout, including control and disable signals, ECON-D interface registers, and the FIFO-related control, status, and data registers. 83

Figure 6.12. Snippet of the lpGBT input file for Fast Command FIFO test. . . 86

Figure 6.13. CSV output generated by the FIFO reader Python script for FCCs. The data reflects the logging of fast commands triggered by the error flags. 87

Figure 7.1. IPbus register fabric. 91

Figure 7.2. Capture Block IPbus configuration registers. 91

Figure 7.3. Autorecovery Block IPbus control and status registers. 92

Figure 7.4. Flag thresholds and priorities of Autorecovery Block used in Stage-1 firmware tests. 95

Figure 7.5. Snippet of input lpGBT data file showing the configured 64-bit header and payload data used in the simulation for "Good Packets". 97

Figure 7.6. TCDS2 YAML configuration used to send fast commands (ECR and L1A) at predefined BX positions. 98

Figure 7.7. Snippet of the Capture Block output showing valid ECON-D data with correct BC and EC fields, and confirming a "Good Packet" status for all connected ECON-Ds. 99

Figure 7.8. FIFO status and output registers indicating no fast command activity, consistent with error-free ECON-D input. 100

Figure 7.9. Snippet of the CSV output generated by the updated FIFO reader script, showing real-time FIFO status and fast command entries. . 100

Figure 7.10. Snippet of the configured 64-bit `lpGBT` input header corresponding to BX = 29 and Event-ID = 2. 101

Figure 7.11. Modified TCDS2 emulator YAML file without the ECR command, leading to Event-ID mismatch. 102

Figure 7.12. FIFO status output showing full flags asserted for FCCs 1, 2, 3, 4, and 8 with correct fast command outputs. 103

Figure 7.13. Snippet of the CSV output of FIFO readout script confirming expected FCCs and fast command value for Event-ID mismatch. . . 103

Figure 7.14. Snippet of the lpGBT input file containing both Event-ID mismatch (BX=29, Event number=2) and BX mismatch (BX=31, Event number=2) packet configurations used in the test scenario. . 105

Figure 7.15. TCDS2 emulator YAML configuration file set to send ECR and L1A commands at BX=25, resulting in actual command dispatch at BX=29. 106

Figure 7.16. Terminal snapshot displaying the Fast Command FIFO outputs, clearly indicating FCC channel assignments and corresponding fast commands triggered due to BX mismatch (`0x95`) and Event-ID mismatch (`0xd2`). 107

Figure 7.17. CSV log file generated by the Python script, documenting FIFO status, data counts, and output data, confirming the expected fast command entries and FIFO states during the test execution. . . . 107

Figure 7.18. The lpGBT input file configuration used in the timeout scenario. Headers with deliberately induced CRC errors for ECON-D 1–4 and Event-ID mismatch headers for ECON-D 5–7 are indicated. . 108

Figure 7.19. The lpGBT input file configuration used in the timeout scenario. Headers with deliberately induced CRC errors for ECON-D 1–4 and Event-ID mismatch headers for ECON-D 5–7 are indicated. . 109

Figure 7.20. Terminal snapshot of FIFO register outputs showing correct FCC allocations for the Event-ID mismatch (`0xd2`) and Timeout (`0x1d`) conditions. 110

Figure 7.21. CSV file output generated by the FIFO reading Python script, providing timestamped status and data from FIFO registers confirming the expected fast command responses. 111

LIST OF TABLES

Table 3.1. Description of the 3-bit ECON-D status flags. 29

Table 5.1. Error flag informations for verification. 54

Table 5.2. Routing map for verification. 55

Table 6.1. Flag information EMP test. 74

Table 6.2. Routing map for EMP test. 74

Table 6.3. Routing map for Fast Command FIFO test. 86

Table 7.1. e-link to ECON-D connections for Capture Blocks 1 and 2. 94

Table 7.2. Routing map for Stage-1 test. 96

LIST OF SYMBOLS

E_T	Transverse Energy
L	Integrated Luminosity
p_T	Transverse Momentum
X_0	Radiation Length
η	Pseudorapidity
λ	Interaction Length
$\mathcal{L}$	Instantaneous Luminosity

LIST OF ACRONYMS/ABBREVIATIONS

ALICE	A Large Ion Collider Experiment
APD	Avalanche Photodiode
ASIC	Application-Specific Integrated Circuit
ATCA	Advanced Telecommunications Computing Architecture
ATLAS	A Toroidal LHC ApparatuS
BE	Back-End
BX	Bunch Crossing
CE-E	Electromagnetic Endcap Calorimeter
CE-H	Hadronic Endcap Calorimeter
CMS	Compact Muon Solenoid
CRC	Cyclic Redundancy Check
CSC	Cathode Strip Chamber
DAQ	Data Acquisition
DTH	DAQ and Timing Hub
DUT	Device Under Test
ECAL	Electromagnetic Calorimeter
ECR	Event Counter Reset
EMP	Extensible Modular Processor
FEC	Focused Expression Coverage
FE	Front-End
FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
HB	HCAL Barrel
HCAL	Hadron Calorimeter
HE	HCAL Endcap
HEP	High Energy Physics
HF	HCAL Forward

HGCAL	High Granularity Calorimeter
HL-LHC	High-Luminosity Large Hadron Collider
HLT	High-Level Trigger
HO	HCAL Outer
IPMC	Intelligent Platform Management Controller
L1A	Level-1 Accept
LHC	Large Hadron Collider
LHCb	Large Hadron Collider beauty
LINAC	Linear Accelerator
lpGBT	Low Power Gigabit Transceiver
LS	Long Shutdown
LS1	Long Shutdown 1
LS2	Long Shutdown 2
LS3	Long Shutdown 3
MicroTCA	Micro Telecommunications Computing Architecture
PCB	Printed Circuit Board
PCIe	Peripheral Component Interconnect Express
PS	Proton Synchrotron
PSB	Proton Synchrotron Booster
RF	Radio Frequency
ROC	Readout Chip
RMW	Read-Modify-Write
RPC	Resistive Plate Chamber
SAR ADC	Successive Approximation Analog-to-Digital Converter
SCA	Slow Control Adapter
SiPM	Silicon Photomultiplier
SM	Standard Model
SoC	System on Chip
SPS	Super Proton Synchrotron
SS	Stainless Steel
TCDS2	Phase-2 Timing and Control Distribution System

TDC	Time-to-Digital Converter
TTC	Timing, Trigger and Control
TPG	Trigger Primitive Generator
uHAL	Micro Hardware Access Library
URAM	Ultra RAM
VHPI	VHDL Procedural Interface
VPI	Verilog Procedural Interface
VTRx+	Versatile Transceiver Plus

1. INTRODUCTION

CERN is an intergovernmental research organization operating the largest particle physics laboratory in the world, located in Meyrin, Geneva, along the France–Switzerland border. CERN hosts the LHC, the highest-energy particle collider ever built. The primary objective of the LHC is to accelerate protons or heavy ions in two high-energy beams traveling in opposite directions at nearly the speed of light. Collisions occur at four intersection points, each instrumented with sophisticated particle detectors designed to investigate fundamental physics phenomena.

High-energy interactions, such as lead-lead, proton-lead, and proton-proton collisions, enable the observation of previously unexplored physics processes. Notably, in July 2012, the LHC facilitated the discovery of the Higgs boson, a cornerstone of the Standard Model (SM). In 2015, the LHC achieved proton-proton collisions at a center-of-mass energy of 13 TeV. Despite this significant milestone, exploring rare events requires higher collision luminosity. To address this need, the HL-LHC is being developed to increase integrated luminosity by an order of magnitude, enhancing the discovery potential of rare processes.

The CMS [1] detector is one of the two primary general-purpose experiments operating at the LHC. Its physics program spans a broad spectrum, including studies of the Higgs boson, searches for extra dimensions, and exploration of dark matter candidates. To fully exploit the physics opportunities offered by the HL-LHC, CMS is undergoing an extensive Phase-2 upgrade program. Among these upgrades, the CMS endcap calorimeters will be replaced with the High-Granularity Calorimeter (HGCAL). This upgrade is essential for maintaining high performance under significantly increased radiation exposure and managing large data volumes generated from high-luminosity collisions.

The HGCAL BE DAQ system aggregates and processes data transmitted from the FE electronics via approximately 9000 optical fibers, each operating at 10 Gb/s, at an event acquisition rate of 750 kHz. This corresponds to an aggregate bandwidth requirement of approximately 15 Tb/s, with an average event size of 2.5 MB. Following event building, processed data are transmitted to the central CMS DAQ through approximately 1100 optical fibers, each running at 24 Gb/s. The HGCAL BE DAQ is implemented on 96 Serenity The Advanced Telecommunications Computing Architecture (ATCA) [2] boards, each handling an average data rate of 160 Gb/s at an operational frequency of 320 MHz.

The firmware architecture presented in this thesis, the Autorecovery Block, is a dedicated module integrated within the Capture Block of the BE DAQ system. Its primary function is to process error flags generated by the Capture Block's Event Builder, which arise from discrepancies detected during the comparison of FE concentrator ASIC (ECON-D) data with Level-1 Accept (L1A) trigger information; upon detecting such mismatches, the Autorecovery Block generates and routes fast command signals to resynchronize or reconfigure the FE electronics as needed. The central contributions of this thesis are the enhancement, verification, and comprehensive hardware testing of the Autorecovery Block. Specifically, the enhanced architecture is adapted to integrate seamlessly with the extensible modular (data) processor (EMP) framework [3] and IPBB firmware tools, enabling reliable monitoring, verification, and validation through targeted tests on Serenity hardware.

The thesis is structured as follows. Chapter 2 provides an overview of the LHC, describes the CMS detector, and introduces the HGCAL upgrade, covering motivations, architecture, and core components. Chapter 3 outlines the architecture and operational features of the HGCAL BE DAQ System. Chapter 4 discusses the firmware tools and frameworks utilized for testing and hardware integration. The verification process and results of the enhanced Autorecovery Block are presented in Chapter 5, while Chapter 6 provides details of hardware tests conducted using the EMP framework on the Serenity ATCA platform. Finally, Chapter 7 covers the integration and validation of the

enhanced Autorecovery Block within the Stage-1 beam-test firmware, evaluated using Phase-2 Timing and Control Distribution System (TCDS2) emulator in preparation for actual beam tests.

2. CMS DETECTOR AND UPGRADE FOR HL-LHC

2.1. Large Hadron Collider

In operation since 2009, the world's largest and highest-energy particle accelerator is designed to collide protons (and occasionally heavy ions) in the center-of-mass frame at energies as high as 13-14 TeV, and these highly energetic collisions are studied at CERN to discover the fundamental constituents of matter, test the predictions of the SM and investigate physical phenomena beyond current theories. The discovery of the Higgs boson in 2012 confirms the mechanism responsible for electroweak symmetry breaking and mass formation, and is one of the most important discoveries of the LHC. CERN and its collaborators are working on an upgrade collectively known as the HL-LHC. Due to the machine's extreme complexity and optimization, such an upgrade requires careful consideration. The necessary improvements will take about a decade to prototype, test, and realize. A number of significant cutting-edge technologies that present exceptional scientific challenges will be necessary for the HL-LHC, a unique machine configuration.

2.1.1. Design and Physical Principles

The 27-kilometer-long LHC is located 100 meters underground in a tunnel once used by the Large Electron-Positron Collider. It is not a perfect ring, but it consists of eight arcs and eight long straight sections. It has four main collision points, each of which is home to one major experiment, the accelerator complex is shown in Figure 2.1.

Two of these experiments, CMS and A Toroidal LHC ApparatuS (ATLAS) [4], are designed for large-scale particle physics research, while A Large Ion Collider Experiment (ALICE) [5] is used for heavy ion collisions and Large Hadron Collider beauty (LHCb) [6] for the study of small separations between matter and antimatter. Luminosity is a critical parameter in particle accelerator experiments as it determines

the number of particle collisions occurring over a specified period. Higher luminosity corresponds to an increased collision rate, enabling greater data collection and thereby improving the observation potential for rare processes and elusive particles. A more detailed explanation on luminosity is presented in Section 4.1. Magnets, radio frequency (RF) systems and beam optics work together to maximize this luminosity as part of the LHC's basic design.

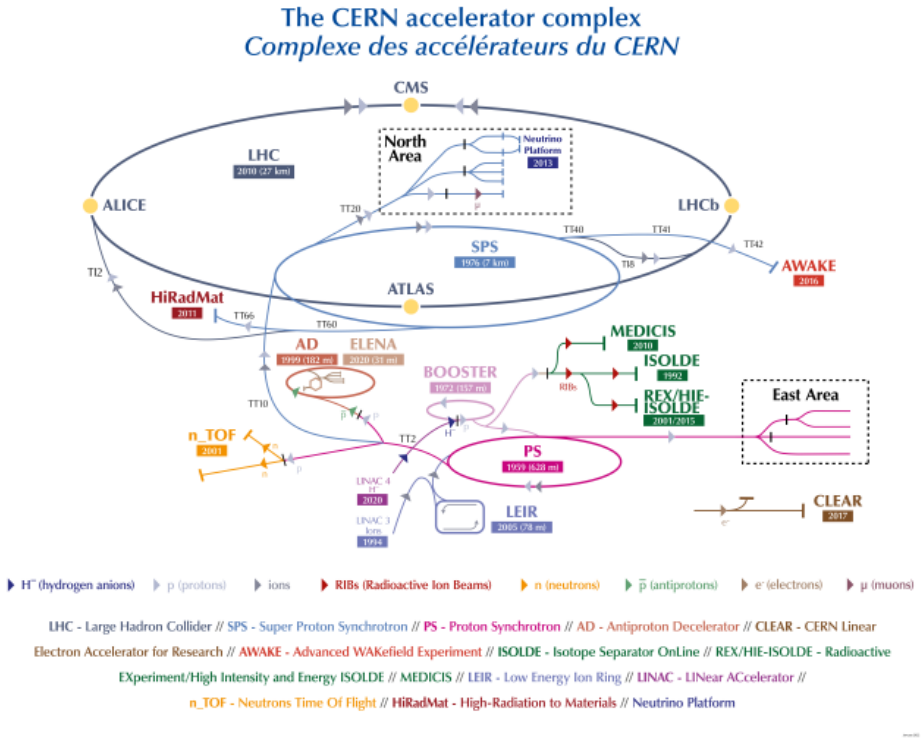


Figure 2.1. CERN's Accelerator Complex.

2.1.1.1. Injector Chain and Proton Acceleration. At the LHC, protons start in a linear accelerator (LINAC) and then pass through a series of synchrotrons called the LHC injector chain shown in Figure 2.2. LINAC first accelerates protons to several MeV. The beam is then transferred to the Proton Synchrotron Booster (PSB). This raises proton's energy to the GeV range. The beam energy is then increased to a few tens of GeV in the Proton Synchrotron (PS), and to around 450 GeV in the Super Proton Synchrotron (SPS). After reaching this level, in order to be accelerated to several TeV, the protons are injected into the LHC's main ring [7].

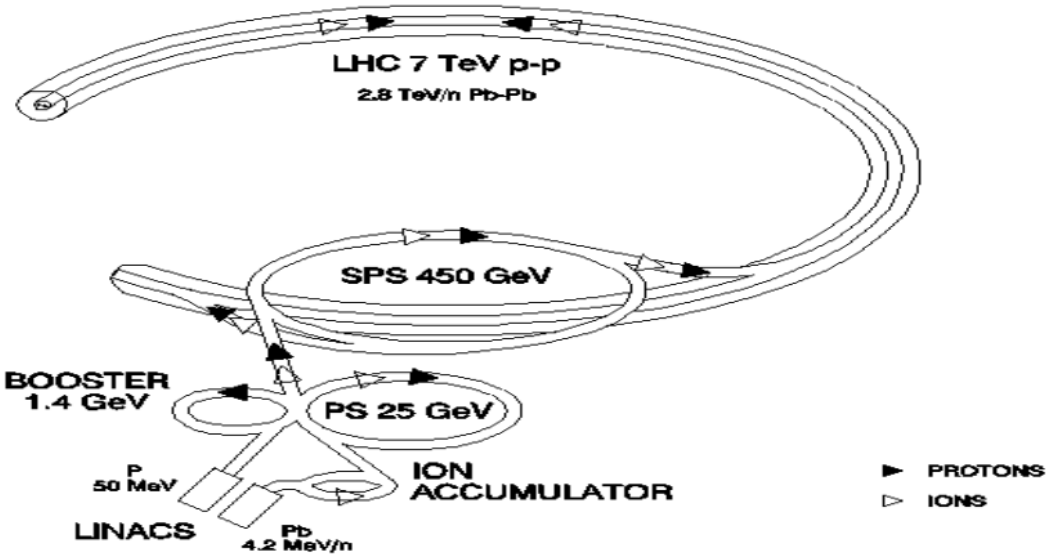


Figure 2.2. LHC Injection Chain.

Within the LHC ring, the beams are divided into discontinuous groups called *bunches*, each containing about 10^{11} protons. Depending on the operating configuration, these bunches can be separated by intervals of 25 or 50 ns. Superconducting magnets bend and focus the protons until they collide head-on at specific experimental stations, circulating the beams in the LHC about 11,000 times per second.

2.1.1.2. Magnets and Beam Dynamics. The magnet system consists of more than a thousand double-aperture dipole magnets, each about 15 meters long, cooled to $-271\,^{\circ}\mathrm{C}$ using superfluid helium. These dipoles bend the beam and provide magnetic fields of about 8.3 T at nominal currents to keep it on a circular trajectory in a fixed-radius tunnel [8]. The quadrupole magnets prevent the beam from hitting the vacuum tube walls or spreading out by "focusing" it in two transverse dimensions. To squeeze the beam either vertically or horizontally, they contain four magnetic poles symmetrically placed around the beam pipe. The FODO lattice, also known as alternating focusing and defocusing quadrupoles, is the basis for the arrangement of magnets around the LHC ring. Each focusing or defocusing quadrupole changes the

beam envelope so that the net effect is a stable closed trajectory over a full circuit.

The dipole magnets are also fitted with sextupole, octupole, and decapole magnets to compensate for minor flaws in the magnetic field at the dipoles' extremities.

2.1.1.3. RF Cavities. RF cavities control longitudinal dynamics at about 400 MHz. They continuously accelerate protons with synchronized oscillations shown in Figure 2.3. This process continues for each individual beam, from the injection energy up to 6.5–7 TeV per proton [9]. Each bunch remains coherent since the RF cavities provide carefully timed energy spikes and correct for small phase shifts in the beam.

Maintaining stable beams at such high energies requires the electric field of the RF and the timing of the circulating protons to be precisely in phase. If the phase shifts, the bunches can lengthen or scatter, reducing luminosity or increasing the risk of beam loss. The LHC uses sophisticated feedback systems to keep the beam shapes uniform; otherwise, small instabilities can grow.

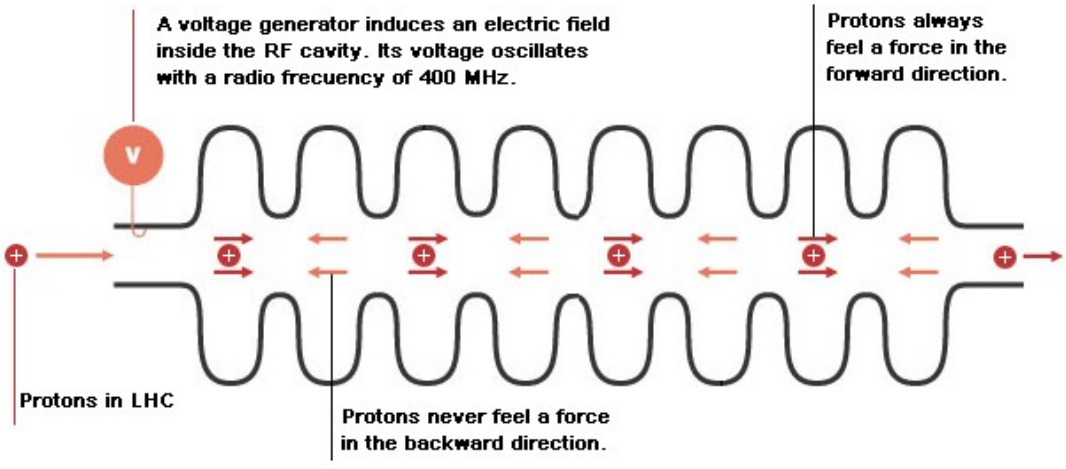


Figure 2.3. LHC RF System.

2.2. Overview of the Compact Muon Solenoid Detector

The CMS experiment is one of two general-purpose detectors situated at the LHC at CERN, designed to explore fundamental particle physics. The primary research goals of the CMS experiment include investigations into the Higgs boson, extra dimensions, and dark matter candidates. The CMS detector has dimensions of 21 meters in length, 15 meters in diameter, and a total weight of approximately 14,000 tonnes.

The core of the CMS detector consists of a superconducting solenoid magnet, generating a magnetic field of 3.8 T, essential for particle trajectory measurements. Surrounding this magnet is a steel return yoke weighing around 12,500 tonnes, which guides the magnetic flux. Inside the solenoid, the tracker and calorimeters precisely measure particle momenta and energies, while muon detectors are embedded within the return yoke outside the magnet. An overview of the CMS structure can be seen in Figure 2.4.

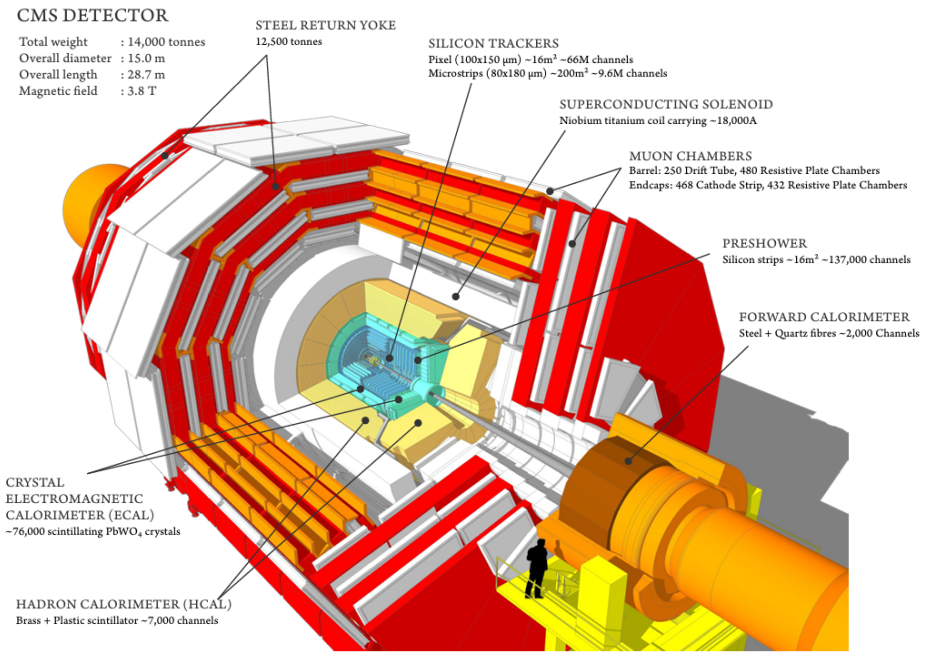


Figure 2.4. CMS detector cutaway diagram.

During collisions, each beam has a radius of 17 μm at the interaction point, with a crossing angle of 285 μrad between beams. At peak luminosity, each LHC beam contains 2,808 bunches of 1.15 $\times 10^{11}$ protons, with collisions occurring every 25 ns.

2.2.1. Coordinate System

CMS utilizes a coordinate system in which the x-axis points radially inward toward the core of the LHC, the y-axis points vertically upward, and the origin is centered at the experiment's nominal collision point. Therefore, from LHC Point 5, the z-axis points along the beam path toward the Jura mountains. The x-axis in the xy-plane is used to measure the azimuthal angle, or ϕ, while the z-axis is used to measure the polar angle θ.

The definition of pseudorapidity is given by $\eta = -\ln\tan(\theta/2)$. Consequently, the x and y components are used to calculate the momentum and energy observed transverse to the beam direction, represented by p_T and E_T, respectively. Figure 2.5 provides a visualization of the CMS coordinate system [10].

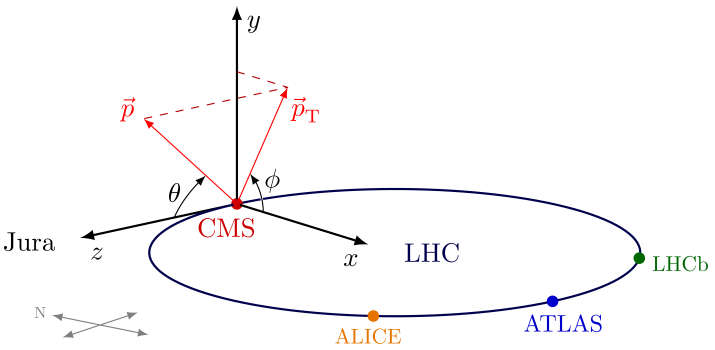


Figure 2.5. CMS coordinate system. [10].

2.2.2. Tracker

The CMS Tracker is critical for measuring the momentum and trajectory of charged particles resulting from collisions [11]. Constructed entirely from silicon-based

technology, the tracker minimizes particle interaction through its lightweight yet robust materials. Its innermost layer comprises pixel detectors, which consist of approximately 124 million pixels, each measuring 100 μm $\times$ 150 μm, arranged in four cylindrical layers at radii of 3 cm, 7 cm, 11 cm, and 16 cm, complemented by disks at either end. These pixels detect charged particles by collecting charges created when particles traverse silicon sensors. The signals from these sensors are amplified by custom-designed readout chips (ROCs), fabricated in a 0.25 μm CMOS process. The resulting electronic signals are transmitted via approximately 40,000 optical fiber links for off-detector processing and analysis. The pixel detector is shown in Figure 2.6.

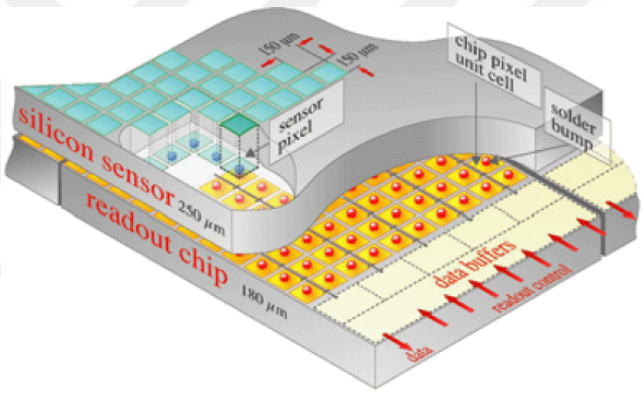


Figure 2.6. CMS silicon pixel detector.

2.2.3. Electromagnetic Calorimeter

The Electromagnetic Calorimeter (ECAL) precisely measures electron and photon energies using homogeneous lead tungstate ($PbWO_4$) crystals [12]. The ECAL is composed of a barrel section and two endcaps, ensuring complete coverage. The barrel contains 61,200 crystals arranged in 36 supermodules, each crystal measuring 22 $\times$ 22 mm^2 in cross-section and 230 mm in length. The endcaps contain a total of 14,648 crystals, arranged in supercrystals of 5 $\times$ 5 arrays, each crystal measuring 28.6 $\times$ 28.6 mm^2 in cross-section and 220 mm in length. The ECAL uses silicon avalanche photodiodes (APDs) in the barrel and vacuum phototriodes (VPTs) in the endcaps, both suitable for high magnetic field environments. Temperature control

within 0.1 °C is crucial due to the sensitivity of both crystals and photodetectors to temperature fluctuations. For the majority of the endcap pseudorapidity range, a preshower device is positioned in front of the crystal calorimeter. The layout of the CMS electromagnetic calorimeter is shown in Figure 2.7.

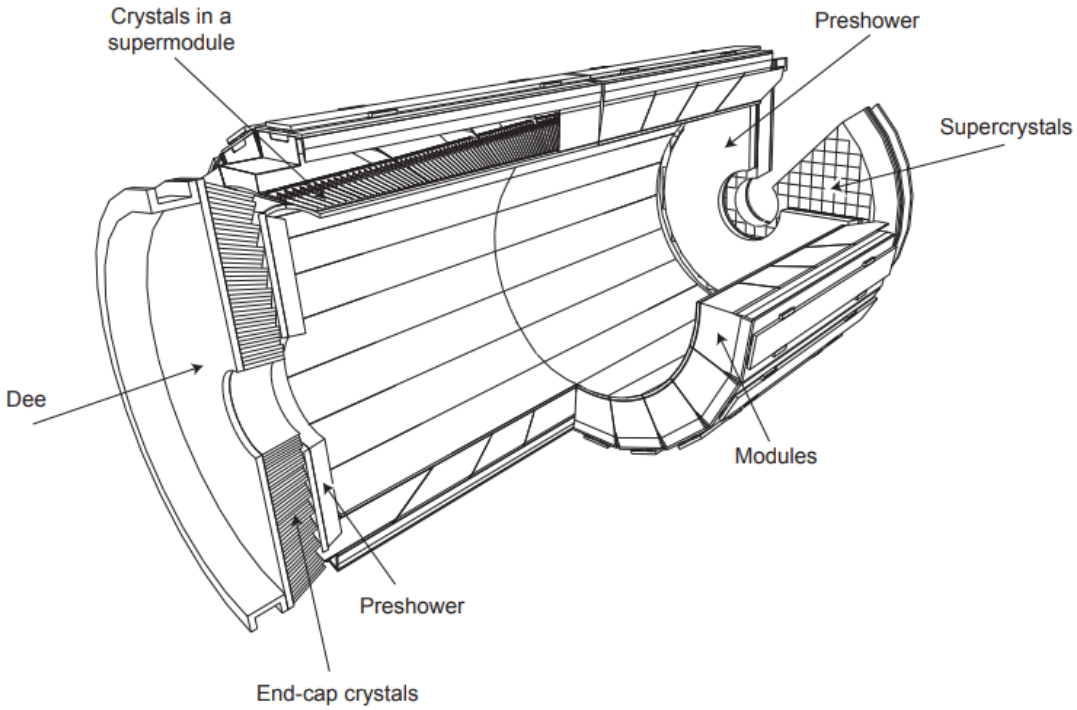


Figure 2.7. Layout of the CMS electromagnetic calorimeter.

2.2.4. Hadron Calorimeter

The Hadron Calorimeter (HCAL) is a sampling calorimeter designed to measure hadronic particle energies, comprising alternating layers of dense absorbers (brass or steel) and scintillator tiles [1]. It consists of barrel (HB), outer barrel (HO), endcap (HE), and forward (HF) calorimeters, providing comprehensive coverage as shown in Figure 2.8. The scintillator layers detect particle-induced showers, emitting light collected by wavelength-shifting fibers and transmitted to silicon photomultipliers (SiPMs). These photodetectors convert and amplify the optical signals proportionally to the deposited energy, forming calorimeter towers for energy measurement and event

reconstruction. Radiation-resistant materials are employed in regions exposed to high radiation levels to maintain performance.

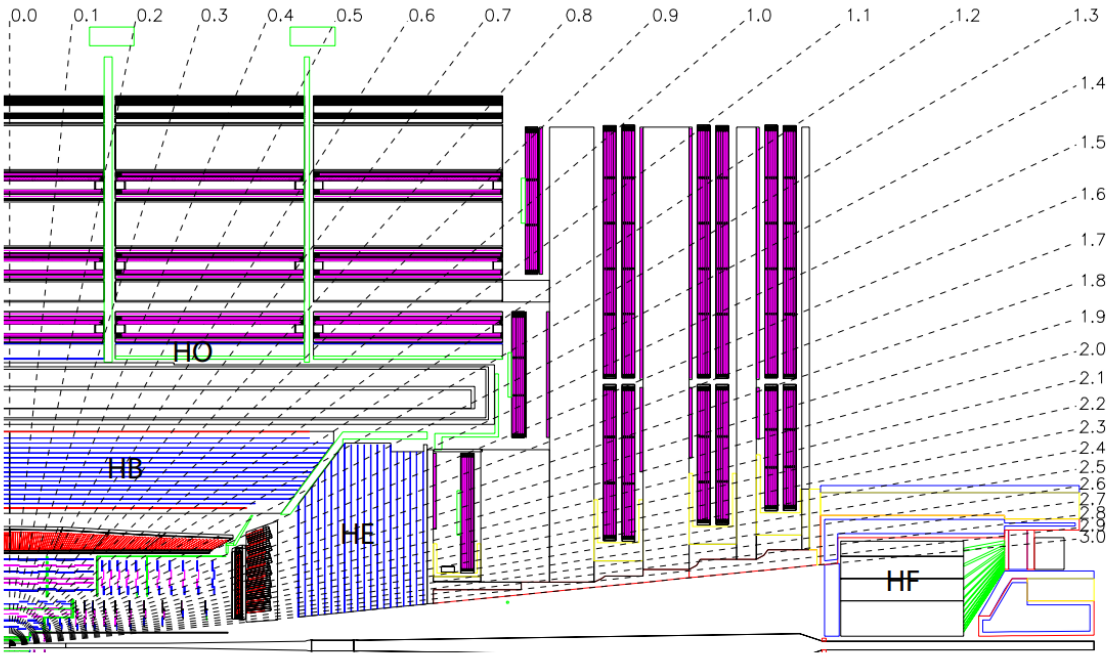


Figure 2.8. A cross section view of the CMS HCAL.

2.2.5. Magnet

The CMS superconducting magnet generates a uniform magnetic field of 3.8 T within a bore measuring 6 meters in diameter and 12.5 meters in length. The magnet stores an energy of approximately 2.6 GJ at full operational current. The magnetic flux is channeled through a 12,500 tonnes steel yoke structure consisting of five wheels and two endcaps, each composed of three disks. Safety dump circuits dissipate energy during magnet quenching events, with a notably long circuit time constant of approximately 39 hours, the longest at CERN. The primary role of the CMS magnet is to bend charged particle trajectories, facilitating precise momentum measurement.

2.2.6. Muon System

The CMS Muon System detects muons, which uniquely penetrate the calorimeters and reach the outer regions of the detector. Muon detection employs three gaseous detector technologies: Drift Tube (DT) chambers in the barrel region, Cathode Strip Chambers (CSCs) in the endcap regions, and Resistive Plate Chambers (RPCs) in both regions [12]. DT chambers are suitable for the lower muon rates and magnetic fields in the barrel. CSCs, capable of handling higher muon rates and intense magnetic fields, cover the endcap regions. RPCs provide rapid timing and bunch crossing (BX) identification, complementing DTs and CSCs despite their coarser spatial resolution. The RPCs utilize double-gap designs with linseed oil coating to improve noise performance and operational stability under high particle rates. The layout of one quarter of the CMS muon system is shown in Figure 2.9.

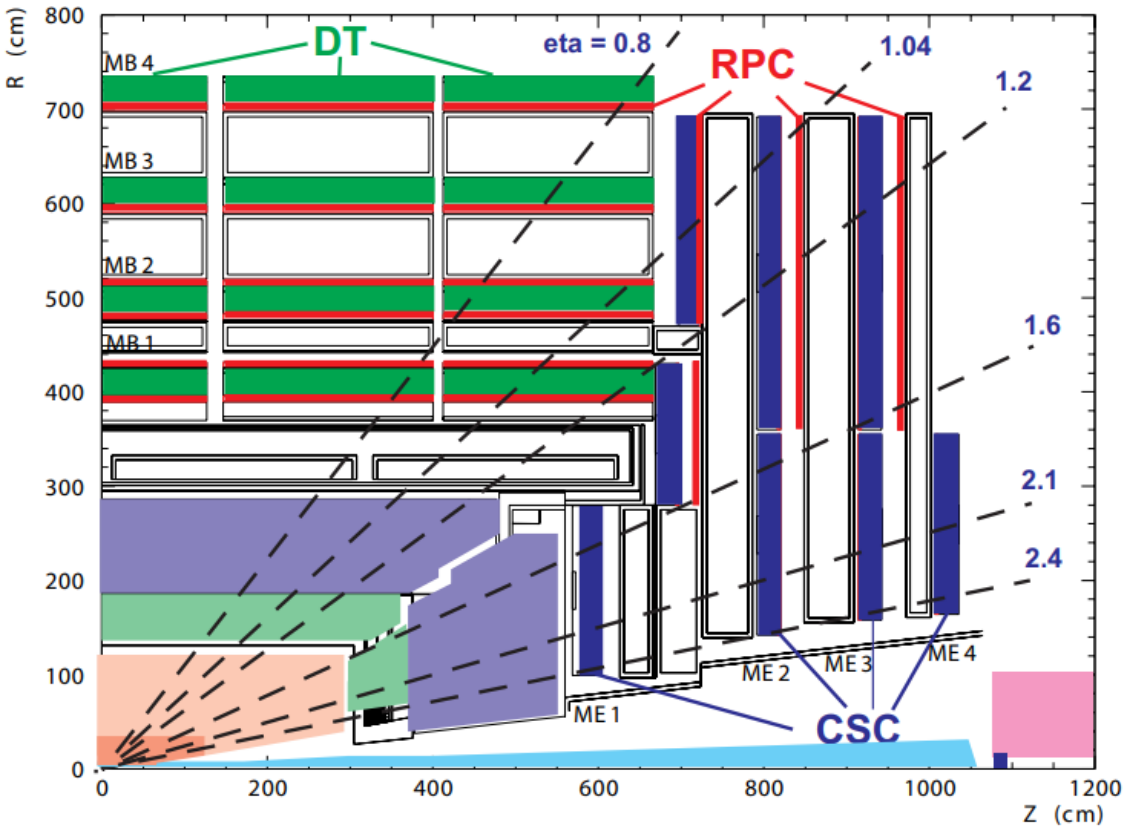


Figure 2.9. The layout of one quarter of the CMS muon system.

2.3. HGCAL Upgrade

2.3.1. HL-LHC and Motivation for HGCAL

Luminosity, often referred to as instantaneous luminosity, is defined as the rate at which beam particles traverse a given area per unit time, typically expressed in units of $\mathrm{cm}^{-2}\,\mathrm{s}^{-1}$. In the context of collider experiments, particularly for head-on collisions, the luminosity $\mathcal{L}$ is given by [13]:

$$\mathcal{L} = \frac{N_1 N_2 n_b f_{\mathrm{rev}}}{A} \tag{2.1}$$

where N_1 and N_2 represent the number of particles in each colliding bunch from two beams circulating in opposite directions, n_b is the number of bunches per beam, f_{rev} is the revolution frequency, and A is the effective beam overlap area at the interaction point. The integrated luminosity is defined as the integral of the instantaneous luminosity over time, expressed as:

$$L = \int \mathcal{L}\, dt \tag{2.2}$$

Integrated luminosity effectively represents the total amount of data collected by an experiment and is a critical parameter for evaluating the performance and physics potential of an accelerator. It is typically measured in inverse cross-sectional units such as inverse femtobarns (fb^{-1}).

The HL-LHC aims to increase the integrated luminosity by approximately an order of magnitude compared to the original LHC design. Specifically, the HL-LHC targets achieving a peak luminosity of 5×10^{34} $\mathrm{cm}^{-2}\mathrm{s}^{-1}$ and an integrated luminosity of about 3000 fb^{-1} over its operational lifetime.

The operational plan of the LHC consists of alternating periods of data-taking, referred to as runs, and extended upgrade and maintenance intervals known as Long Shutdowns (LS). To date, the LHC has undergone four main periods: Run-1, Long Shutdown 1 (LS1), Run-2, and Long Shutdown 2 (LS2). Throughout these intervals, incremental improvements in both luminosity and beam energy have been realized.

Currently, CERN is conducting its third data-taking phase, known as Run-3. Upon its conclusion, the accelerator will undergo the Long Shutdown 3 (LS3), marking the transition from Phase-1 to Phase-2 operations, commonly referred to as the HL-LHC era. A timeline of the transition from LHC to HL-LHC operations is presented in Figure 2.10.

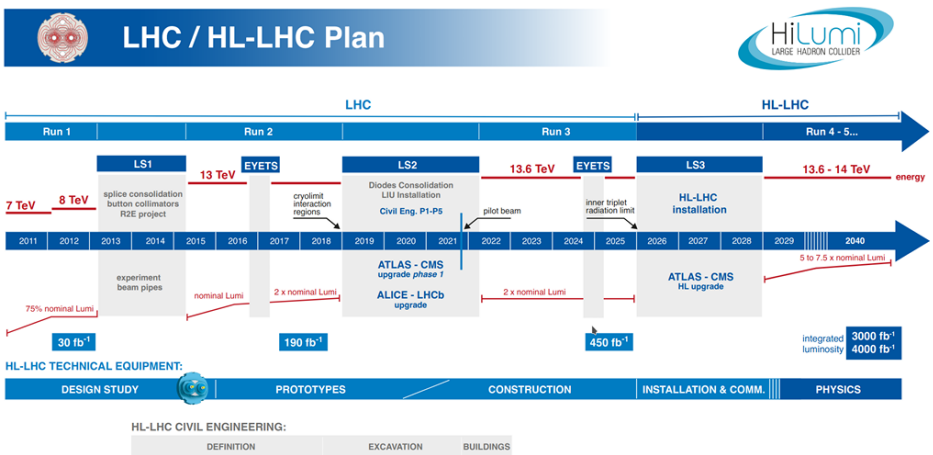


Figure 2.10. Timeline of LHC and HL-LHC operations and upgrades.

2.3.2. HGCAL Overview

The HGCAL comprises an electromagnetic compartment (CE-E) followed by a hadronic compartment (CE-H). The entire calorimeter is housed within a thermally shielded enclosure cooled by a two-phase CO_2 cooling system, maintaining an operating temperature of approximately $-35\,^{\circ}\mathrm{C}$ [14]. Figure 2.11 depicts the cross-sectional view of the upper half of one endcap.

The CE-E is composed of 26 sampling layers with a total thickness of 34 cm, equivalent to approximately 27.7 radiation lengths (X_0) and 1.5 nuclear interaction lengths (λ). The primary detector element consists of hexagonal silicon sensors, each 163 mm wide, fabricated from 8-inch wafers. These sensors are selected according to radiation exposure levels, with thinner sensors deployed in regions of higher fluence.

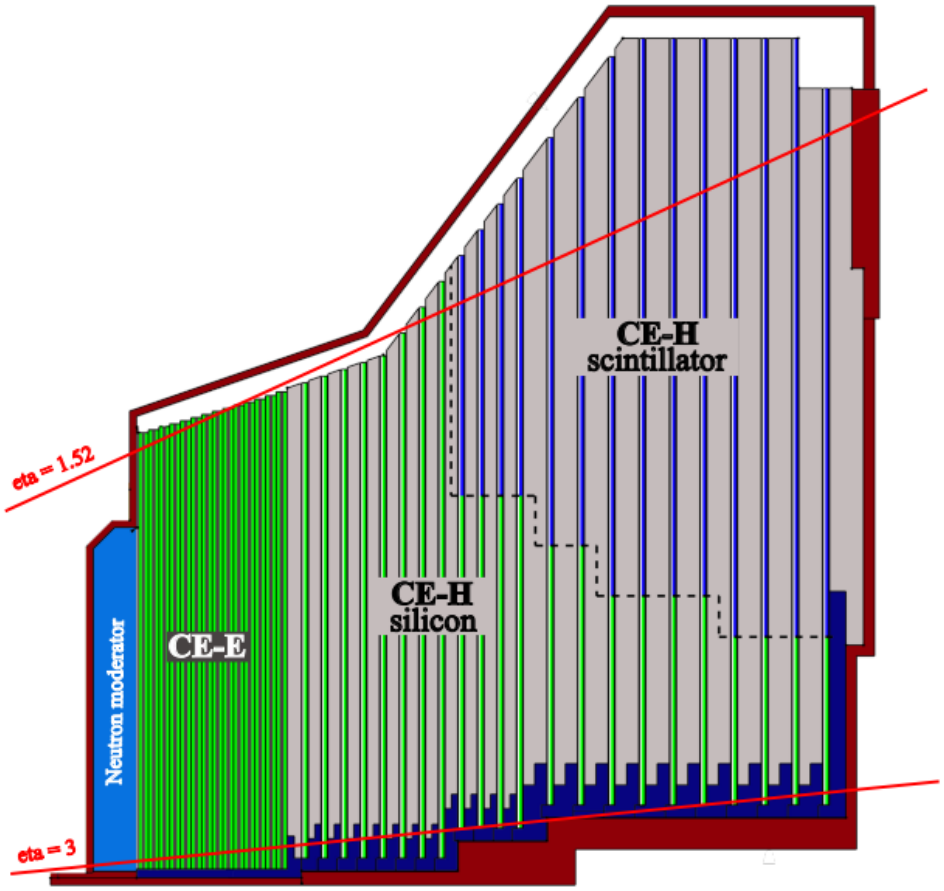


Figure 2.11. Longitudinal cross section of HGCAL endcap calorimeter.

Each silicon sensor is integrated into a module formed by sandwiching it between a 1.4 mm thick tungsten-copper (CuW, 75% W and 25% Cu) baseplate and a printed circuit board hosting the FE electronics. The silicon modules are positioned on both sides of a 6 mm thick copper cooling plate, forming an absorber layer together with the adjacent CuW baseplates. Alternating absorber layers consist of two lead plates, each 2.1 mm thick and clad with 0.3 mm stainless steel sheets, placed around the cooling plate-module assembly. This structure is organized into 60° segments called cassettes, with 14 layers of these cassettes providing the total 26 sampling layers.

In the CE-H, the absorber structure comprises 12 stainless steel (SS) plates, each 35 mm thick, followed by an additional 12 SS plates, each 68 mm thick. Silicon modules and scintillator tileboards are mounted onto 6 mm copper cooling plates and

arranged into 30° wide cassettes, similar to those in the electromagnetic compartment but with sensors installed only on one side. The absorber layers in CE-H are structured separately from the sensor-cooling assemblies.

Overall, the combined thickness of the calorimeter, perpendicular to the sensor planes, is approximately 10.7 (λ). This includes the electromagnetic compartment and a 120 mm thick polyethylene neutron moderator placed in front to reduce neutron flux entering the Tracker. While all layers are read out for energy measurements, only alternate layers in CE-E and all layers in CE-H contribute to Level-1 trigger primitives.

2.3.3. Active Elements

2.3.3.1. Silicon Sensors. The silicon sensors for the CE-E and CE-H compartments will be planar, DC-coupled hexagonal sensors fabricated from 8-inch silicon wafers. The hexagonal geometry optimizes wafer usage by reducing peripheral inactive area relative to active area. Sensors will have varying active thicknesses of 300, 200, and 120 μm, optimized to maintain efficient charge collection and performance throughout the calorimeter's lifetime. For the 300 and 200 μm thick sensors, the baseline substrate material is physically thinned p-type Float Zone silicon wafers. A potential alternative for the 120 μm thick sensors is p-type epitaxial silicon on a handle wafer. P-type sensors are preferred due to their demonstrated robustness against radiation-induced non-Gaussian noise.

Sensor cells are DC-coupled directly to the on-module readout electronics, which maintain the sensors at virtual ground and can supply up to 20 μA per cell. Connections between sensors and readout electronics are made via wire bonds through dedicated openings in the readout printed circuit board (PCB).

2.3.3.2. Silicon Modules. Approximately 27,000 silicon detector modules will be assembled and installed within the CE-E and parts of the CE-H sections of the HGCAL. Each CE-E module comprises several components stacked together: a baseplate, a

Kapton-gold insulating layer, a silicon sensor, and a hexagonal PCB known as the hexaboard, which carries the FE electronics [14]. These components share roughly identical areas. The baseplate features precision reference holes to ensure accurate alignment and positioning during assembly. A Kapton foil coated with a thin layer of gold is adhered to the baseplate to provide electrical insulation between the silicon sensor's back-plane and the grounded baseplate. The hexagonal silicon sensors and hexaboards feature small cutouts at each corner, facilitating access for wirebond connections to the gold-coated Kapton layer used for sensor biasing. The hexaboard houses the FE readout ASICs, known as HGCAL Readout Chip (HGCROC). Signals from individual sensor pads are routed directly to the HGCROC ASICs for digitization. Wirebonding holes strategically located in the hexaboard provide access for connecting sensor pads in groups of up to four. The assembly of baseplate, Kapton-gold layer, silicon sensor, and hexaboard is bonded using epoxy, resulting in a robust and unified detector module shown in Figure 2.12.

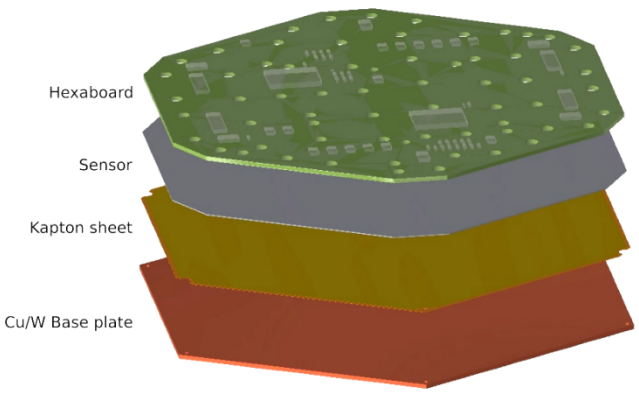


Figure 2.12. Layers of silicon-based HGCAL modules.

2.3.3.3. Plastic Scintillators and Scintillator Tile-modules. The hadronic section of the CMS endcap calorimeter requires replacement due to anticipated signal degradation before the commencement of the HL-LHC operational phase. To address this, two types of scintillator materials have been evaluated: polyvinyltoluene-based and polystyrene-based scintillators. After considering factors such as cost, performance, and ease of

manufacturing, cast and machined polyvinyltoluene-based scintillators have been selected for the front portion, whereas injection-molded polystyrene-based scintillators will be implemented in the remainder of the hadronic section.

SiPMs will read the scintillation light emitted by these tiles. These photodetectors, previously employed in several CMS systems including the Phase-1 HCAL upgrade, offer exceptionally high gain exceeding 10^5 and demonstrate excellent photodetection efficiencies greater than 30%.

The scintillator-based active materials in the hadronic calorimeter are strategically utilized in regions where the cumulative radiation dose remains below 3 kGy, and neutron fluence does not exceed 8×10^{13} neq/cm^2. This approach ensures robust and stable performance throughout the entire lifespan of the HL-LHC operations, with additional safety margins considered. Consequently, the region beyond a pseudorapidity of $|\eta| > 2.4$ is exclusively instrumented with silicon sensors due to higher radiation levels that surpass the operational limits of scintillator materials.

2.3.4. Electronics

2.3.4.1. Front-end Electronics. The FE electronics offer high-precision time-of-arrival measurements, digitize the silicon sensor or SiPM signal, and provide digital data to the BE electronics. In order to construct trigger primitives, the FE also calculates digital sums of nearby cells for each BX (2×2 cells for the 1.2 cm^2 silicon pads and 3×3 cells for the 0.5 cm^2 pads). At 40 MHz, the charge and time-of-arrival are measured by the specially designed HGCROC [15]. The HGCROC must meet strict criteria: high dynamic range (from a few fC to 10 pC), low noise (∼2,000 electrons), high-precision timing (to mitigate pileup), and low power consumption (15 mW/channel). It must also operate in a severe radiation environment, tolerating up to 300 MRad. The HGCROC includes a 10-bit 40 MHz successive-approximation analog-to-digital converter (SAR ADC) that provides charge measurements within the preamplifier's linear range. It features 72 analog channels, each with low-noise, high-gain preamplifiers and shapers.

A discriminator and a time-to-digital converter (TDC) with 50 ps binning measure charge over a 200 ns dynamic range in the saturation region. Another fast discriminator and TDC provide timing information to 25 ps precision. While waiting for a L1A, charge and timing data are stored in memory. For trigger primitive generation, data from adjacent channels (4 or 9, depending on sensor granularity) are transmitted at 40 MHz over 1.28 Gb/s electrical lines to the ECON-D, where data suppression occurs. Digital sums of these channels are also sent to another concentrator ASIC, ECON-T, for each BX. The ECON-D aggregates, formats, and serializes the data at an L1A rate of up to 750 kHz. The ECON-T selects relevant trigger sums, packages the data, and stores them in First-in-First-out (FIFO) buffers. Data are transmitted via optical links to the trigger primitive generator (TPG) circuits under defined latency constraints. Together, ECON-D and ECON-T act as data and control hubs, receiving up to twelve 1.28 Gb/s links from HGCROCs and outputting trigger and readout data at matched speeds. They are linked to a Versatile Transceiver Plus (VTRx+) [16], a Slow Control Adapter ASIC (SCA) [17], and low-power Gigabit Transceivers (lpGBTs) [18] via bidirectional links to BE electronics. All components were developed for high-energy physics (HEP) research. Slow control links operate using the I2C protocol at 1 Mb/s, while fast control links operate at 320 Mb/s. VTRx+ optoelectronic transceivers handle optical communication.

PCB layouts for the FE electronics are optimized for data throughput and component layout. For high-density silicon modules, ECONs are placed on a single PCB (wagon board), with lpGBTs and VTRx+ located on a separate PCB (engine board). For low-density modules, ECONs are on mezzanine boards independent of wagon boards. Scintillator module motherboards integrate the VTRx+, lpGBT, and ECON components. Miniature connectors link wagon boards to engine boards. lpGBTs utilize the FEC5 protocol for error correction in the data path. The average event size for the full HGCAL is about 2.5 MB, increasing nearly linearly with pileup. This results in a total data rate of 15 Tb/s from the FE electronics at a 750 kHz L1A rate. For the trigger path, the average rate is 50 Tb/s, or 1.25 Mbit per BX. Roughly 10,000 lpGBT lines, each operating at 10.24 Gb/s, connect the FE to both DAQ and TPG BE

electronics. Figure 2.13 shows a schematic representation of the HGCAL FE readout, trigger, and control chain.

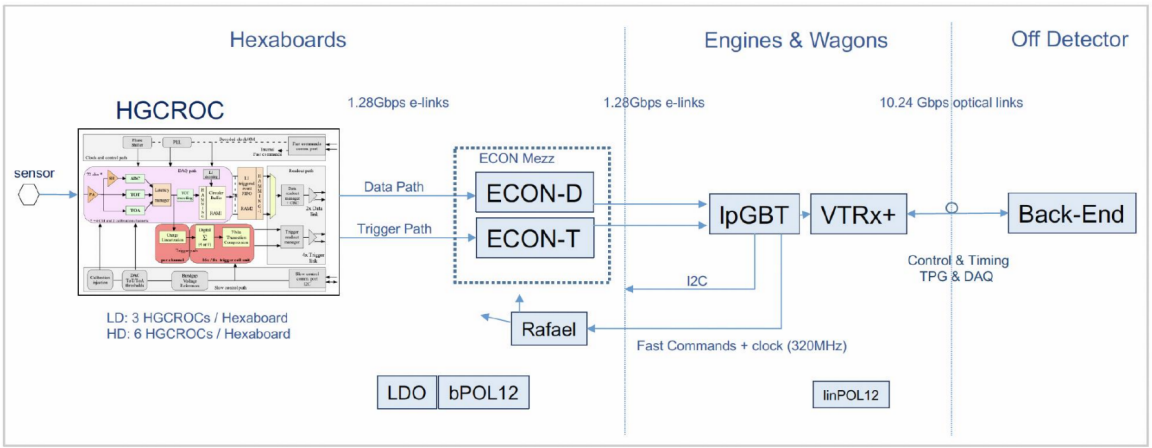


Figure 2.13. CMS HGCAL FE chain.

2.3.4.2. Back-end Electronics. The DAQ and TPG systems together constitute the BE electronics. ATCA format is used to implement both the DAQ and TPG systems. The CMS common "Serenity" boards [19] and DAQ and Timing Hub (DTH) ATCA boards [20], which serve as the core interface for the DAQ and timing systems, are housed in ATCA crates. The DAQ system is composed of ATCA boards containing a single high-end FPGA with sufficient bandwidth to drive 12 optical links to and from a DTH in the same crate, as well as 108 links in both directions to and from the FE electronics. The connections to the FE electronics will operate at 10.24 Gb/s and 2.56 Gb/s, respectively, while the DTH board links will operate at 25 Gb/s. The FPGA on these boards will manage and process the data, and the firmware will provide the necessary buffering, event building, FE emulation, and monitoring functionalities. The TPG system is responsible for forming 3D clusters from trigger cells and creating the overall energy map from the coarse granularity HGCROC energy sums.

3. HGCAL BACK-END DATA ACQUISITION SYSTEM

The primary objective of the DAQ system is to manage the data pathway and provide temporal decoupling between synchronous detector readout and initial data reduction at Level-1 trigger, asynchronous event selection via the High-Level Trigger (HLT), local storage of selected events at the experimental facility, and eventual transfer to Tier-0 for permanent offline storage and further analysis.

3.1. Uplink Data Path (Front-End to Back-End)

Each BE DAQ FPGA is designed to receive and process data from up to 108 lpGBTs. Due to the non-uniform occupancy within the detector endcaps, the mapping of FE components to BE elements is highly variable, resulting in differing numbers of ECON-D modules per lpGBT link. Specifically, some ECON-Ds require two lpGBT links, while others can share a single link with as many as six ECON-Ds. To minimize maintenance complexity arising from multiple firmware variants, a single, robust, and flexible firmware design is developed to handle all scenarios.

In the DAQ FPGA firmware, 54 processing block instances named Capture Block handle data reception and buffering from the 108 lpGBT links [21]. Each Capture Block handles data from two lpGBT channels and is capable of accommodating up to 12 ECON-Ds. The excess ECON-D processing units that are instantiated in the Capture Blocks that serve fewer than 12 ECON-Ds are left inactive. Consequently, a single FPGA manages data from up to 648 ECON-Ds. Data packets from ECON-Ds received by each processing unit are subsequently transferred to dedicated event buffers. Event frames from multiple event buffers are merged into larger data packets, which are then transmitted to the DTH boards via twelve 25 Gb/s SLINK channels. Figure 3.1 shows the high-level firmware blocks needed in the DAQ FPGA.

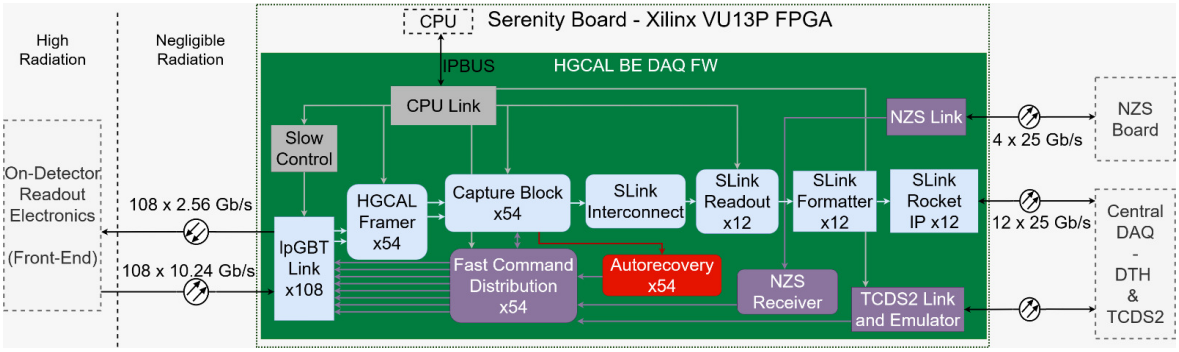


Figure 3.1. High level BE DAQ FPGA firmware blocks.

Due to variable data rates across the FPGAs, mapping the 54 processing units to these twelve SLINK outputs requires careful management. To facilitate this, configurable interconnection logic provides dynamic and flexible routing, allowing each SLINK output to aggregate data from between one and twelve event buffers, ensuring balanced utilization of fixed-rate output links.

3.2. Fast Control Commands and Autorecovery Block

The BE DAQ board receives the clock and fast control commands (fast commands) from the DTH boards via the ATCA backplane. A dedicated fast command block is implemented on the BE DAQ board to receive these fast commands, encode them into 8-bit system-specific fast commands, and distribute them across all fast command channels (FCCs).

Additionally, the fast command block includes functionality for local generation of FCs. Dedicated local fast command registers, accessible via software [21], independently drive each FCC. Furthermore, the Autorecovery Block can directly interface with FCCs to issue FCs to specific FE ASICs. This capability is especially crucial for swiftly recovering individual FE ASICs that have temporarily fallen out of synchronization. The interconnection between the Autorecovery Block and FCCs is illustrated in Figure 3.2. The Autorecovery Block constitutes the core focus of this thesis and will be extensively analyzed in subsequent chapters. A detailed list of available fast

commands, including their binary encoding and functional descriptions, is provided in Figure 3.3.

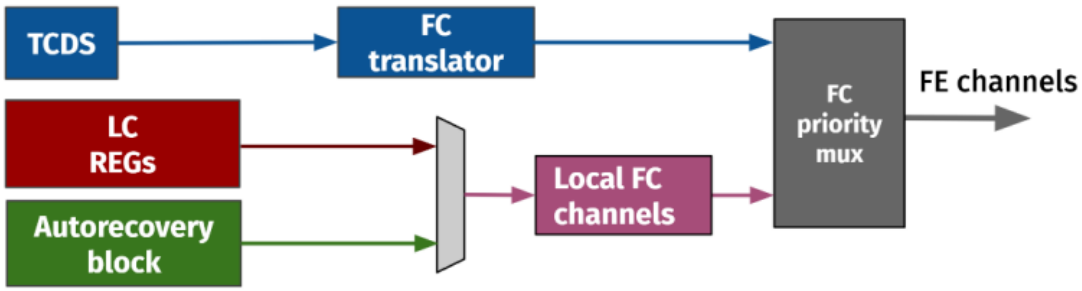


Figure 3.2. A simplified diagram of the fast control command blocks.

Commands	Binary Code	Description
Idle	0b00110110	Used for LHC clock phase recovery
PreL1A	0b11001100	Used to signal an incoming L1A
L1A	0b01001011	Level-1 Accept
L1A+PreL1A	0b11100001	Combined PreL1A and L1A
L1A+NZS	0b00001111	Level-1 Accept requesting NZS data
L1A+NZS+PreL1A	0b00101011	Combined PreL1A, L1A and NZS request
L1A+BCR	0b01110001	Combined L1A and BCR
L1A+BCR+PreL1A	0b10100101	Combined L1A and BCR and PreL1A
L1A+CalPulseInt	0b00111001	Combined L1A and CalPulseInt
L1A+CalPulseExt	0b10000111	Combined L1A and CalPulseExt
L1A+CalPulseInt+PreL1A	0b11100010	Combined L1A, CalPulseInt and PreL1A
L1A+CalPulseExt+PreL1A	0b11110000	Combined L1A, CalPulseExt and PreL1A
BCR	0b00011101	Resets BX counters
BCR+PreL1A	0b10100011	Combined BCR and preL1A
BCR+OCR	0b10010101	Resets Orbit and BX counters
CalPulseInt	0b00101101	HGCROC Internal Calibration Pulse
CalPulseExt	0b01111000	HGCROC External Calibration Pulse
CalPulseInt+PreL1A	0b01010101	Combined CalPulseInt and PreL1A
CalPulseExt+PreL1A	0b10010011	Combined CalPulseExt and PreL1A
ChipSync	0b11010010	Clears buffers, resets counters
EBR	0b11010001	Resets event buffers
ECR	0b10101001	Resets event counters
LinkResetROCT	0b10011001	HGCROC outputs special TPG pattern
LinkResetROCD	0b10011010	HGCROC outputs special DAQ pattern
LinkResetECONT	0b10101010	ECON-T outputs special TPG pattern
LinkResetECOND	0b10111000	ECON-D outputs special DAQ pattern
ECOND-Watchdog-Clear	0b10001110	Clears the ECON-D watchdog timer
ROC-Serializer-Reset	0b10011100	Resets the HGCROC serialiser

Figure 3.3. Fast Command Table.

3.3. Capture Block

The firmware developed by the CMS collaboration exhibits a highly modular and sophisticated architecture, reflecting the complexity of the data acquisition requirements in high-energy physics experiments. Among its critical components, the Capture Block plays a central role in the real-time processing pipeline.

This section provides an examination of the Capture Block and its interactions with upstream and downstream systems. The Capture Block is designed to aggregate data corresponding to the same event from all ECON-D ASICs connected via a given fibre pair [22]. This procedure, referred to as event building, consists of two primary stages: packet detection and packet matching.

To ensure compatibility with a wide range of FE configurations—specifically, those that distribute the output of up to 12 ECON-Ds across 14 e-links per fibre pair—the architecture is made fully configurable. This design flexibility is achieved while maintaining efficient utilization of FPGA logic resources. The overall structure of the Capture Block, including its key submodules and input/output interfaces, is illustrated in Figure 3.4.

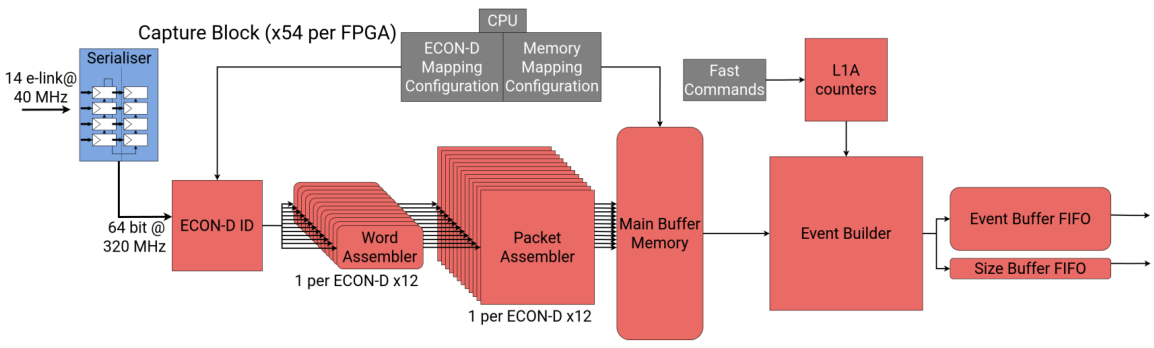


Figure 3.4. Capture Block architecture showing the interface with lpGBT input, ECON-D deserialization, packet detection, event building, and output buffering mechanisms.

Particular emphasis is placed on the Event Builder module, which constitutes the core logic of the Capture Block. It is responsible not only for assembling complete event fragments from buffered ECON-D data but also for interfacing with the Autorecovery Block through a state machine that flags error conditions arising during event construction.

3.3.1. Packet Detection

Prior to transmission to the Capture Block, a serializer module aggregates pairs of e-links and converts the resulting data into a 64-bit stream. This stream operates at the Capture Block's system frequency of 320 MHz and is synchronized with the LHC clock to maintain deterministic timing alignment.

Upon reception within the Capture Block, each e-link word is assigned an identifier corresponding to its originating ECON-D ASIC. This mapping is established via configuration registers that are accessible through the IPbus protocol [23]. Once the association is made, the data stream is forwarded to twelve dedicated Word Assembler units. Each assembler selectively filters and aligns data words associated with its assigned ECON-D, thereby generating twelve separate data streams, each containing only the payload from a single ECON-D. These refined data streams are then prepared for the subsequent packet detection phase of the event-building process.

Within each ECON-D-specific data stream, a Packet Assembler module parses the incoming words to detect valid ECON-D packets. This is accomplished by identifying the presence of a packet header marker that follows an idle word. To further mitigate the risk of falsely detecting spurious packets, a cyclic redundancy check (CRC) validation step is implemented. Each packet header includes an embedded 8-bit CRC field, which is checked against a freshly computed CRC value within the packet assembler. Only when the CRC values match is the packet deemed valid, prompting the assembler to extract and store the full packet into a shared main buffer memory accessible to all ECON-D channels.

Given that ECON-D words are transmitted over 32-bit e-links, yet the internal data handling is performed in 64-bit format, the packet assembler also appends a padding word when necessary to ensure proper data alignment. This guarantees consistency in downstream data processing stages. The structure of an ECON-D packet, including the header, payload, and idle word, is illustrated in Figure 3.5.

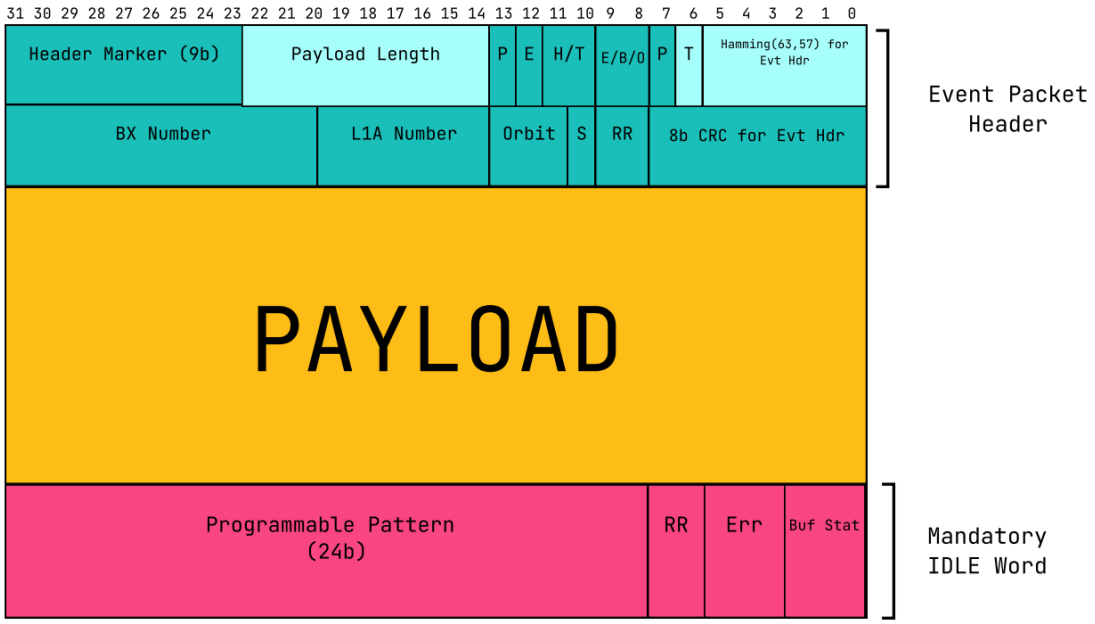


Figure 3.5. Format of an ECON-D packet.

3.3.2. Main Buffer

The main buffer memory is implemented using high-density UltraRAM (URAM) resources and is shared among all packet assembler modules. This architectural choice enables efficient utilization of FPGA memory by allowing dynamic control over memory allocation through configuration registers, which are accessible via the IPbus protocol. Such configurability makes it possible to tailor the buffer size allocated to each packet assembler based on the anticipated data rate of its associated ECON-D channel. Additionally, when fewer than twelve ECON-Ds are active in a given configuration, memory resources can be conserved by disabling allocation for unused packet assemblers.

3.3.3. Packet Matching and Output Buffer

The Capture Block also receives the L1A fast command signal transmitted from the FE electronics, triggering the acquisition of events. Dedicated local counters corresponding to the LHC orbit, BX, and Event-ID numbers ensure synchronization with other system components by generating timestamps indicating the precise timing of each event. Upon reception of an L1A signal, the current values of these local counters are stored in a dedicated FIFO memory, which is subsequently accessed by the Event Builder module.

The final stage of the event-building process, termed *packet matching*, is initiated once the Event Builder, implemented as a state machine, detects the simultaneous availability of data in the main buffer and a corresponding L1A timestamp. The state machine then assembles event fragments from each ECON-D channel into a coherent packet intended for transmission from the BE FPGA. During this assembly, the Event Builder verifies consistency by comparing the local L1A counters with those embedded in the incoming ECON-D packet headers. If these timestamps match, the validated event data is retrieved from the main buffer and stored in the Event Buffer FIFO. If a mismatch or another error condition is detected (such as a timeout or buffer overflow), appropriate metadata flags are set in the Capture Block header.

The Capture Block header encapsulates essential metadata, including the Capture Block's L1A timestamp and error indicators specific to each ECON-D channel, such as timestamp mismatches, timeout occurrences, or main buffer overflows. This structured header format facilitates subsequent parsing and analysis by offline processing software within the central CMS DAQ system.

The format of the Capture Block event header, detailing the associated 3-bit error code descriptions and payload structure, is depicted in Figure 3.6. Descriptions of the 3-bit error codes applicable to ECON-D data are summarized in Table 3.1.

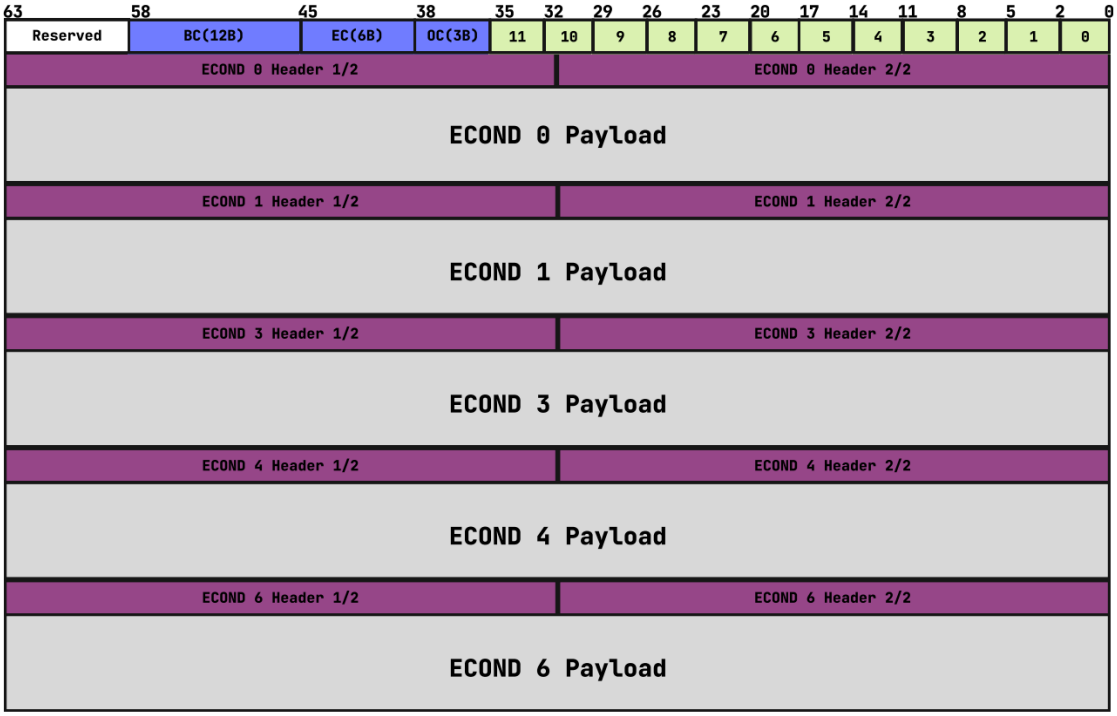


Figure 3.6. Capture Block event header format highlighting two primary header fields: the L1A Timestamp (combining the Bunch Crossing (BC), Event Counter (EC), and Orbit Counter (OC)) and the 3-bit ECON-D Packet Status fields.

Table 3.1. Description of the 3-bit ECON-D status flags.

3-bit ECON-D Status Flag	*Meaning*	*ECON-D Data Present*
0b000	Good packet	✓
0b001	Reserved	N/A
0b010	Reserved	N/A
0b011	Packet with Event-ID mismatch	✓
0b100	No ECON-D packet received (Capture Block timeout)	×
0b101	BX mismatch between Capture Block and ECON-D	× by default, ✓ if BX mismatch passthrough enabled
0b110	L1A received but no ECON-D data (buffer overflow)	×
0b111	ECON-D is not connected	×

3.3.4. The Event Builder State Machine

In order to fully understand the work presented in Chapter 7, it is essential to have a clear grasp of the Event Builder's finite state machine (FSM). The Event Builder

FSM consists of nine distinct states; the relatively high number of states is primarily due to the read latency constraints associated with accessing the Main Buffer. The complete VHDL implementation of the Event Builder FSM is provided in Appendix A.1, while a state transition diagram illustrating the FSM behavior visually is presented in Figure 3.7. For the purpose of this thesis, emphasis is placed specifically on the first five states, as the payload-related states are outside the scope of this investigation.

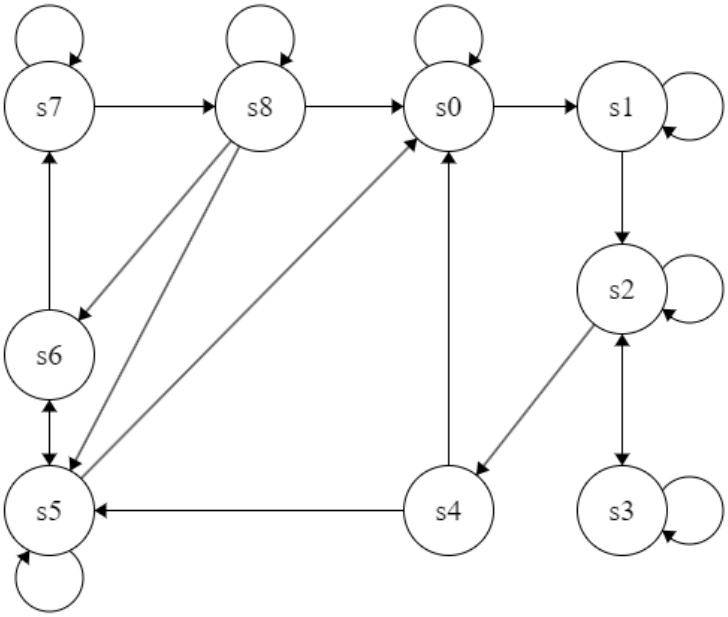


Figure 3.7. FSM diagram of the Event Builder.

The FSM states of particular importance are summarized as follows:

(i) *s0_WaitforL1a*: This state serves as an idle or initialization state in which signals are assigned their default values, and all error flags are cleared. The actual event-building process begins when a L1A signal is detected in the local L1A FIFO. Provided the event buffer is not full, the local Orbit, BX, and Event-ID identifiers associated with this L1A are captured and stored, and the FSM transitions to the next state.

(ii) *s1_WaitforPackets*: In this state, the FSM checks the status of the empty flags for each active channel-side buffer, waiting until packets from all ECON-D channels

arrive in the Main Buffer or until a predefined timeout occurs. If packets from all expected ECON-Ds become available, the FSM moves directly to the subsequent processing step. If a timeout is reached first, the FSM proceeds by reading only the available packets, explicitly tagging any missing ECON-D packets for further error handling.

(iii) *s2_GetHdrBX*: The `s2_GetHdrBX` state is responsible for checking each ECON-D module to determine whether it has provided a valid packet following an L1A trigger. It iterates through all ECON-Ds using an index (`econd`), and for each one, it checks the `econd_pkt_det` flag to see if a packet is present. If a packet is detected, it sets the corresponding memory address (`mb_addr_o`) and resets a delay counter to prepare for timestamp validation in the next state (`s3_Timestamp_Check`). If no packet is found (indicating a timeout), it marks this condition in the event header (`be_header`) with a specific 3-bit flag (`"100"`) and sets a timeout flag in the error tracking array. The FSM then proceeds to the next ECON-D by incrementing the index. Once all ECON-Ds are processed (`econd = econ_num`), it appends the L1A timestamp (BX, Event-ID, Orbit) into the event header and transitions to the `s4_Header` state to continue building the complete event record.

(iv) *s3_Timestamp_Check*: The `s3_Timestamp_Check` state verifies the correctness of each ECON-D packet's timestamp by comparing its BX and Event-ID against the local L1A values after the required memory read latency (`mb_READ_LATENCY_B`). If the BX matches, the packet is marked as valid (`econd_pkt_good`) and a read signal is sent to the packet info FIFO. Then, if the Event-ID also matches, the event is considered a fully good packet and is encoded as `"000"` in the back-end event header. However, if the Event-ID does not match, the packet is still accepted but encoded as `"011"` to indicate an Event-ID mismatch, and this condition is explicitly logged by setting `error_flags_array(econd + 1).ev_mismatch <= '1'`, flagging it for the Autorecovery Block to track and potentially react to. If the BX does not match, the state evaluates other error conditions: a buffer full flag sets the error code to `"110"` and logs `error_flags_array(econd + 1).buf_overflow <= '1'`, while a large packet condition sets the code to `"001"` and also flags `buf_overflow`. If no special flags are set, the design checks whether

BX mismatch passthrough is enabled; if not, the packet is rejected and marked with `"101"` for timestamp mismatch, and the corresponding error is recorded with `error_flags_array(econd + 1).bx_mismatch <= '1'`. These error flags are vital, as they are forwarded to the Autorecovery Block logic, which uses them to assess the condition of each ECON-D channel and apply corrective actions based on configurable thresholds. After handling each ECON-D, the FSM increments the ECON-D index and returns to `s2_GetHdrBX` to process the next one. If the memory read is not yet complete, it simply increments the delay counter and remains in this state.

(v) *s4_Header*: The `s4_Header` state handles the insertion of the BE Event Header into the Event Buffer FIFO and determines whether the full readout process should proceed. At the start of this state, it resets the delay and econd counters, sets `ev_data_o` to the completed `be_header`, and asserts `ev_wren` to write this header into the event FIFO. It also asserts `error_flags_valid` to indicate that the error flags collected so far are now valid. Then, it checks whether none of the ECON-Ds provided valid data (i.e., `econd_pkt_good` is all zeros). If no valid packets were received, it triggers an event counter reset (`rst_ev_cnt <= '1'`) and returns to the initial state `s0_WaitforL1a` to wait for the next L1A trigger. Otherwise, it proceeds to the `s5_startReadout` state to begin reading the valid ECON-D packet data into the FIFO.

4. BACK-END FIRMWARE TOOLS AND FRAMEWORKS

4.1. Serenity Board

The Serenity board is an advanced FPGA-based carrier card developed for the CMS Phase-2 upgrade at the LHC. Designed for versatile use across multiple sub-detector systems, Serenity primarily serves as a high-performance data processing platform capable of handling data rates of up to 5 TBps per board. It integrates both electrical and optical high-speed communication via SAMTEC FireFly connectors [24]. The hardware is coupled with the EMP, facilitating rapid development and deployment of algorithms independent of the underlying FPGA choice or hardware specifics.

The Serenity hardware platform comprises several key components:

- *Carrier Card:* This constitutes the core board, providing essential services such as power management, Ethernet communication, FPGA control via a service FPGA, and system management facilitated through an Intelligent Platform Management Controller (IPMC). Additionally, the carrier includes two mezzanine daughter-card connectors, various programming and debugging interfaces, as well as visual diagnostics through integrated LED indicators.
- *Daughter Cards:* Two interchangeable daughter-card slots are provided, each designed to host main FPGA modules responsible for high-throughput data processing. These daughter cards employ mezzanine connectors, offering modularity and customization tailored to specific use-cases or performance requirements.
- *Embedded CPU:* A COM Express module running CentOS 7 constitutes the on-board CPU solution. This compact computing module integrates an SSD on the carrier's underside for system storage. Peripheral connectivity includes USB 3.0 interfaces suitable for standard input/output devices (such as keyboards, mice,

or external storage), as well as a DisplayPort connector capable of outputting VGA or HDMI video signals via adapters. The COM Express module provides direct access to multiple onboard resources, serving as the primary interface to the FPGA devices. Furthermore, UART connectivity ensures communication between the CPU module and the IPMC.

- *ATCA Integration:* Serenity is engineered to function within a standard ATCA crate environment. However, for development or bench-testing scenarios, a dedicated micro-backplane can also be provided, supporting standalone operation. In such scenarios, external power (+48 V) and network connectivity via dual RJ45 Ethernet sockets (mirrored configuration for redundancy) are available.
- *Clock Distribution:* The system clock for the board can be sourced from various inputs, enhancing flexibility. Available sources include the ATCA backplane pins, an onboard precision crystal oscillator, or external clock inputs provided through MMCX differential connectors or SMA connectors on the micro-backplane. The clock distribution system has been designed to accommodate diverse synchronization requirements and thus can be configured for both simple and sophisticated clocking schemes.

The SERENITY carrier card is designed with a highly flexible connectivity architecture, providing multiple communication pathways between the carrier card and the attached FPGA daughter cards. This modular approach ensures adaptability and ease of integration into different experimental setups. Figure 4.1 illustrates all possible connection methods available for the Serenity platform.

Initially, the recommended communication methods include:

- *Ethernet* communication to the embedded COM Express CPU module (through the ATCA backplane).
- *JTAG interface* for direct programming of the FPGA daughter cards.
- *PCI Express (PCIe)* communication from the CPU to the daughter cards, leveraging the `empbutler` software framework, as utilized in this study.

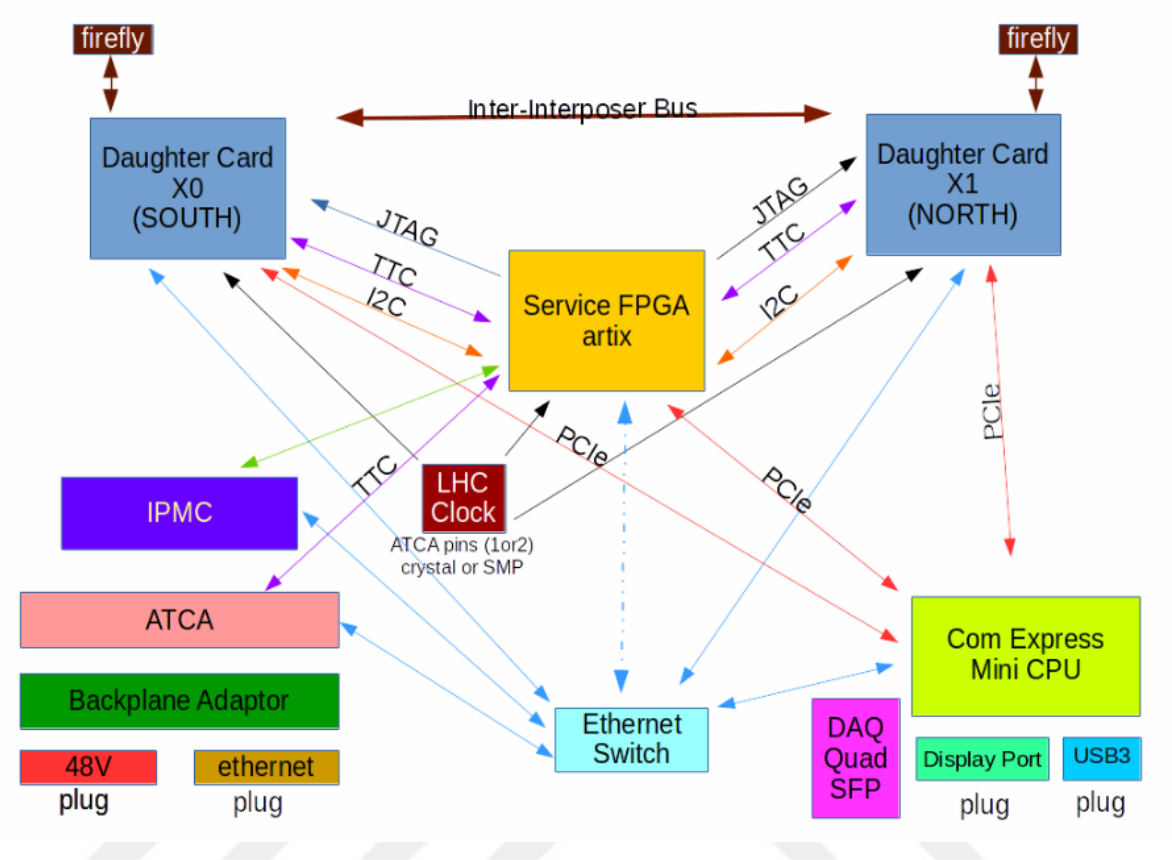


Figure 4.1. Connectivity of the Serenity board.

Specifically, there are three primary methods available to program and configure the FPGA daughter cards:

(i) *ATCA Backplane → Ethernet Switch → COM Express CPU → PCIe → Artix Service FPGA → JTAG → Daughter Card FPGA.*

(ii) *COM Express CPU → PCIe → Artix Service FPGA → JTAG → Daughter Card FPGA.*

(iii) *Direct JTAG Interface → Artix Service FPGA → JTAG → Daughter Card FPGA.*

For the work presented in this thesis, the first configuration method was employed, as described in subsequent chapters.

4.2. IPbus Protocol

Historically, the electronics systems for particle physics experiments constructed over the last few decades have typically employed the VMEbus standard. Recently, however, new electronics systems in many high-energy physics experiments have transitioned to the ATCA and Micro Telecommunications Computing Architecture (MicroTCA), collectively referred to as xTCA standards. These modern standards incorporate industry-standard serial communication technologies, such as Gigabit Ethernet; nonetheless, unlike the VMEbus standard, they do not specify a hardware access protocol for external software applications to directly read or modify the memory spaces of xTCA-based hardware.

To address this requirement, the IPbus protocol was developed. IPbus is a straightforward, packet-based control protocol specifically designed for reading from and writing to memory-mapped resources within FPGA-based IP-aware hardware [23]. It is supported by a comprehensive suite of tightly integrated software and firmware components and is employed extensively to control off-detector electronics based on xTCA standards, such as in the Phase-2 upgrades of the CMS experiment. IPbus operates on an A32/D32 (32-bit address and data) bus and provides the following defined operations:

- *Read*: Reads of user-defined depth, supporting two modes: address-incrementing mode, suitable for accessing contiguous registers in memory space, and non-incrementing mode, ideal for reading data from FIFOs or ports.
- *Write*: Writes of user-defined depth, similarly supporting both address-incrementing and non-incrementing modes to accommodate register banks and FIFOs/ports respectively.
- *Read-Modify-Write bits (RMWbits)*: Performs an atomic bitwise modification, defined by the operation $X := (X \wedge A) \vee B$. This operation efficiently sets or clears specific bits within a 32-bit register without additional overhead.
- *Read-Modify-Write sum (RMWsum)*: Executes an atomic increment operation,

described by $X := X + A$, facilitating addition or subtraction (using two's complement arithmetic) directly within registers.

4.2.1. IPbus Suite

The IPbus suite comprises firmware and software components designed to facilitate robust, packet-based communication between FPGA-based hardware and external control software [25]. The main components of the IPbus suite utilized in this thesis include the IPbus firmware module, ControlHub application, and the Micro Hardware Access Library (uHAL).

- *IPbus Firmware:* The IPbus 2.0 firmware provides a reference System-on-Chip (SoC) implementation of an IPbus 2.0 UDP server in VHDL, designed specifically for deployment on FPGAs. This firmware interprets and executes IPbus transactions within the FPGA, sharing resources seamlessly alongside application-specific processing logic, such as trigger or data acquisition algorithms. Reliability is ensured by an IPbus-specific error correction mechanism, which automatically detects and compensates for any packet loss, duplication, or reordering occurring in the UDP communication channel.
- *ControlHub:* ControlHub is a dedicated software application acting as a centralized access point for IPbus communication between control software and multiple FPGA-based devices. It performs arbitration of concurrent control commands from multiple client applications, managing simultaneous interactions seamlessly. Additionally, ControlHub implements the IPbus reliability mechanism at the software-device interface level to ensure robust communication over UDP packets. Communication between ControlHub and higher-level control applications is conducted over TCP, benefiting from its sophisticated congestion mitigation and flow-control features.
- *uHAL Library:* uHAL provides a convenient, user-oriented API in both C++ and Python for performing IPbus transactions, including read, write, and read-modify-write (RMW) operations. uHAL employs a delayed dispatch execution

model, allowing multiple transactions to be queued and combined into a single IPbus packet payload. These packets are dispatched to hardware either explicitly upon calling the dispatch method or automatically upon exceeding the maximum packet size.

Furthermore, uHAL utilizes XML files to define device-specific register maps. Each node within the XML structure corresponds to either an individual register, a block RAM, a FIFO, or an organized group of these components. The XML schema supports referencing external address files, significantly simplifying the management and integration of repeated firmware modules without redundant content. Consequently, the software control architecture naturally mirrors the hierarchical, modular nature of the firmware designs it manages.

4.3. EMP Framework

The EMP framework provides a comprehensive firmware and software infrastructure designed to facilitate the development and deployment of user-specific algorithms on FPGA-based systems. This infrastructure is particularly suited for data-processing tasks commonly encountered in particle physics experiments, such as the reconstruction algorithms utilized within Level-1 track finders or trigger processing boards. The EMP framework offers several core capabilities:

- *Control Bus Master:* Provides centralized communication to configure, control, and monitor payload firmware components.
- *TTC Generation and Reception:* Manages Timing, Trigger, and Control (TTC) signals critical for synchronizing the payload logic with the global experiment timing and control infrastructure.
- *High-Speed Data Transmission:* Supports robust data transfer over high-speed electrical and optical communication links, enabling efficient input/output data flow.
- *Datapath Buffer Management:* Facilitates injection of user-defined data patterns into payload firmware via the control bus, enabling the playback and capture of

payload-generated output data for subsequent analysis.

The EMP infrastructure is available through a repository (*emp-fwk*) containing top-level designs tailored for a variety of FPGA platforms and hardware boards. Each EMP top-level design instantiates a common FPGA entity named `emp_payload`, which encapsulates the user-specific payload firmware. This payload entity is systematically connected to the clocking infrastructure, control bus interfaces, and dedicated data buffers to ensure seamless integration and operation.

Several configurable parameters of the EMP infrastructure, such as input/output clock frequencies and I/O buffer depths, are adjustable through constants defined in the dedicated VHDL package `emp_payload_decl`. This provides flexibility and adaptability, allowing the framework to be easily customized for various applications and performance requirements.

4.4. Firmware Build and Test Workflow

Before firmware can be compiled and deployed to the FPGA on the SERENITY carrier board, several prerequisite tools must be installed. These include `Vivado` [26], the `uHAL` library, `Python`, and the IPbus Build tool known as IPBB.

The IPBB is a command-line utility developed to facilitate firmware project setup, version control, and reproducible builds in complex FPGA designs. Given that modern firmware projects typically comprise hundreds of HDL source files, along with multiple dependencies and reusable modules from collaborative repositories, IPBB plays a critical role in simplifying project management.

In addition to providing a straightforward command-line interface for Vivado-based project creation, synthesis, and implementation, IPBB also supports automated generation of IPbus address decoding logic directly from `uHAL` address tables, streamlining the integration of software-defined register maps.

To incorporate a custom algorithmic payload within the EMP firmware infrastructure, the user must implement their algorithm in an entity named `emp_payload`. This entity must follow the EMP framework's defined interface specification.

The firmware testing workflow facilitates the injection of user-defined data into the payload and allows for the corresponding output to be captured and analyzed. This procedure encompasses several stages, including FPGA programming, preparation of the uHAL connections file, loading of input data, configuration of output buffers, and retrieval of captured data. A comprehensive explanation of each step is provided in the EMP Framework documentation.

5. AUTORECOVERY BLOCK AND BEHAVIORAL VERIFICATION

5.1. Autorecovery Block Architecture

The Autorecovery Block is a dedicated firmware module integrated within the Capture Block of the BE DAQ system. Its primary role is to monitor and process error flags originating from the ECON-D FE concentrator ASICs, which are received via the Event Builder. Each error flag associated with an ECON-D input is independently counted and analyzed. When a flag exceeds a pre-defined threshold, a corresponding fast command is generated and routed to the appropriate FCC. This mechanism is essential for ensuring robust and responsive mitigation strategies against transient errors in the detector FE. The Autorecovery Block comprises two main submodules: the *Arbiter* and the *Router*. The Arbiter is responsible for the accumulation and threshold comparison of the incoming flag signals. Upon validation, it signals the Router submodule, which subsequently directs the appropriate fast command to one of the predefined output channels.

The architecture of this block was initially developed by Alp Şarkışla at Boğaziçi University, under the KareblokLab research group, for integration within the CMS HGCAL BE electronics.

5.1.1. Thresholds, Priorities, and Fast Commands

The generation and routing of fast commands within the Autorecovery Block are governed by a set of configurable thresholds and priority rules. Each incoming error flag from an ECON-D input channel is associated with a predefined threshold value. When the number of consecutive occurrences of a given flag exceeds this threshold, a corresponding fast command is issued. Multiple ECON-D sources may be mapped to a single fast command output channel. In cases of contention, a fixed priority scheme is

applied: ECON-D instances with lower indices take precedence over those with higher indices (e.g., `ECON-D 1` has priority over `ECON-D 2`). This priority logic is statically defined and hard-coded within the Router submodule.

Furthermore, each type of error flag within an ECON-D is associated with a unique fast command type. When multiple flags from the same ECON-D simultaneously reach their respective thresholds, an internal flag priority scheme determines which fast command is issued. This ensures deterministic behavior in the presence of concurrent error conditions. The mapping between error flags, associated thresholds, priority levels, and corresponding fast commands is parameterized via VHDL generics within the top-level `Autorecovery` entity. An illustration of the configuration structure is provided in Figure 5.1. The figure presents the initial implementation, where the routing map was hardcoded as a generic parameter within the `Autorecovery` entity. For reference, the original generic-based routing assignment is included but commented out in the figure.

```
entity Autorecovery is
  generic (
    --! flag is triggered on successive count on corresponding error flag (column) of corresponding ECOND(row)
    cb_counter_matrix : cb_counter_matrix_t := ((3, 4, 16, 256, 8, 2),
    (3, 4, 16, 256, 8, 2),
    (3, 4, 16, 256, 8, 2),
    (3, 4, 16, 256, 8, 2),
    (3, 4, 16, 256, 8, 2),
    (3, 4, 16, 256, 8, 2),
    (3, 4, 16, 256, 8, 2),
    (3, 4, 16, 256, 8, 2),
    (3, 4, 16, 256, 8, 2),
    (3, 4, 16, 256, 8, 2),
    (3, 4, 16, 256, 8, 2),
    (3, 4, 16, 256, 8, 2));
    --! Priarity of the flags for each  ECOND (row), for flags(columns) 0 means no action , 1st has the highest prior
    cb_priority_matrix : cb_priority_matrix_t := ((1, 2, 3, 4, 0, 2),
    (1, 2, 3, 4, 0, 2),
    (1, 2, 3, 4, 0, 2),
    (1, 2, 3, 4, 0, 2),
    (1, 2, 3, 4, 0, 2),
    (1, 2, 3, 4, 0, 2),
    (1, 2, 3, 4, 0, 2),
    (1, 2, 3, 4, 0, 2),
    (1, 2, 3, 4, 0, 2),
    (1, 2, 3, 4, 0, 2),
    (1, 2, 3, 4, 0, 2),
    (1, 2, 3, 4, 0, 2));
    --! corresponding fast control flag triggered for each flag (column) and for each ECOND(row)
    cb_fcc_matrix : fcc_matrix_t := ((fcc.EBR, fcc.BCR, fcc.ECR, fcc.BCR_OCR, fcc.chipSync, fcc.linkRstEconD),
    (fcc.EBR, fcc.BCR, fcc.ECR, fcc.BCR_OCR, fcc.chipSync, fcc.linkRstEconD),
    (fcc.EBR, fcc.BCR, fcc.ECR, fcc.BCR_OCR, fcc.chipSync, fcc.linkRstEconD),
    (fcc.EBR, fcc.BCR, fcc.ECR, fcc.BCR_OCR, fcc.chipSync, fcc.linkRstEconD),
    (fcc.EBR, fcc.BCR, fcc.ECR, fcc.BCR_OCR, fcc.chipSync, fcc.linkRstEconD),
    (fcc.EBR, fcc.BCR, fcc.ECR, fcc.BCR_OCR, fcc.chipSync, fcc.linkRstEconD),
    (fcc.EBR, fcc.BCR, fcc.ECR, fcc.BCR_OCR, fcc.chipSync, fcc.linkRstEconD),
    (fcc.EBR, fcc.BCR, fcc.ECR, fcc.BCR_OCR, fcc.chipSync, fcc.linkRstEconD),
    (fcc.EBR, fcc.BCR, fcc.ECR, fcc.BCR_OCR, fcc.chipSync, fcc.linkRstEconD),
    (fcc.EBR, fcc.BCR, fcc.ECR, fcc.BCR_OCR, fcc.chipSync, fcc.linkRstEconD),
    (fcc.EBR, fcc.BCR, fcc.ECR, fcc.BCR_OCR, fcc.chipSync, fcc.linkRstEconD))
    --! maps each econd to specified fast control cahnnel output, 0 means that econd is not connected to any channel
   -- ROUTING_MAP : routing_map_t(1 to NO_OF_ECONDs) := (1, 2, 2, 3, 3, 4, 5, 6, 6, 7, 7, 8)
  );
```

Figure 5.1. Generic interface of the Autorecovery entity.

5.1.2. Declaration File

A dedicated VHDL package is utilized to encapsulate the constant definitions, user-defined record types, and helper functions required by the `Arbiter` and `Router` submodules of the Autorecovery Block. This package defines the composite types used to represent arrays of error flags and FCCs, as well as internal data structures needed for state tracking and routing logic. Among the declared elements is a function that serializes the `cb_flag_t` record type into a format suitable for flag counting operations within the `Arbiter` module. A complete listing of the VHDL package, including all type declarations and utility functions, is provided in Appendix A.2.

5.1.3. Outline of Internal Processes

5.1.3.1. Arbiter. The `Flag Counter Process` is a key process within the `Arbiter` module, responsible for monitoring error flag activity and determining whether a fast command should be generated based on threshold exceedance. The process is instantiated six times, once for each error flag, and operates synchronously with the system clock. The process behavior is defined as follows and the corresponding VHDL process is outlined below:

- The process is sensitive to the rising edge of the clock signal `clk_i`.
- Upon assertion of the active-low synchronous reset `rst_ni`, the error flag counter is reset to zero, and the associated action signal is deasserted.
- When a valid error flag input is detected, and the `disable_flags_i` signal is not asserted, the counter increments on each successive clock cycle in which the error condition persists.
- If the error condition is not continuous (i.e., the flag signal deasserts between assertions), the counter resets.
- When the counter reaches its predefined threshold, an action signal is asserted for one cycle, and the counter is reset.

```
flag_counter: for i in 0 to ERROR_FLAG_COUNT-1 generate
  process (clk_i)
    constant counter_bitsize : integer :=
        integer(ceil(log2(real(cb_counter_array(i)) + real(1)))
           );
  begin
    if rising_edge(clk_i) then
      if (rst_ni = '0') then
        error_flag_counter(i)      <= to_integer(to_unsigned(0,
            counter_bitsize));
        error_flag_action_array(i) <= '0';
      elsif(disable_flags_i = '0' and error_flag_array(i) = '1'
            and is_capture_block_valid_i = '1') then
        if (error_flag_counter(i) = cb_counter_array(i)-1) then
          error_flag_counter(i)      <= to_integer(to_unsigned
              (0, counter_bitsize));
          error_flag_action_array(i) <= '1';
        else
          error_flag_counter(i) <= to_integer(
             to_unsigned(error_flag_counter(i) + 1,
                counter_bitsize));
          error_flag_action_array(i) <= '0';
        end if;
      else
        error_flag_counter(i)      <= to_integer(to_unsigned(0,
            counter_bitsize));
        error_flag_action_array(i) <= '0';
      end if;
    end if;
  end process;
end generate flag_counter;
```

The `Valid Decision Process` logic is responsible for determining whether a fast command should be issued based on the current state of the error flags and their configured priorities. This process is sensitive to changes in the `error_flag_action_array` and utilizes a temporary variable to aggregate decisions across all monitored flags within a single clock cycle. The corresponding VHDL process is outlined below:

- For each flag, if the corresponding priority is zero, the flag is ignored.
- Otherwise, the action signal from the counter logic is passed to a variable array.
- A logical OR operation is applied to the resulting vector to determine whether any valid fast command should be generated.

```
process (error_flag_action_array)
  variable error_flag_valid_array : std_logic_vector(0 to
     ERROR_FLAG_COUNT-1) := (others => '0');
begin
  for i in 0 to ERROR_FLAG_COUNT-1 loop -- flag loop
    if (cb_priority_array(i) = 0) then
      error_flag_valid_array(i) := '0';
    else
      error_flag_valid_array(i) := error_flag_action_array(i);
    end if;
  end loop;
  is_fast_command_valid <= or error_flag_valid_array;
end process;
```

The `Fast Command Selection Process` logic identifies which specific fast command should be triggered, based on the active error flags and their associated priorities. This process also reacts to changes in the `error_flag_action_array`. The corresponding VHDL process is outlined below:

- The process iterates over all error flags, grouped by descending priority value.

- Within each priority group, it checks whether any flag is active.
- The first valid match (highest priority) sets the output fast command using the `cb_fcc_array` mapping.

```
process (error_flag_action_array)
begin
  fast_command <= (others => '0'); -- default
  for j in ERROR_FLAG_COUNT downto 1 loop -- priority loop
    for i in 0 to ERROR_FLAG_COUNT-1 loop -- flag loop
      if (cb_priority_array(i) = j and error_flag_action_array(
          i) = '1') then
        fast_command <= cb_fcc_array(i);
      end if;
    end loop;
  end loop;
end process;
```

Output Assignment. The results of these two processes, `is_fast_command_valid` and `fast_command`, are propagated to the output ports of the Arbiter module, where they are used by the downstream Router submodule logic to direct commands to the correct FCCs.

5.1.3.2. Router Submodule. `Valid Output Process` logic describes the decision logic for asserting the `is_fast_command_valid` signal for each FCC. The corresponding VHDL process is outlined below:

- The outer `generate` statement iterates over all FCCs (`NO_OF_FC_CHs`), creating a dedicated process for each channel.
- The process is sensitive to changes in `is_fast_command_valid_i`, which represents the input valid signals for all ECON-Ds.

- A `fast_command_valid_array` variable is declared to store intermediate validity information for each ECON-D. It is initialized to `'0'` for all elements.
- Inside the process, a `for` loop iterates over all ECON-Ds (`NO_OF_ECONDs`).
  - If the routing map (`ROUTING_MAP(j)`) assigns ECON-D j to channel i, then `fast_command_valid_array(j)` is updated with `is_fast_command_valid_i(j)`.
  - Otherwise, it is set to `'0'`.
- The output signal `is_fast_command_valid(i)` is assigned the logical OR reduction of the `fast_command_valid_array`, effectively determining if any ECON-D assigned to channel i has a valid fast command.

```
1  ch: for i in 1 to NO_OF_FC_CHs generate begin --flag
2      process (is_fast_command_valid_i)
3        variable fast_command_valid_array : std_logic_vector( 1
             to NO_OF_ECONDs ) := (others => '0');
4      begin
5        for j in 1 to NO_OF_ECONDs loop --flag
6          if (ROUTING_MAP(j)=i) then
7            fast_command_valid_array(j) :=
                 is_fast_command_valid_i(j);
8          else
9            fast_command_valid_array(j) := '0' ;
10         end if;
11       end loop;
12       is_fast_command_valid(i) <= or fast_command_valid_array;
13     end process;
14   end generate;
```

`Trigger Fast Command Process` logic implements the selection logic to determine which fast command should be triggered for each FCC. The corresponding VHDL process is outlined below:

- The process is sensitive to `is_fast_command_valid_i` and `fast_command_array_i`, ensuring that changes to these signals trigger reevaluation.
- The outer loop iterates over all FCCs (`NO_OF_FC_CHs`), processing one channel i at a time.
- For each channel, the corresponding output `fast_command_array(i+1)` is initialized to a default `don't care` value (`'-'`).
- A nested loop iterates over all ECON-Ds in descending order (`NO_OF_ECONDs downto 1`).
  - The convention is that a lower ECON-D number has higher priority.
  - If `is_fast_command_valid_i(j)` is asserted (`'1'`) and `ROUTING_MAP(j)` matches the current FCC (`i+1`), then `fast_command_array(i+1)` is assigned the fast command value from `fast_command_array_i(j)`.
- Due to the ordering of the nested loop, this structure ensures that the fast command from the highest priority ECON-D (lowest index j) is selected for channel `i+1`.

```
process (is_fast_command_valid_i,fast_command_array_i) begin
    for i in 0 to NO_OF_FC_CHs-1 loop --channel
      fast_command_array(i+1)    <= (others =>  '-');  --
         default
      for j in NO_OF_ECONDs downto 1 loop --priority highest is
          the lowest number of econd
        if (is_fast_command_valid_i(j)='1' and ROUTING_MAP(j)=i
           +1) then
          fast_command_array(i+1) <= fast_command_array_i (j);
        end if;
      end loop;
    end loop;
  end process;
```

The outputs of these two processes, namely `is_fast_command_valid` and `fast_command_array`, are propagated to the output ports of the `Router` submodule. From there, they are routed to the FCCs in the DAQ firmware, where they serve as actual fast command signals directed to the FE electronics.

5.2. Configurable Routing Map

A significant enhancement introduced to the Autorecovery Block is the implementation of a configurable routing map via IPbus registers. This feature introduces a flexible mechanism for dynamically assigning connections between ECON-D error flag outputs and FCCs at runtime.

This capability is critical due to the varying FE configurations across different FPGA instances within the detector. The mapping between ECON-Ds and e-link channels differs depending on the specific electrical connections and layout constraints of each system, as documented in CMS integration and connectivity schematics.

By allowing the routing logic to be initialized through software at startup, the system ensures that fast command signals can be properly directed from each ECON-D instance to the appropriate fast command output channel. This not only simplifies deployment across multiple hardware configurations but also future-proofs the design against changes in system topology or hardware upgrades.

In collider-mode operation, the routing map is configured during system initialization, allowing for per-detector customization without requiring firmware recompilation.

5.2.1. Implementation of Routing Control Registers

Initially, the routing logic for mapping ECON-D error flags to FCCs was implemented using generics within the VHDL entity as described before. However, to support dynamic reconfiguration through IPbus, this static routing mechanism was

transformed into a vector-based signal interface.

To enable this flexibility, a new array type was defined in the declaration package to represent the routing configuration:

```
type routing_fc is array(natural range <>) of std_logic_vector
   (3 downto 0);
```

This type defines an array of 4-bit vectors, each encoding the destination FCC for a given ECON-D input. At the top level of the `Autorecovery` entity, a signal of type `routing_fc` was instantiated to accommodate the routing for up to 12 ECON-Ds, corresponding to the maximum supported by the architecture.

To match the design with verification behavior, the entity was augmented with an additional input of type `ipb_reg_v`, the standard IPbus register interface type. This type represents an array of 32-bit registers accessible via software.

Given the requirement of 4 bits per ECON-D for routing and 12 ECON-Ds in total, 48 bits of configuration space were needed. Instead of allocating one full 32-bit register per ECON-D, which would be resource inefficient, two 32-bit IPbus registers were allocated, and their contents were logically partitioned into twelve 4-bit routing fields. This compact representation minimized IPbus register usage while maintaining full routing flexibility. An illustration of this partitioning scheme is provided in Figure 5.2.

Subsequently, the `Router` submodule was updated to receive its routing configuration via an input signal of type `routing_fc`, thereby replacing the static generic interface with a reconfigurable signal interface suitable for runtime updates.

```
routing_ctrl(1)  <=  ctrl_regs(0)(3 downto 0);
routing_ctrl(2)  <=  ctrl_regs(0)(7 downto 4);
routing_ctrl(3)  <=  ctrl_regs(0)(11 downto 8);
routing_ctrl(4)  <=  ctrl_regs(0)(15 downto 12);
routing_ctrl(5)  <=  ctrl_regs(0)(19 downto 16);
routing_ctrl(6)  <=  ctrl_regs(0)(23 downto 20);
routing_ctrl(7)  <=  ctrl_regs(0)(27 downto 24);
routing_ctrl(8)  <=  ctrl_regs(0)(31 downto 28);
routing_ctrl(9)  <=  ctrl_regs(1)(3 downto 0);
routing_ctrl(10) <=  ctrl_regs(1)(7 downto 4);
routing_ctrl(11) <=  ctrl_regs(1)(11 downto 8);
routing_ctrl(12) <=  ctrl_regs(1)(15 downto 12);
```

Figure 5.2. Connection of routing map signals and IPbus control registers.

5.3. Cocotb-Based Verification Environment

Cocotb [27] is a coroutine-based cosimulation testbench framework designed to verify VHDL and SystemVerilog RTL designs using Python. In this study, the *Questa*^{TM} *SIM 23.1* simulator [28] was employed as the simulation backend. The combination of Cocotb and Questa SIM was selected for its rapid development cycle, powerful verification capabilities, and comprehensive support for coverage analysis, making it a suitable choice for verifying both ASIC and FPGA designs.

The Design Under Test (DUT) is instantiated directly as the top-level module within the simulator, eliminating the need for wrapper modules or additional structural logic. Stimulus is applied to the DUT's inputs and responses are monitored via Python test routines. Makes use of the VHDL Procedural Interface (VHPI) or Verilog Procedural Interface (VPI) as shown in Figure 5.3. This abstraction significantly enhances the readability, maintainability, and reusability of testbenches.

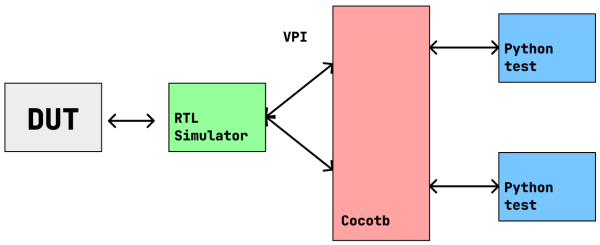


Figure 5.3. Cocotb interface.

In Cocotb, a test is defined as a Python coroutine function. Execution alternates between the simulator advancing time and the Python runtime executing commands. The `await` keyword is used to yield control from Python back to the simulator. Multiple coroutine threads can be spawned within a test to simulate concurrent behaviors, enabling effective modeling of asynchronous systems and concurrent interface protocols.

5.3.1. Corner Cases

A key component of the verification process is the validation of the Autorecovery Block under critical corner cases. The design incorporates two levels of prioritization: *flag-based priority* and *ECON-D-based priority*, both of which must be thoroughly tested to ensure deterministic and correct behavior.

Case 1: Simultaneous Flags from Different ECON-Ds to the Same FCC.

- *Description:* Multiple input channels (ECON-Ds) that are routed to the same FCC assert error flags simultaneously.
- *Example:* ECON-D 2 and ECON-D 3 each raise flags that reach their defined thresholds at the same clock cycle and are both connected to FCC channel 2.
- *Expected Behavior:* FCC 2 should output the fast command corresponding to ECON-D 2, which has higher channel priority than ECON-D 3. This should

occur regardless of the relative priority levels of the individual error flags.

Case 2: Simultaneous Flags of Different Types from the Same ECON-D.

- *Description:* A single ECON-D raises multiple error flags simultaneously, each corresponding to a distinct fault condition.
- *Example:* ECON-D 1 asserts both the `econd_rr` and `timeout` flags after meeting the configured thresholds in the same clock cycle.
- *Expected Behavior:* The `econd_rr` flag, having the higher priority, should take precedence and trigger the associated fast command on the designated FCC.

In addition to the corner case tests, functional verification was conducted for auxiliary control signals such as the active-low reset signal (`rst_ni`), the disable control for each ECON-D channel, and the output validity indicator. These tests ensure comprehensive coverage of the control logic and its integration within the system.

5.3.2. Verification Structure

The verification strategy employed for the Autorecovery Block leverages the `cocotb` framework and is implemented entirely in Python. A central coroutine, `set_flags`, is responsible for generating test stimuli by asserting various error flags. This coroutine is defined using the `async def` syntax, which facilitates asynchronous execution by allowing the function to yield control back to the simulator, enabling simulation time to advance only when specific events (e.g., clock edges) occur.

Upon completion of the stimulus generation over a predefined number of clock cycles, the expected fast commands are compared against the actual outputs from the FCCs. Any mismatch between the expected and observed outputs triggers a test failure, and simulation is halted to prevent error masking.

To ensure a robust validation process, the following coverage metrics are also utilized:

- *Functional Coverage:* Coverpoints are defined for key functional elements, including ECON-D indices, FCCs, error flag types, threshold levels, priority encodings, and the resulting fast commands. This ensures that every possible operational condition is exercised at least once during testing.
- *Code Coverage:* Metrics such as branch, condition, expression, and statement coverage are collected via the Questa SIM simulator. These options are enabled in the `cocotb` Makefile to provide a comprehensive analysis of the HDL code execution paths.

The specific configuration of threshold values, priorities, and corresponding hexadecimal equivalents of fast commands used in this test campaign are provided in Table 5.1. The routing map utilized for mapping ECON-Ds to FCCs is outlined in Table 5.2.

Table 5.1. Error flag informations for verification.

Flag	*Threshold*	*Priority*	*Fast Command*
econd_rr	3	1	D1
timeout	4	2	1D
no_idle	16	3	A9
bx_mismatch	256	4	95
ev_mismatch	8	0	D2
buf_overflow	2	2	B8

Table 5.2. Routing map for verification.

ECON-D	*FCC*
1	1
2	2
3	2
4	3
5	3
6	4
7	5
8	6
9	6
10	7
11	7
12	8

5.3.3. Test Results

Simulation results were obtained using the Questa SIM simulator, and waveform plots corresponding to the verification corner cases were extracted for analysis.

Figure 5.4 illustrates one such scenario: both `ECON-D 2` and `ECON-D 3` are mapped to the same FCC2. `ECON-D 2` counts the `no_idle` flag, which has a threshold of 16, while `ECON-D 3` counts the `timeout` flag, which has a threshold of 4. Once both thresholds are reached, fast commands are generated simultaneously by both channels. The `timeout` flag, assigned a priority of 2, corresponds to the fast command `0x1D`, whereas the `no_idle` flag, with a priority of 3, triggers `0xA9`. Despite the `timeout` flag having a higher flag-level priority, the waveform clearly shows `0xA9` on FCC2. This confirms that ECON-D-level priority takes precedence over flag-level priority, thereby giving `ECON-D 2` higher arbitration rights over `ECON-D 3`.

Subsequently, after four additional clock cycles (the threshold of the `timeout` flag), the `0x1D` command is seen in the output, verifying the persistent counting of the `timeout` flag. Key transitions and signals of interest are annotated directly on the waveform for clarity.

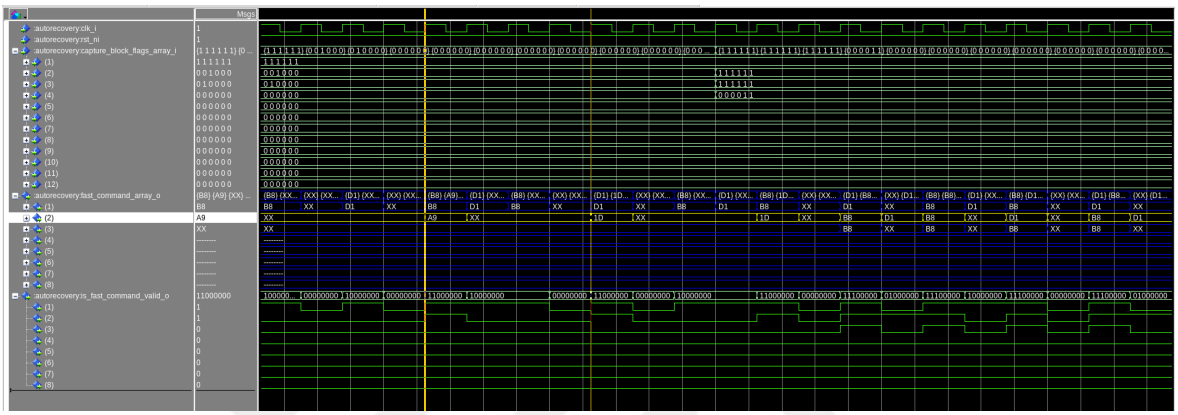


Figure 5.4. Waveform demonstrating the priority arbitration between ECON-D 2 and ECON-D 3 routed to FCC2. Despite the `timeout` flag having higher priority over `no_idle`, ECON-D 2's fast command `0xA9` is seen first due to its ECON-D-level priority.

In the simulation illustrated in Figure 5.5, the behavior of the Autorecovery Block is examined in a scenario where multiple error flags within a single `ECON-D` module simultaneously reach their configured thresholds. Specifically, `ECON-D 1` is connected to FCC 1 and is configured to generate two types of error flags: `econd_rr` and `timeout`. These error flags are configured with threshold values of 3 and 4, respectively.

During simulation, both error flags meet their thresholds within the same evaluation cycle. According to the configuration parameters provided to the Autorecovery Block, the `econd_rr` flag has a higher priority (priority level 1) compared to the `timeout` flag (priority level 2). Each flag is also associated with a corresponding fast command: `0xD1` for `econd_rr` and `0x1D` for `timeout`. The waveform confirms that the Autorecovery Block correctly asserts the fast command associated with the higher-priority flag, `0xD1`, on the output of FCC 1. This verifies that intra-ECON-D flag prioritization

is correctly implemented. Additionally, both fast commands appear on the output at their respective threshold completion cycles, further validating the expected timing behavior.

This test case effectively demonstrates the correct operation of the priority arbitration mechanism within a single ECON-D, ensuring that higher-priority error conditions are addressed first in real-time recovery scenarios.

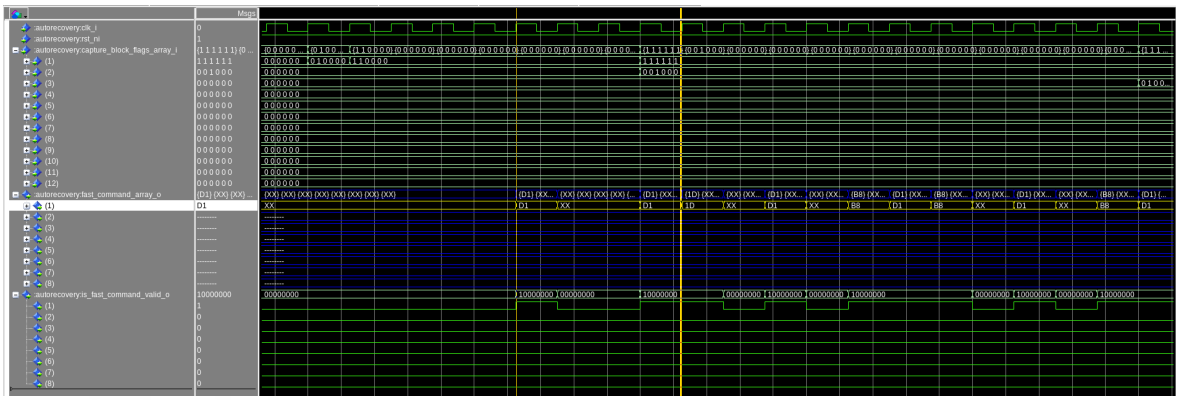


Figure 5.5. Simulation waveform showing flag-level prioritization in `ECON-D 1`. Both `econd_rr` and `timeout` flags reach their respective thresholds simultaneously, yet the fast command corresponding to the higher-priority `econd_rr` (`0xD1`) is issued.

The simulation results shown in Figure 5.6 illustrate the functional behavior of the Autorecovery Block under the influence of key control signals: the `disable_flags_i` signal (used to mask specific `ECON-D` channels), the global synchronous `reset`, and the `is_capture_block_valid_array_i` flag that reflects the validity of the Capture Block.

Initially, multiple fast commands are generated from the input flags that reach their respective thresholds. When the `disable_flags_i` signal is asserted for a specific `ECON-D` channel, it prevents further flag accumulation for that channel, effectively suppressing additional fast command generation. Upon de-assertion of this signal, the system resumes counting from the initial state, as demonstrated in the waveform.

The impact of the global `reset` signal is also evaluated. When activated, it reinitializes all internal counters and output channels, halting any ongoing flag counting or fast command emission. Once de-asserted, the system resumes normal operation, allowing flags to be counted again based on current inputs.

In addition, the `is_capture_block_valid_array_i` signal is tested to simulate scenarios where incoming data from the specific ECON-D from the Capture Block is deemed invalid. As shown, any `ECON-D` whose data is marked invalid will not contribute to the error flag accumulation, and thus, no fast commands are generated from that input regardless of the flag state.

This simulation confirms that all auxiliary control signals are functioning as intended and that the Autorecovery Block behaves reliably under dynamic runtime conditions.

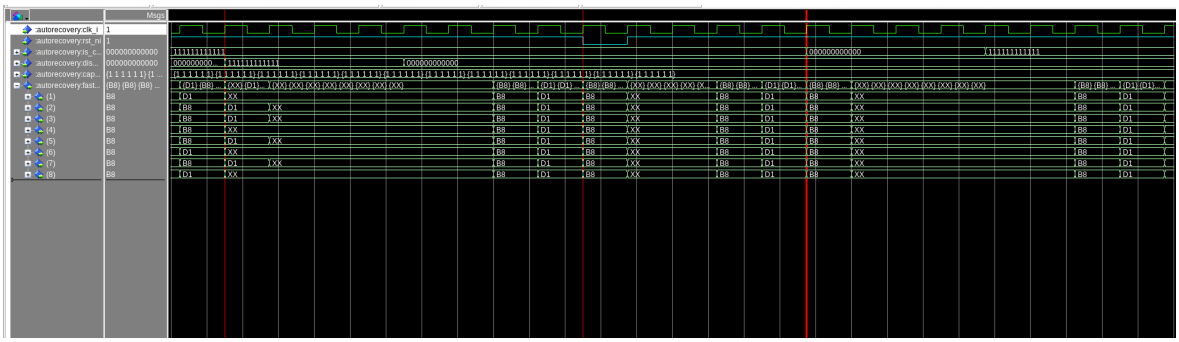


Figure 5.6. Waveform showing the effect of key control signals on the Autorecovery Block logic. Assertion of `disable_flags_i` halts fast command generation for selected `ECON-D` channels; global `reset` clears counters and outputs; `is_capture_block_valid_array_i` suppresses counting for invalid Capture Blocks.

Figure 5.7 illustrates the verification results for the fast command outputs during the simulation. This stage of the testbench involves asserting whether the output fast commands generated by the Autorecovery Block align with the expected values calculated based on the configured thresholds, priorities, and routing map.

The test successfully completes without any assertion failures, indicating full functional correctness of the design under the tested conditions. Each fast command emitted by the Autorecovery Block is checked against its expected counterpart within the verification environment. The assertions are executed in real time during simulation, and any mismatch would have immediately triggered a test failure and halted execution.

Additionally, as shown in the transcript window, the flag corresponding to `ev_mismatch` does not produce any fast command output. This is consistent with the design specification, as the priority assigned to this flag is set to zero. Flags with zero priority are effectively masked from triggering fast commands, regardless of whether their thresholds are met.

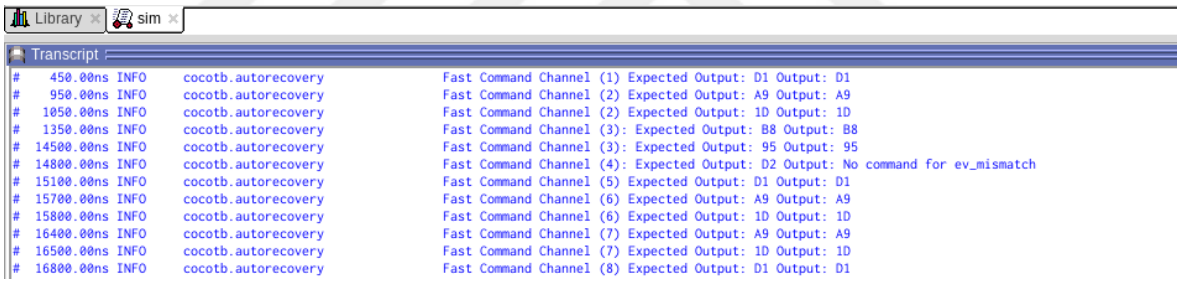


Figure 5.7. Transcript window output. Assertions compare output fast commands against expected values.

5.3.4. Coverage Results

To ensure exhaustive verification, the simulation was extended to cover all functional scenarios and corner cases of the Autorecovery Block. The testbench was executed with complete stimulus coverage, including combinations of error flag thresholds, ECON-D priority routing, reset behavior, and disable control signals.

Figure 5.8 presents the code coverage summary obtained using the Questa SIM simulator. The summary includes metrics for statement coverage, branch coverage,

and condition coverage. As observed, all branches and conditional paths within the Autorecovery Block were exercised during the simulation, achieving 100% code coverage across all categories.

This result confirms that the verification environment effectively stimulated the entire design, validating not only the nominal functionality but also the handling of edge cases, such as conflicting flag priorities and dynamic reconfiguration of routing. The complete coverage score further affirms the correctness and robustness of the Autorecovery Block logic under varying input and control scenarios.

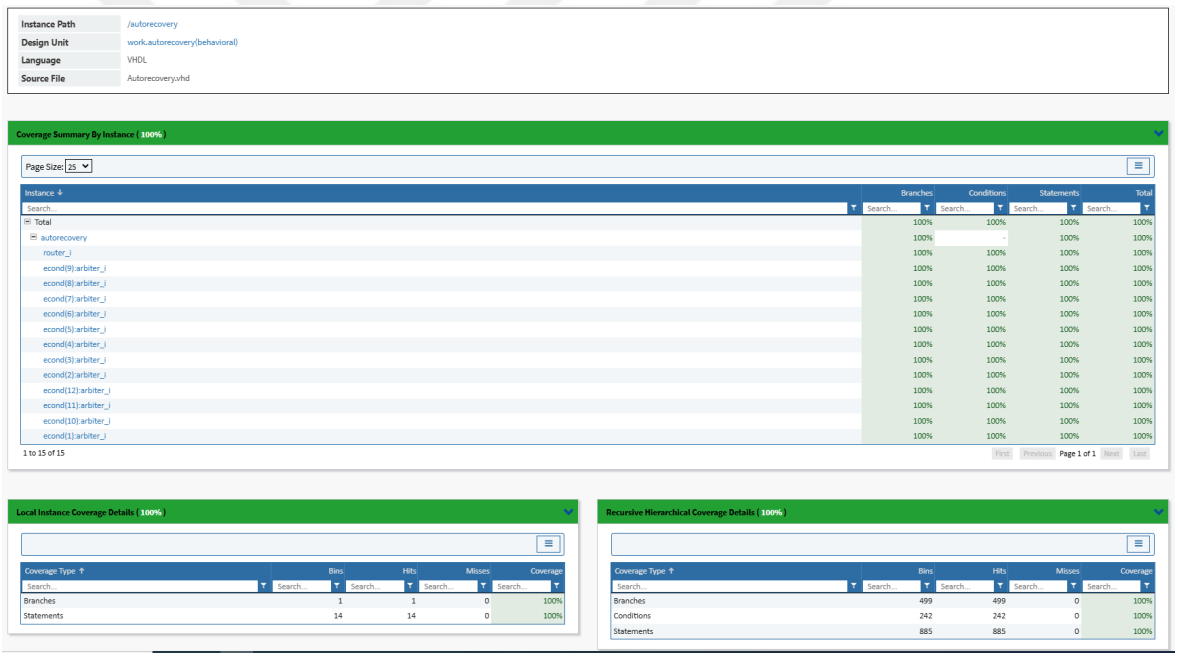


Figure 5.8. Code coverage summary from Questa SIM for the Autorecovery Block. All statements, branches, and conditions were exercised during simulation, resulting in 100% total coverage.

In addition to standard code coverage metrics, a Focused Expression Coverage (FEC) analysis was conducted using Questa SIM to assess the effectiveness of the verification environment in exercising complex boolean expressions within the Autorecovery Block. FEC is particularly valuable in identifying whether each sub-expression within compound conditions has been evaluated for both true and false outcomes.

Figure 5.9 illustrates the FEC results from the simulation. As shown, all targeted expressions were fully evaluated, demonstrating that the verification testbench provided a comprehensive set of input combinations necessary to traverse all logical decision paths. This further reinforces the completeness of the functional coverage and the robustness of the test infrastructure.

FEC Condition rows	FEC Conditions hit	FEC Conditions missed	FEC Condition %	FEC Condition graph
62	62	0	100.00%	
23	23	0	100.00%	
23	23	0	100.00%	
23	23	0	100.00%	
23	23	0	100.00%	
23	23	0	100.00%	
23	23	0	100.00%	
23	23	0	100.00%	
23	23	0	100.00%	
23	23	0	100.00%	
23	23	0	100.00%	
23	23	0	100.00%	
23	23	0	100.00%	
23	23	0	100.00%	
2	2	0	100.00%	

Figure 5.9. FEC report generated by Questa SIM for the Autorecovery Block. Each boolean sub-expression within compound conditions was fully exercised, confirming exhaustive evaluation.

Coverpoints were defined for all relevant signals and configuration parameters, including ECON-D indices, FCCs, error flags, priority levels, and threshold crossings. Figure 5.10 presents the resulting functional coverage summary. As observed, all defined bins were hit at least once, confirming that the verification environment was capable of stimulating every significant condition and configuration expected in real operation. This outcome demonstrates the thoroughness of the testbench in validating both nominal and corner-case behaviors.



Figure 5.10. Functional coverage summary of the Autorecovery Block. All user-defined coverpoint bins were exercised at least once during the simulation.

6. TESTING THE UPDATED AUTORECOVERY WITH EMP ON SERENITY

The enhanced Autorecovery Block, featuring a configurable routing map implemented via IPbus control registers, was evaluated using the EMP system on the Serenity FPGA platform at Boğaziçi University. Prior to hardware integration, the behavioral functionality of the routing map was extensively verified using Cocotb and QuestaSim, as detailed in Chapter 5. This systematic verification process enabled direct validation of the critical routing functionalities—specifically, the accurate distribution of ECON-D error flags to the designated FCC outputs—in a real hardware environment.

6.1. EMP Payload Integration

To validate the enhanced Autorecovery Block functionality, integration with the EMP payload was performed. Inputs to the Autorecovery Block originate from the EMP payload's lpGBT channels and are mapped to internal Autorecovery Block signals: `capture_block_flags_array` (type `cb_flag_array_t`) and `is_capture_block_valid_array` (type `std_logic_vector(1 to NO_OF_ECONDs)`). These inputs precisely replicate those provided by the Event Builder in the operational system. Similarly, Autorecovery Block outputs, comprising fast command signals, are routed to EMP payload output channels for data capture and further analysis using the EMP software. The VHDL code implementing these connections is illustrated below:

```
1  flags_array_connections: for i in 1 to NO_OF_ECONDs generate
2    capture_block_flags_array(i).econd_rr     <= d(i-1).data(0)
       ;
3    capture_block_flags_array(i).timeout      <= d(i-1).data(1)
       ;
4    capture_block_flags_array(i).no_idle      <= d(i-1).data(2)
       ;
5    capture_block_flags_array(i).bx_mismatch  <= d(i-1).data(3)
       ;
6    capture_block_flags_array(i).ev_mismatch  <= d(i-1).data(4)
       ;
7    capture_block_flags_array(i).buf_overflow <= d(i-1).data(5)
       ;
8    is_capture_block_valid_array(i) <= d(i-1).data(6);
9    disable_flags_array(i) <= d(i-1).data(7);
10 end generate;
11
12 fc_connections : for i in 1 to NO_OF_FC_CHs generate
13   q(i-1).data(7 downto 0) <= fast_command_array(i);
14   q(i-1).data(8) <= is_fast_command_valid(i);
15 end generate;
```

IPbus communication within the payload entity is managed via an IPbus select fabric, handling address decoding and signal distribution. The address decoding logic is generated automatically from XML-based definitions through the command ipbb ipbus gendecoders. At the top-level `Autorecovery` entity, the IPbus register fabric is instantiated with 3 control and 12 status registers as shown below:

```
ipbus_reg: entity work.ipbus_syncreg_v
  generic map(
      N_CTRL => 3,
      N_STAT => 12,
      SWAP_ORDER => false
  )
  port map(
      clk     => clk_ipbus,
      rst     => rst_ipbus,
      ipb_in  => ipb_in,
      ipb_out => ipb_out,

      slv_clk => clk_ipbus,
      d => stat_regs,
      q => ctrl_regs
  );
```

The control registers serve two primary functions: setting the routing map of ECON-D error flags to FCCs, and disabling individual ECON-D modules when required. For efficient resource utilization, each 32-bit control register is subdivided into 4-bit segments, each configuring the routing of a specific ECON-D module, as shown below:

```
-- Control Regs
routing_ctrl(1)  <= ctrl_regs(0)(3 downto 0);
routing_ctrl(2)  <= ctrl_regs(0)(7 downto 4);
routing_ctrl(3)  <= ctrl_regs(0)(11 downto 8);
routing_ctrl(4)  <= ctrl_regs(0)(15 downto 12);
routing_ctrl(5)  <= ctrl_regs(0)(19 downto 16);
routing_ctrl(6)  <= ctrl_regs(0)(23 downto 20);
routing_ctrl(7)  <= ctrl_regs(0)(27 downto 24);
routing_ctrl(8)  <= ctrl_regs(0)(31 downto 28);
routing_ctrl(9)  <= ctrl_regs(1)(3 downto 0);
routing_ctrl(10) <= ctrl_regs(1)(7 downto 4);
routing_ctrl(11) <= ctrl_regs(1)(11 downto 8);
routing_ctrl(12) <= ctrl_regs(1)(15 downto 12);

gen_disable_regs: for i in 0 to 11 generate
  disable_flags_array(i+1) <= ctrl_regs(2)(i);
end generate gen_disable_regs;
```

To monitor real-time system performance, status registers provide continuous readouts of the internal error flag counters from the Autorecovery Block's Arbiter module. Each status register is explicitly mapped to these counters to facilitate rapid verification and debugging, as demonstrated below:

```
-- Status Regs
gen_stat_regs_flags: for i in 0 to 11 generate
  stat_regs(i)(3 downto 0)   <= std_logic_vector(to_unsigned(
     error_flag_counter_o(i+1)(0),4));
  stat_regs(i)(8 downto 4)   <= std_logic_vector(to_unsigned(
     error_flag_counter_o(i+1)(1),5));
  stat_regs(i)(14 downto 9)  <= std_logic_vector(to_unsigned(
     error_flag_counter_o(i+1)(2),6));
  stat_regs(i)(23 downto 15) <= std_logic_vector(to_unsigned(
     error_flag_counter_o(i+1)(3),9));
  stat_regs(i)(27 downto 24) <= std_logic_vector(to_unsigned(
     error_flag_counter_o(i+1)(4),4));
  stat_regs(i)(31 downto 28) <= std_logic_vector(to_unsigned(
     error_flag_counter_o(i+1)(5),4));
end generate gen_stat_regs_flags;
```

This comprehensive integration ensures precise control and monitoring of the Autorecovery Block, thereby enabling robust hardware verification and reliable operation within the overall EMP payload system.

6.1.1. IPbus Register Mapping and Packaging Project

Precise IPbus address table XML files were developed to ensure proper integration and accurate register addressing within the EMP system. The top-level EMP address table divides the IPbus hierarchy into six distinct nodes, as illustrated below:

```
<node id="TOP">
    <node id="info" module="file://emp_info.xml" address="0x0"/
       >
    <node id="ctrl" module="file://emp_ctrl.xml" address="0x200
       "/>
    <node id="ttc" module="file://emp_ttc.xml" address="0x400"/
       >
    <node id="datapath" module="file://emp_datapath.xml"
       address="0x2000"/>
    <node id="slink" module="file://emp_slink.xml" address="0
       x1000000" />
    <node id="payload" module="file://emp_payload.xml" address=
       "0x80000000" fwinfo="endpoint;width=31"/>
</node>
```

The `payload` node was subsequently extended by introducing an additional IPbus address node designated as `ipregtest`, as shown below. Address ranges for endpoint nodes adhere to IPbus address decoder guidelines, which require the base address to be aligned to multiples of 2^N, where N is the address width.

```
<node id="payload" address="0x80000000">
    <node id="generator" address="0x0" module="file://
       emp_slink_generator.xml" fwinfo="endpoint;width=0"/>
    <node id="ipregtest" address="0x20" module="file://
       ipregtest.xml" fwinfo="endpoint;width=4"/>
</node>
```

Within the `ipregtest` node, detailed node definitions—including unique identifiers, addresses, and corresponding bit masks—were established to precisely map control and status registers. These mappings are shown in Figure 6.1.

```
<node id="ipregtest">
    <node id="ctrl1"        address="0x0" mask="0x0000000F"  />
    <node id="ctrl2"        address="0x0" mask="0x000000F0"  />
    <node id="ctrl3"        address="0x0" mask="0x00000F00"  />
    <node id="ctrl4"        address="0x0" mask="0x0000F000"  />
    <node id="ctrl5"        address="0x0" mask="0x000F0000"  />
    <node id="ctrl6"        address="0x0" mask="0x00F00000"  />
    <node id="ctrl7"        address="0x0" mask="0x0F000000"  />
    <node id="ctrl8"        address="0x0" mask="0xF0000000"  />
    <node id="ctrl9"        address="0x1" mask="0x0000000F"  />
    <node id="ctrl10"       address="0x1" mask="0x000000F0" />
    <node id="ctrl11"       address="0x1" mask="0x00000F00" />
    <node id="ctrl12"       address="0x1" mask="0x0000F000" />

    <node id="disable1"       address="0x2" mask="0x00000001" />
    <node id="disable2"       address="0x2" mask="0x00000002" />
    <node id="disable3"       address="0x2" mask="0x00000004" />
    <node id="disable4"       address="0x2" mask="0x00000008" />
    <node id="disable5"       address="0x2" mask="0x00000010" />
    <node id="disable6"       address="0x2" mask="0x00000020" />
    <node id="disable7"       address="0x2" mask="0x00000040" />
    <node id="disable8"       address="0x2" mask="0x00000080" />
    <node id="disable9"       address="0x2" mask="0x00000100" />
    <node id="disable10"       address="0x2" mask="0x00000200" />
    <node id="disable11"       address="0x2" mask="0x00000400" />
    <node id="disable12"       address="0x2" mask="0x00000800" />

    <node id="econd1_rr"        address="0x10" mask="0x0000000F" />
    <node id="econd1_t"        address="0x10" mask="0x000001F0" />
    <node id="econd1_i"        address="0x10" mask="0x00007E00" />
    <node id="econd1_bx"        address="0x10" mask="0x00FF8000" />
    <node id="econd1_ev"        address="0x10" mask="0x0F000000" />
    <node id="econd1_of"        address="0x10" mask="0xF0000000" />

    <node id="econd2_rr"        address="0x11" mask="0x0000000F" />
    <node id="econd2_t"        address="0x11" mask="0x000001F0" />
    <node id="econd2_i"        address="0x11" mask="0x00007E00" />
    <node id="econd2_bx"        address="0x11" mask="0x00FF8000" />
    <node id="econd2_ev"        address="0x11" mask="0x0F000000" />
    <node id="econd2_of"        address="0x11" mask="0xF0000000" />

    <node id="econd3_rr"        address="0x12" mask="0x0000000F" />
    <node id="econd3_t"        address="0x12" mask="0x000001F0" />
    <node id="econd3_i"        address="0x12" mask="0x00007E00" />
    <node id="econd3_bx"        address="0x12" mask="0x00FF8000" />
    <node id="econd3_ev"        address="0x12" mask="0x0F000000" />
    <node id="econd3_of"        address="0x12" mask="0xF0000000" />

    <node id="econd4_rr" address="0x13" mask="0x0000000F" />
    <node id="econd4_t" address="0x13" mask="0x000001F0" />
    <node id="econd4_i" address="0x13" mask="0x00007E00" />
    <node id="econd4_bx" address="0x13" mask="0x00FF8000" />
    <node id="econd4_ev" address="0x13" mask="0x0F000000" />
    <node id="econd4_of" address="0x13" mask="0xF0000000" />

    <node id="econd5_rr" address="0x14" mask="0x0000000F" />
    <node id="econd5_t" address="0x14" mask="0x000001F0" />
    <node id="econd5_i" address="0x14" mask="0x00007E00" />
    <node id="econd5_bx" address="0x14" mask="0x00FF8000" />
    <node id="econd5_ev" address="0x14" mask="0x0F000000" />
    <node id="econd5_of" address="0x14" mask="0xF0000000" />

    <node id="econd6_rr" address="0x15" mask="0x0000000F" />
    <node id="econd6_t" address="0x15" mask="0x000001F0" />
    <node id="econd6_i" address="0x15" mask="0x00007E00" />
    <node id="econd6_bx" address="0x15" mask="0x00FF8000" />
    <node id="econd6_ev" address="0x15" mask="0x0F000000" />
    <node id="econd6_of" address="0x15" mask="0xF0000000" />

    <node id="econd7_rr" address="0x16" mask="0x0000000F" />
    <node id="econd7_t" address="0x16" mask="0x000001F0" />
    <node id="econd7_i" address="0x16" mask="0x00007E00" />
    <node id="econd7_bx" address="0x16" mask="0x00FF8000" />
    <node id="econd7_ev" address="0x16" mask="0x0F000000" />
    <node id="econd7_of" address="0x16" mask="0xF0000000" />

    <node id="econd8_rr" address="0x17" mask="0x0000000F" />
    <node id="econd8_t" address="0x17" mask="0x000001F0" />
    <node id="econd8_i" address="0x17" mask="0x00007E00" />
    <node id="econd8_bx" address="0x17" mask="0x00FF8000" />
    <node id="econd8_ev" address="0x17" mask="0x0F000000" />
    <node id="econd8_of" address="0x17" mask="0xF0000000" />

    <node id="econd9_rr" address="0x18" mask="0x0000000F" />
    <node id="econd9_t" address="0x18" mask="0x000001F0" />
    <node id="econd9_i" address="0x18" mask="0x00007E00" />
    <node id="econd9_bx" address="0x18" mask="0x00FF8000" />
    <node id="econd9_ev" address="0x18" mask="0x0F000000" />
    <node id="econd9_of" address="0x18" mask="0xF0000000" />

    <node id="econd10_rr" address="0x19" mask="0x0000000F" />
    <node id="econd10_t" address="0x19" mask="0x000001F0" />
    <node id="econd10_i" address="0x19" mask="0x00007E00" />
    <node id="econd10_bx" address="0x19" mask="0x00FF8000" />
    <node id="econd10_ev" address="0x19" mask="0x0F000000" />
    <node id="econd10_of" address="0x19" mask="0xF0000000" />

    <node id="econd11_rr" address="0x1A" mask="0x0000000F" />
    <node id="econd11_t" address="0x1A" mask="0x000001F0" />
    <node id="econd11_i" address="0x1A" mask="0x00007E00" />
    <node id="econd11_bx" address="0x1A" mask="0x00FF8000" />
    <node id="econd11_ev" address="0x1A" mask="0x0F000000" />
    <node id="econd11_of" address="0x1A" mask="0xF0000000" />

    <node id="econd12_rr" address="0x1B" mask="0x0000000F" />
    <node id="econd12_t" address="0x1B" mask="0x000001F0" />
    <node id="econd12_i" address="0x1B" mask="0x00007E00" />
    <node id="econd12_bx" address="0x1B" mask="0x00FF8000" />
    <node id="econd12_ev" address="0x1B" mask="0x0F000000" />
    <node id="econd12_of" address="0x1B" mask="0xF0000000" />
</node>
```

Figure 6.1. Autorecovery Block IPbus register address table.

Upon completion of these definitions, IPBB commands were executed to generate the IPbus address decoder logic, perform Vivado synthesis and implementation, and subsequently package the bitstream. This resulted in the production of the final FPGA bitfile alongside the necessary IPbus address tables, ready for deployment onto the FPGA hardware.

6.1.2. Creating lpGBT Input File for Autorecovery Block Test

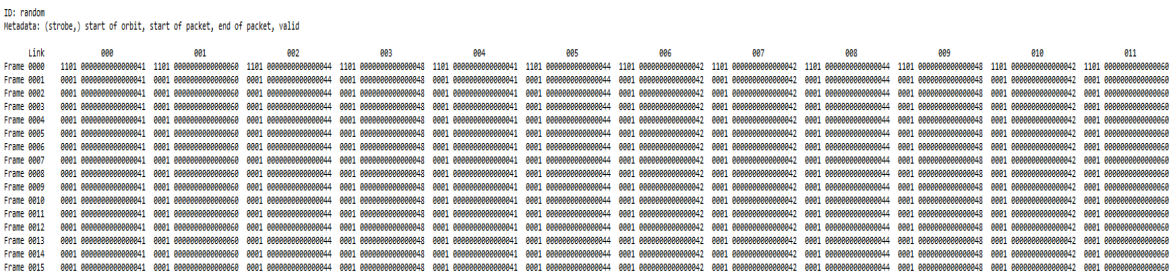
To thoroughly validate the Autorecovery Block, specific lpGBT input files were generated. These files emulate realistic detector conditions and input behaviors necessary for comprehensive hardware testing. Each input buffer comprises four quads, providing 72 channels of lpGBT data. The data structure used for these channels is defined by a custom record type, `lword`, as declared in the EMP VHDL definition file:

```
constant LWORD_WIDTH : integer := 64;

type lword is record
  data            : std_logic_vector(LWORD_WIDTH - 1 downto 0);
  valid           : std_logic;
  start           : std_logic;
  start_of_orbit  : std_logic;
  strobe          : std_logic;
  last            : std_logic;
end record;

type ldata is array(natural range <>) of lword;
```

The Autorecovery Block utilizes 12 lpGBT input channels, corresponding directly to the number of ECON-D modules, and generates outputs across 8 FCCs. The `data` field within the `lword` record structure is specifically utilized to transmit ECON-D error flags.

Test scenarios were carefully constructed, incorporating critical conditions and edge cases previously identified and detailed in Chapter 5. The EMP system accepts text-based input files, each consisting of 1024 frames. These frames emulate detector operation by incorporating essential control signals such as the *start of orbit*, *start and end of packet indicators*, and *validity bits*.

In a typical test file, the first frame indicates both the start of orbit and the start of a packet. All subsequent frames are marked as valid, while the final frame signifies the end of the packet. An example of the lpGBT input test file structure is illustrated in Figure 6.2.


ID: random
Metadata: (strobe,) start of orbit, start of packet, end of packet, valid

Link	000	001	002	003	004	005	006	007	008	009	010	011
Frame 0000	1101 00000000000041	1101 00000000000060	1101 00000000000044	1101 00000000000048	1101 00000000000041	1101 00000000000044	1101 00000000000042	1101 00000000000044	1101 00000000000048	1101 00000000000042	1101 00000000000060
Frame 0001	0001 00000000000041	0001 00000000000060	0001 00000000000044	0001 00000000000048	0001 00000000000041	0001 00000000000044	0001 00000000000042	0001 00000000000044	0001 00000000000048	0001 00000000000042	0001 00000000000060
Frame 0002	0001 00000000000041	0001 00000000000060	0001 00000000000044	0001 00000000000048	0001 00000000000041	0001 00000000000044	0001 00000000000042	0001 00000000000044	0001 00000000000048	0001 00000000000042	0001 00000000000060
Frame 0003	0001 00000000000041	0001 00000000000060	0001 00000000000044	0001 00000000000048	0001 00000000000041	0001 00000000000044	0001 00000000000042	0001 00000000000044	0001 00000000000048	0001 00000000000042	0001 00000000000060
Frame 0004	0001 00000000000041	0001 00000000000060	0001 00000000000044	0001 00000000000048	0001 00000000000041	0001 00000000000044	0001 00000000000042	0001 00000000000044	0001 00000000000048	0001 00000000000042	0001 00000000000060
Frame 0005	0001 00000000000041	0001 00000000000060	0001 00000000000044	0001 00000000000048	0001 00000000000041	0001 00000000000044	0001 00000000000042	0001 00000000000044	0001 00000000000048	0001 00000000000042	0001 00000000000060
Frame 0006	0001 00000000000041	0001 00000000000060	0001 00000000000044	0001 00000000000048	0001 00000000000041	0001 00000000000044	0001 00000000000042	0001 00000000000044	0001 00000000000048	0001 00000000000042	0001 00000000000060
Frame 0007	0001 00000000000041	0001 00000000000060	0001 00000000000044	0001 00000000000048	0001 00000000000041	0001 00000000000044	0001 00000000000042	0001 00000000000044	0001 00000000000048	0001 00000000000042	0001 00000000000060
Frame 0008	0001 00000000000041	0001 00000000000060	0001 00000000000044	0001 00000000000048	0001 00000000000041	0001 00000000000044	0001 00000000000042	0001 00000000000044	0001 00000000000048	0001 00000000000042	0001 00000000000060
Frame 0009	0001 00000000000041	0001 00000000000060	0001 00000000000044	0001 00000000000048	0001 00000000000041	0001 00000000000044	0001 00000000000042	0001 00000000000044	0001 00000000000048	0001 00000000000042	0001 00000000000060
Frame 0010	0001 00000000000041	0001 00000000000060	0001 00000000000044	0001 00000000000048	0001 00000000000041	0001 00000000000044	0001 00000000000042	0001 00000000000044	0001 00000000000048	0001 00000000000042	0001 00000000000060
Frame 0011	0001 00000000000041	0001 00000000000060	0001 00000000000044	0001 00000000000048	0001 00000000000041	0001 00000000000044	0001 00000000000042	0001 00000000000044	0001 00000000000048	0001 00000000000042	0001 00000000000060
Frame 0012	0001 00000000000041	0001 00000000000060	0001 00000000000044	0001 00000000000048	0001 00000000000041	0001 00000000000044	0001 00000000000042	0001 00000000000044	0001 00000000000048	0001 00000000000042	0001 00000000000060
Frame 0013	0001 00000000000041	0001 00000000000060	0001 00000000000044	0001 00000000000048	0001 00000000000041	0001 00000000000044	0001 00000000000042	0001 00000000000044	0001 00000000000048	0001 00000000000042	0001 00000000000060
Frame 0014	0001 00000000000041	0001 00000000000060	0001 00000000000044	0001 00000000000048	0001 00000000000041	0001 00000000000044	0001 00000000000042	0001 00000000000044	0001 00000000000048	0001 00000000000042	0001 00000000000060
Frame 0015	0001 00000000000041	0001 00000000000060	0001 00000000000044	0001 00000000000048	0001 00000000000041	0001 00000000000044	0001 00000000000042	0001 00000000000044	0001 00000000000048	0001 00000000000042	0001 00000000000060

Figure 6.2. Example of an lpGBT input test file. The file consists of 1024 frames emulating detector signals with control and data fields.

The input data is expressed in hexadecimal format, with bits 0 through 5 encoding specific error flags, while bit 6 represents the valid data indicator. For instance, the hexadecimal value `0x48` corresponds to the binary pattern `0b01100010`, indicating simultaneous activation of timeout and buffer overflow flags and data validity.

6.1.3. Register Read/Writes

To configure the Autorecovery Block's routing map and monitor the real-time status of error flags within the system, Python scripts leveraging the `uHAL` framework were developed. The IPbus control and status registers were accessed using `uHAL` based on the connection XML file `my_connections.xml`, illustrated below:

```
<?xml version="1.0" encoding="UTF-8"?>
<connections>
  <connection id="x0" uri="ipbuspcie-2.0:///dev/serenity_pcie/
     x0/h2c,/dev/serenity_pcie/x0/c2h"
              address_table="file://./autorecovery_0/addrtab/
                 top_emp.xml"/>
</connections>
```

This XML configuration enabled structured access to the relevant hardware nodes for register manipulation. A Python script was prepared to write to the IPbus control registers, thereby establishing the required routing configuration and testing the functionality of ECON-D disable signals. Following this setup, the script is utilized to read the real-time error flags for each ECON-D channel through status registers. The script is included in Appendix B.1. This systematic approach provided an efficient means to validate the routing logic and ECON-D disable functionality, ensuring robust monitoring and control during testing.

6.1.4. Test Workflow

Following successful project synthesis, implementation, and preparation of the Python scripts required for IPbus register access, the utilities of the EMP framework were used for firmware upload, data injection, and capture. The test workflow involved the following sequence of steps:

(i) *FPGA Programming:* The synthesized bitstream file was first loaded onto the FPGA located on the Serenity board:

```
serenitybutler program x0 /home/cmx/Berkay/autorecovery_0/
   autorecovery_0.bit
```

(ii) *Routing Initialization:* Python scripts were executed to initialize the routing map and configure IPbus control registers:

```
python3 autorec_reg.py
```

(iii) *lpGBT Input Injection:* A predefined lpGBT test case file was injected into the receiver (Rx) buffers to simulate input conditions:

```
empbutler -c my_connections.xml do x0 buffers rx PlayOnce -
   c 0-13 --inject file://case01.txt
```

(iv) *Transmit Buffer Capture:* Tx buffers were set into capture mode for subsequent analysis:

```
empbutler -c my_connections.xml do x0 buffers tx Capture
```

(v) *Data Capture:* Finally, both Rx and Tx buffers were captured for analysis:

```
empbutler -c my_connections.xml do x0 capture --tx 0-13 --
   rx 0-13
```

6.1.5. Test Results

Configuration of threshold values, priorities, and corresponding hexadecimal equivalents of fast commands used in this test are provided in Table 6.1. The routing map utilized for mapping ECON-Ds to FCCs is outlined in Table 6.2.

Table 6.1. Flag information EMP test.

Flag	*Threshold*	*Priority*	*Fast Command*
econd_rr	1	1	D1
timeout	4	2	1D
no_idle	16	3	A9
bx_mismatch	256	4	95
ev_mismatch	0	0	D2
buf_overflow	2	2	B8

Table 6.2. Routing map for EMP test.

ECON-D	*FCC*
1	2
2	3
3	4
4	3
5	5
6	5
7	5
8	1
9	7
10	7
11	2
12	8

The lpGBT input file in Figure 6.2 is used in this test. Figure 6.3 illustrates how ECON-Ds are routed to the FCCs, thresholds of the error flags, and shows the test results obtained from executing the described workflow. Each of the eight rows corresponds to a FCC, clearly indicating associated ECON-D modules along with their configured counter thresholds. The reset signal for the Autorecovery Block is released at Frame 4; hence the analyzed data commences from Frame 5.

The lower-right section of the figure presents the output format, where the first byte represents the fast command itself, followed immediately by a valid bit. The valid bit, when set to logic `high`, signifies the presence of a legitimate fast command.

An important observation can be made regarding Channel 5, connected to ECON-D modules 5, 6, and 7. Notably, ECON-D 5 was intentionally disabled via the control registers, hence no corresponding output is observed. ECON-D 7, with a counter threshold value of 4, initially displays its error flags until ECON-D 6, which has higher priority, concurrently signals an error condition. Consequently, the output prioritizes ECON-D 6's fast command. Additionally, the right-hand side of the figure demonstrates real-time status register values, indicating each ECON-D module's error flag count at the captured instant.

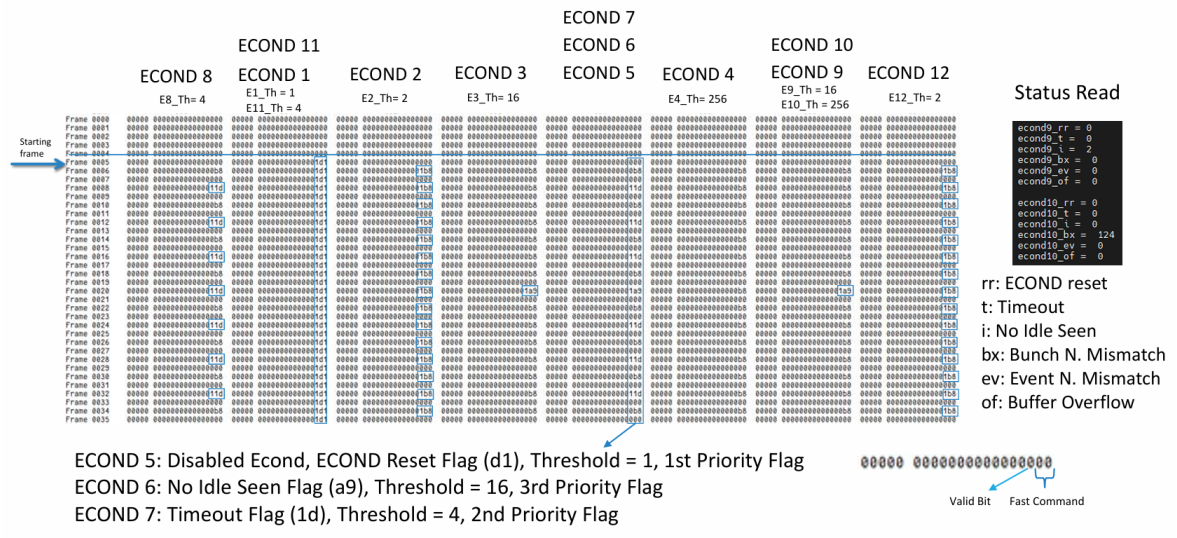


Figure 6.3. FCC test results following Autorecovery Block reset at Frame 4. The diagram displays channel outputs, associated ECON-D modules, and their counter thresholds. Channel 5 behavior illustrates output prioritization among ECON-D 6 and ECON-D 7, with ECON-D 5 disabled. The valid bit and fast command structure are shown alongside real-time error flag status.

6.2. FIFO Upgrade

The Autorecovery Block will be integrated and tested as part of the complete DAQ firmware during beam tests. To improve system reliability and facilitate debugging, it has been decided that fast commands will not be sent directly to the ECON-D modules. Instead, these commands will be buffered in a FIFO memory structure and read periodically. This approach reduces the risk of propagating potential Autorecovery Block malfunctions.

6.2.1. Implementation

The FIFO utilized in this design is implemented using Vivado's *FIFO Generator* IP core. It is configured as a Native Common Clock FIFO, with memory storage implemented in Block RAM (BRAM). The FIFO input data width is set to 8 bits, matching the width of the fast commands, and the depth is set to 1024 entries, resulting in a 10-bit `data_count` signal.

The FIFO exhibits a read latency of two clock cycles from the rising edge of the read clock. However, since the FIFO read operations occur periodically rather than continuously, this latency does not significantly impact overall system performance. Detailed documentation regarding the Vivado FIFO Generator IP can be found in the AMD Xilinx PG057 documentation [29].

With this configuration, the FIFO module provides nine interface ports: `clk`, `din`, `wr_en`, `rd_en`, `dout`, `full`, `empty`, `valid`, and `data_count`. The specifications of the FIFO are illustrated in Figure 6.4.

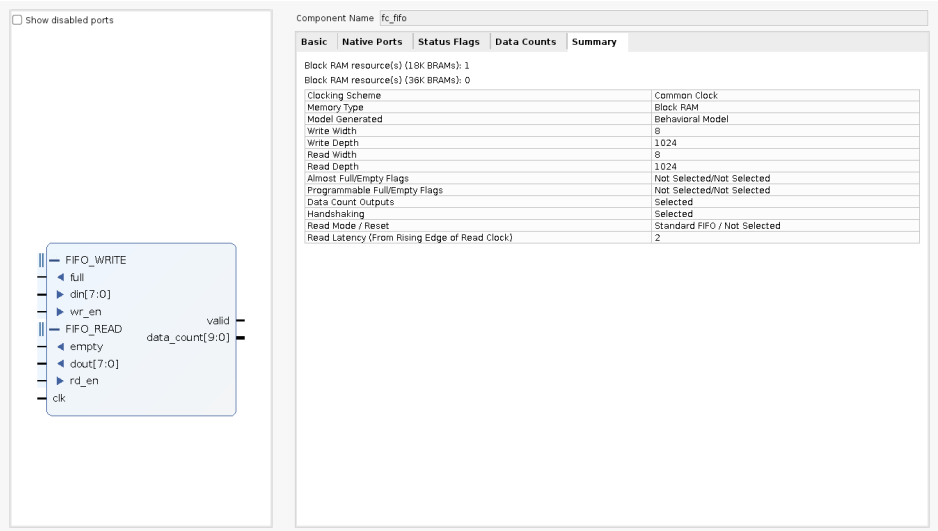


Figure 6.4. Configuration and interface of the FIFO module used in the design.

The fast command array output generated by the Router submodule is connected to the data input of the FIFO, while the simultaneously asserted fast command valid signal is used as the FIFO's write enable signal, as illustrated in Figure 6.5. This configuration ensures that each valid fast command is written into the FIFO precisely when it is generated, preserving synchronization between data and control flow.

```
-- decide valid
ch: for i in 1 to NO_OF_FC_CHs generate begin --flag
  process (is_fast_command_valid_i)
    variable fast_command_valid_array : std_logic_vector( 1 to NO_OF_ECONDs ) := (others => '0');
  begin
    for j in 1 to NO_OF_ECONDs loop --flag
      if (ROUTING_MAP(j)=i) then
        fast_command_valid_array(j) := is_fast_command_valid_i(j);
      else
        fast_command_valid_array(j) := '0' ;
      end if;
    end loop;
    is_fast_command_valid(i) <= or fast_command_valid_array;
    fifo_wren_s(i) <= or fast_command_valid_array;
  end process;
end generate;


-- decide which action to trigger
process (is_fast_command_valid_i,fast_command_array_i) begin
  for i in 0 to NO_OF_FC_CHs-1 loop --channel
    fast_command_array(i+1)    <= (others => '0');  --default
    for j in NO_OF_ECONDs downto 1 loop --priority highest is the lowest number of econd
      if (is_fast_command_valid_i(j)='1' and ROUTING_MAP(j)=i+1) then
        fast_command_array(i+1) <= fast_command_array_i (j);
      end if;
    end loop;
  end loop;
end process;

fast_command_array_o    <= fast_command_array;
is_fast_command_valid_o <= is_fast_command_valid;
fifo_wren <= fifo_wren_s;
```

Figure 6.5. Write enable signal generated in the Router submodule.

The instantiation and signal assignments related to the FIFO within the Autorecovery Block are shown in Figure 6.6.

```
router_i : router port map (
  ROUTING_MAP => ROUTING_MAP,
  fast_command_array_i => fast_command_array,
  is_fast_command_valid_i => is_fast_command_valid,
  fast_command_array_o =>  fast_command_array_out,
  is_fast_command_valid_o => is_fast_command_valid_o,
  fifo_wren =>  fifo_wren_o
);

fast_command_array_o <= fast_command_array_out;


fifo: for i in 1 to NO_OF_FC_CHs generate begin
 fast_command_fifo : fc_fifo
 PORT MAP (
   clk => clk_i,
   din => fast_command_array_out(i),
   wr_en => fifo_wren_o(i),
   rd_en => fifo_ren_i,
   dout => fifo_o(i),
   full => fifo_stats(i)(12),
   empty => fifo_stats(i)(11),
   valid => fifo_stats(i)(10),
   data_count => fifo_stats(i)(9 downto 0)
 );
 end generate;
```

Figure 6.6. Connection between the Router submodule and the FIFO. The fast command array is fed into the FIFO's data input, while the valid signal controls the FIFO's write enable. This ensures that only valid fast commands are written into the FIFO at the moment they are formed.

6.2.1.1. FIFO Read Logic. To provide external control and periodic monitoring of FIFO status, an IPbus control register and two internal logic signals are employed. The read logic follows a straightforward state-transition method, employing a *current–previous* signal tracking scheme. Three signals are central to this mechanism:

- `fifo_rd_en_i`: The input signal provided through the IPbus control register.
- `fifo_ren`: The internal enable signal driving the FIFO read operation.
- `fifo_ren_prev`: An internal signal tracking the previous state of `fifo_rd_en_i`.

Initially, both internal signals (`fifo_ren` and `fifo_ren_prev`) are set to zero. When the external IPbus register signal (`fifo_rd_en_i`) transitions from `'0'` to `'1'`, the internal logic sets `fifo_ren` high for one clock cycle, thereby triggering a single FIFO read operation. Subsequently, the FIFO read logic requires the external signal `fifo_rd_en_i` to return to `'0'` before another read can occur, with `fifo_ren_prev` updated accordingly.

This mechanism ensures that each rising edge on the IPbus register input (`'0'` to `'1'` transition) generates precisely one FIFO read, allowing controlled data retrieval while maintaining system stability. As shown in Figure 6.7, the described logic is implemented using a VHDL process that is sensitive to the system clock.

```
fifo_ren_p : process(clk_i)
begin
   if  rising_edge(clk_i) then
      if fifo_rd_en_i = '1' and fifo_ren_prev = '0' then
         fifo_ren <= '1';
      else
         fifo_ren <= '0';
      end if;
      fifo_ren_prev <= fifo_rd_en_i;
   end if;
end process;
```

Figure 6.7. FIFO read process.

6.2.1.2. Input Sampling Logic. During initial testing, continuous incoming signals at 320 MHz caused the FIFO to fill almost instantaneously, resulting in its `full` status flag remaining persistently asserted. This condition prevented the `data_count` signal from being read effectively for diagnostic purposes. To mitigate this issue, an input sampling logic was implemented to selectively capture input signals, thereby enabling controlled FIFO loading.

This input sampling mechanism comprises two primary control signals, managed through an IPbus control register:

- `capturing_i`: An external control signal provided via the IPbus register.
- `capture`: An internal logic signal used to control data flow into the FIFO.

Initially, the internal `capture` signal is held low. Upon receiving a high-level assertion (`'1'`) from the external `capturing_i` input—while the internal `capture` signal remains low—a variable named `sample_counter` begins incrementing on each rising edge of the system clock. Concurrently, the incoming error flag array (`capture_block_flags_array_i`) is sampled and stored in an internal buffer.

When the `sample_counter` reaches a predefined threshold (800 clock cycles in this implementation), the internal `capture` signal is asserted high. This action halts further sampling and resets the internal buffer signals to zero. As a result, the FIFO is populated with a finite, known quantity of data, facilitating verification that the Autorecovery Block correctly counts error flags according to the specified thresholds.

The described logic is implemented using a VHDL process that is sensitive to the system clock, as illustrated in Figure 6.8.

```
   input_sampler: process(clk_i)
variable sample_counter : integer := 0;
begin
if rising_edge(clk_i) then
 if capturing_i = '1' and capture = '0' then
       sample_counter := sample_counter + 1;
       capture_block_flags_array <= capture_block_flags_array_i;
    if sample_counter = 800 then
       capture <= '1';
       sample_counter := 0;
       capture_block_flags_array <= (others =>(others => '0'));
    end if;
  end if;
    end if;
end process;
```

Figure 6.8. Input sampling logic implemented to control FIFO loading.

6.2.2. Vivado Testbench

The functionality of the Autorecovery Block integrated with the upgraded FIFO has been verified through simulations performed using a Vivado testbench environment. The simulation results confirm that fast commands are correctly buffered within the FIFO and can subsequently be retrieved accurately. Additionally, monitoring of FIFO status signals has been successfully validated, as illustrated in Figure 6.9.

During these tests, critical FIFO control and status indicators, including the full, empty, and valid flags, were observed to behave consistently and correctly in response to the simulated FIFO word count conditions. Furthermore, the testbench simulations validated the correct operation of the read-enable and input-capturing signals, both intended for control via IPbus registers. These signals were demonstrated to function as intended, providing confidence in the robustness of the implemented FIFO logic and its integration within the overall DAQ architecture.

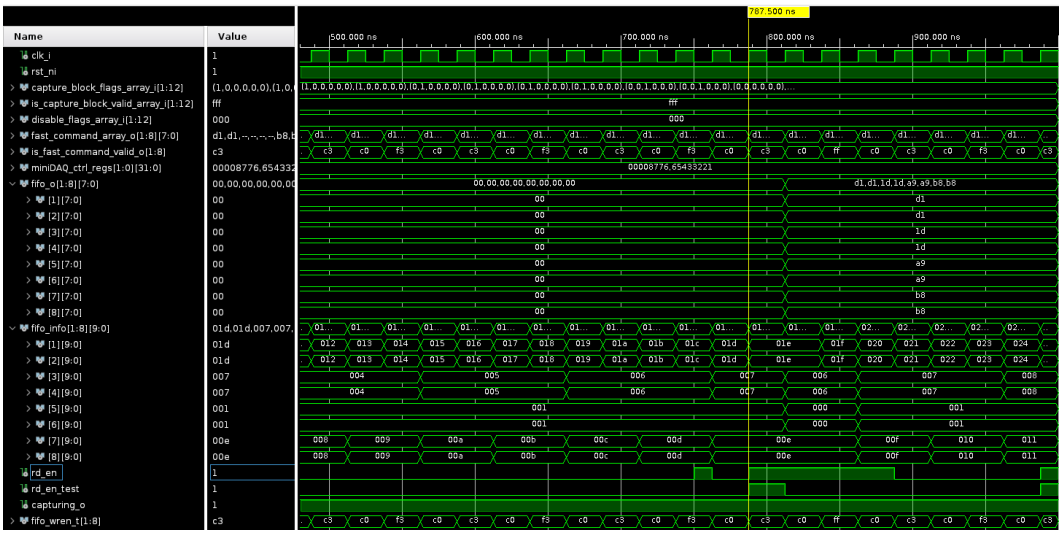


Figure 6.9. Simulation results verifying the functionality of the Autorecovery Block with the integrated FIFO.

6.2.3. EMP Integration

To effectively integrate the FIFO-enhanced Autorecovery Block with the EMP framework and utilize the FIFO functionalities, additional IPbus registers have been instantiated. Specifically, two distinct control registers have been implemented, one dedicated to the FIFO read-enable signal and another for the input-capturing process. Additionally, a total of sixteen status registers have been introduced: eight for monitoring the internal FIFO status (such as data count, full, and empty flags), and eight to hold the output data from the FIFO itself.

With this configuration, each FCC FIFO is allocated two dedicated registers: one to represent its current operational status and another to capture its output data. The detailed assignments and mapping of these updated IPbus registers are provided in Figure 6.10.

```
-- Status Regs

gen_stat_regs_flags: for i in 0 to 11 generate
   miniDAQ_stat_regs(i)(3 downto 0) <= std_logic_vector(to_unsigned(error_flag_counter_o(i+1)(0),4));
   miniDAQ_stat_regs(i)(8 downto 4) <= std_logic_vector(to_unsigned(error_flag_counter_o(i+1)(1),5));
   miniDAQ_stat_regs(i)(14 downto 9) <= std_logic_vector(to_unsigned(error_flag_counter_o(i+1)(2),6));
   miniDAQ_stat_regs(i)(23 downto 15) <= std_logic_vector(to_unsigned(error_flag_counter_o(i+1)(3),9));
   miniDAQ_stat_regs(i)(27 downto 24) <= std_logic_vector(to_unsigned(error_flag_counter_o(i+1)(4),4));
   miniDAQ_stat_regs(i)(31 downto 28) <= std_logic_vector(to_unsigned(error_flag_counter_o(i+1)(5),4));
end generate gen_stat_regs_flags;

gen_stat_regs_fifo: for i in 12 to 19 generate
   miniDAQ_stat_regs(i)(12 downto 0) <= fifo_stats_o(i-11);
end generate gen_stat_regs_fifo;

gen_stat_regs_fifo_o : for i in 20 to 27 generate
   miniDAQ_stat_regs(i)(7 downto 0) <= fifo_out(i-19);
end generate gen_stat_regs_fifo_o;

-- Control Regs

routing_ctrl(1)  <=  miniDAQ_ctrl_regs(0)(3 downto 0);
routing_ctrl(2)  <=  miniDAQ_ctrl_regs(0)(7 downto 4);
routing_ctrl(3)  <=  miniDAQ_ctrl_regs(0)(11 downto 8);
routing_ctrl(4)  <=  miniDAQ_ctrl_regs(0)(15 downto 12);
routing_ctrl(5)  <=  miniDAQ_ctrl_regs(0)(19 downto 16);
routing_ctrl(6)  <=  miniDAQ_ctrl_regs(0)(23 downto 20);
routing_ctrl(7)  <=  miniDAQ_ctrl_regs(0)(27 downto 24);
routing_ctrl(8)  <=  miniDAQ_ctrl_regs(0)(31 downto 28);
routing_ctrl(9)  <=  miniDAQ_ctrl_regs(1)(3 downto 0);
routing_ctrl(10) <=  miniDAQ_ctrl_regs(1)(7 downto 4);
routing_ctrl(11) <=  miniDAQ_ctrl_regs(1)(11 downto 8);
routing_ctrl(12) <=  miniDAQ_ctrl_regs(1)(15 downto 12);

gen_disable_regs: for i in 0 to 11 generate
disable_flags_array(i+1) <= miniDAQ_ctrl_regs(2)(i);
end generate gen_disable_regs;

fifo_read <= miniDAQ_ctrl_regs(3)(0);
capturer <= miniDAQ_ctrl_regs(4)(0);
```

Figure 6.10. IPbus register mapping for the FIFO-enhanced Autorecovery Block.

6.2.3.1. IPbus Register Mapping. Following the implementation of additional IPbus registers, the corresponding address table XML file was revised accordingly. Due to the expanded register space, the largest group of registers now comprises 28 registers in total. Consequently, to maintain proper alignment and adhere to IPbus addressing conventions, the starting address for this register block has been set to decimal 32 (0x20 in hexadecimal).

The updated address table, including detailed definitions of the newly introduced registers along with their respective addresses and bit masks, is presented in Figure 6.11.

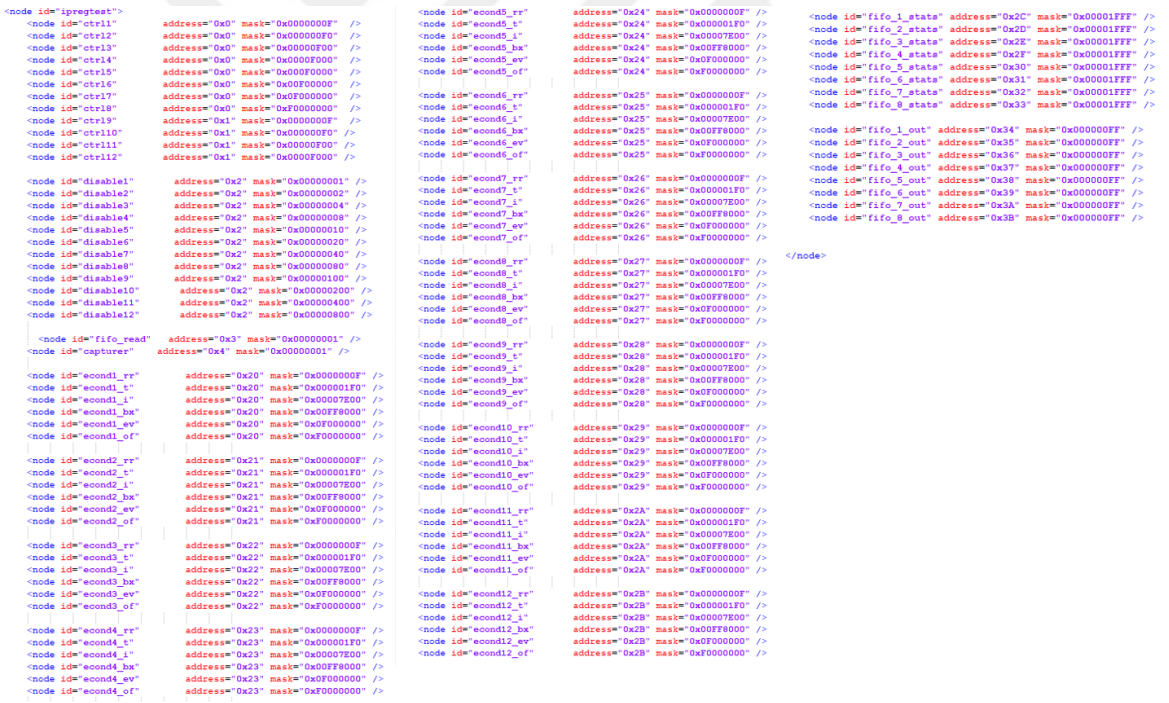


Figure 6.11. Updated IPbus address map used in the Autorecovery Block. The figure consolidates the complete register layout, including control and disable signals, ECON-D interface registers, and the FIFO-related control, status, and data registers.

6.2.4. FIFO Read Scripts

During system operation, after programming the FPGA and injecting the lpGBT input file, the Autorecovery Block initially does not capture any data, consistent with the design of the input sampling logic. To initiate data capture, it is necessary to set the internal capture signal to logic high, thus enabling data flow through the Autorecovery Block. For this purpose, a Python script, illustrated in Appendix B.2, was developed. Executing this script after data injection initiates the data flow, allowing a predetermined number of clock cycles of data to propagate through the Autorecovery Block.

An additional Python script was developed to monitor the FIFO status by periodically reading the relevant status registers, extracting key parameters such as the `full`, `empty`, and `data_count` flags, and retrieving FIFO output data. This monitoring process is executed continuously at regular intervals (every second) during system operation. Each measurement is timestamped and systematically recorded in a CSV file for subsequent analysis. This approach enables detailed tracking of FIFO behavior over extended periods, thereby supporting a thorough evaluation of FIFO functionality and performance. The Python script used for this monitoring process is provided in Appendix B.3.

6.2.5. Test Procedure and Results

The testing workflow for the FIFO-enhanced Autorecovery Block integrated into the EMP system has been revised to incorporate enhanced monitoring and controlled data flow. The updated workflow proceeds as follows:

(i) *FPGA Programming:* The FPGA is initially programmed with the bitstream that includes the FIFO-enhanced Autorecovery Block.

(ii) *Routing Map Initialization:* The routing map is configured by writing to the relevant IPbus control registers.

(iii) *Data Injection:* A predefined case file containing test vectors is injected into the system, initiating data flow through the EMP payload. At this stage, data begins to propagate from the FE modules to the Autorecovery Block.

(iv) *Data Flow Activation:* The Python script written to initiate data flow is executed to assert the `capture` signal. This enables the Autorecovery Block to accept and process data for a fixed duration of 800 clock cycles.

(v) *FIFO Status Logging:* The Python script written to read FIFO is run to continuously monitor and log FIFO status information—specifically the `full`, `empty`, and `data_count` flags—as well as FIFO output data. All measurements are timestamped and stored in a CSV file for offline analysis.

Unlike the previous methodology, explicit capture using EMP buffers is no longer required. Instead, the data flow is strictly limited to 800 cycles, and since EMP capture buffers typically retrieve 1024 frames per read, the Tx buffer outputs remain at zero once the readout is complete.

Given the known configuration of Capture Block error flags and their routing paths to the corresponding FCCs, combined with precise input sampling, it becomes possible to predict the exact number and distribution of fast commands generated in each FCC. These theoretical expectations can then be compared against the actual FIFO output data logged by the monitoring script to validate correct FIFO functionality and the overall performance of the Autorecovery Block.

The routing configuration of ECON-Ds to FCCs is given in Table 6.3. The injected test data used in this workflow is illustrated in Figure 6.12. The error flag thresholds are the same as in Table 6.1.

Table 6.3. Routing map for Fast Command FIFO test.

ECON-D	*FCC*
1	1
2	3
3	4
4	5
5	5
6	5
7	5
8	2
9	7
10	7
11	2
12	8

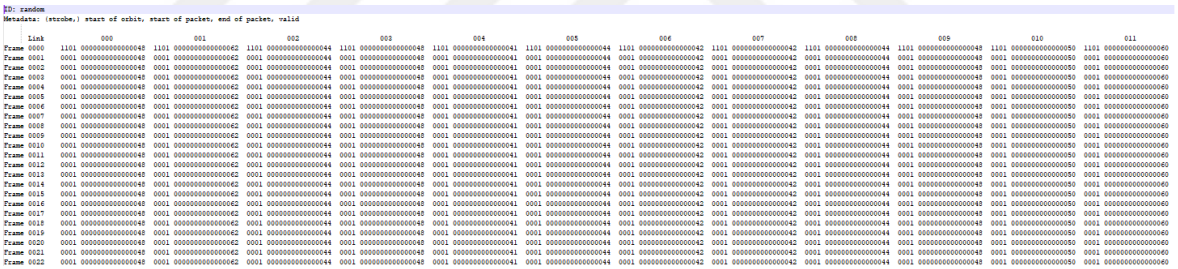


Figure 6.12. Snippet of the lpGBT input file for Fast Command FIFO test.

The first ECON-D (column 0) is routed to FCC 1. The input data assigned to this ECON-D is `0x48` (hexadecimal), which corresponds to the binary pattern `01001000`. This pattern specifically activates the `bx_mismatch` error flag, whose triggering threshold is configured at 256 occurrences. Given that the Autorecovery Block captures and processes input over a fixed duration of 800 clock cycles, it is theoretically expected that this error condition will meet the threshold exactly three times, thereby generating three fast commands.

These fast commands are subsequently written into the FIFO associated with FCC 1. The FIFO status and output are monitored by the Python-based FIFO reader

script described earlier. The recorded data, including timestamps and output words, is logged in a CSV file, which is presented in Figure 6.13.

timestamp	fc_channel	fcc_fifo_out	full	empty	data_count
34:57.7	1	0x95	0	0	2
34:57.7	2	0x1d	0	0	198
34:57.7	3	0xb8	0	0	398
34:57.8	4	0xa9	0	0	48
34:57.8	5	0xd1	0	0	798
34:57.8	6	0x95	0	0	2
34:57.9	7	0xa9	0	0	48
34:57.9	8	0xb8	0	0	398
34:58.9	1	0x95	0	0	1
34:58.9	2	0x1d	0	0	197
34:58.9	3	0xb8	0	0	397
34:58.9	4	0xa9	0	0	47
34:58.9	5	0xd1	0	0	797
34:58.9	6	0x95	0	0	1
34:58.9	7	0xa9	0	0	47
34:58.9	8	0xb8	0	0	397
34:59.9	1	0x95	0	1	0
34:59.9	2	0x1d	0	0	196
34:59.9	3	0xb8	0	0	396
34:59.9	4	0xa9	0	0	46
34:59.9	5	0xd1	0	0	796
34:59.9	6	0x95	0	1	0
34:59.9	7	0xa9	0	0	46
34:59.9	8	0xb8	0	0	396
35:01.0	1	0x95	0	1	0
35:01.0	2	0x1d	0	0	195
35:01.0	3	0xb8	0	0	395
35:01.0	4	0xa9	0	0	45
35:01.0	5	0xd1	0	0	795
35:01.0	6	0x95	0	1	0
35:01.0	7	0xa9	0	0	45
35:01.0	8	0xb8	0	0	395
35:02.0	1	0x95	0	1	0
35:02.0	2	0x1d	0	0	194
35:02.0	3	0xb8	0	0	394
35:02.0	4	0xa9	0	0	44
35:02.0	5	0xd1	0	0	794
35:02.0	6	0x95	0	1	0
35:02.0	7	0xa9	0	0	44
35:02.0	8	0xb8	0	0	394

Figure 6.13. CSV output generated by the FIFO reader Python script for FCCs. The data reflects the logging of fast commands triggered by the error flags.

The logged FIFO output aligns precisely with the theoretical expectations. At the start of data logging, a fast command with the value `0x95` is immediately read from the FIFO output, correctly corresponding to the `bx_mismatch` error flag. In addition, the `data_count` status value confirms that two more fast commands remain stored in the FIFO, thereby validating the anticipated total of three fast commands triggered

for FCC 1.

This behavior is consistently observed across all remaining FCCs, with each channel exhibiting FIFO outputs and status indicators that match the expected results derived from configured error thresholds and input data patterns.

Through these comprehensive tests using real hardware integration, the FIFO-enhanced Autorecovery Block demonstrated correct and robust functionality, successfully passing all evaluated test configurations.

7. STAGE-1 BEAM TEST INTEGRATION

7.1. Beam Test Platform Overview and TCDS2 Emulation Software

Beam tests are periodically conducted at CERN to validate subsystem upgrades under realistic experimental conditions. One such milestone, known as Stage-1, was planned as part of the HGCAL upgrade program for the CMS detector. The associated DAQ firmware is revised accordingly by the HGCAL collaboration in preparation for the September 2024 beam test campaign.

As part of this development process, the enhanced Autorecovery Block described in earlier chapters was integrated into the Stage-1 DAQ firmware. The updated firmware was deployed and tested on the SERENITY platform, operating within the EMP framework. Validation was conducted exclusively in a simulation environment, prior to the actual beam test, using the TCDS2 emulator.

The TCDS2 emulator is used to simulate the timing and control conditions expected during collider operation. It is capable of generating internal (random or regular) or external (e.g., scintillator-based) trigger sequences, and provides essential fast commands such as the Event Counter Reset (ECR), which is used to reset the local event counters. In this work, the emulator was used to inject L1A and ECR signals at specific BX positions, allowing for direct correlation with predefined ECON-D input sequences. These capabilities enabled a complete functional verification of the Autorecovery Block integration in a controlled pre-beam-test environment.

7.2. Integrating the Enhanced Autorecovery Block

7.2.1. Firmware Merge with IPbus

The enhanced Autorecovery Block was successfully integrated into the Stage-1 firmware used in the beam test preparation environment. This firmware architecture employs a modular design, where a `payload_per_lpgbt_pair` entity is instantiated for each `lpGBT` pair. In the current configuration, two such entities are included, each containing a `capture_block` responsible for ECON-D data processing.

To facilitate the integration of the enhanced Autorecovery Block, additional signal pathways for the newly introduced fast command FIFOs were required. These signals, which are managed through IPbus control and status registers, were propagated through two intermediate modules up to the `payload_per_lpgbt_pair` entity. At this level, an IPbus register fabric block is instantiated to enable dynamic configuration of the Capture Block during runtime. The enhanced Autorecovery Block's FIFO control and status signals were also routed to this register fabric for unified management and simplified addressing. This integration allows software-driven initialization and readout of FIFO-related parameters during testing.

Figure 7.1 illustrates the instantiation of the IPbus register block. The IPbus status and control register assignments for both the Capture Block and the enhanced Autorecovery Block integration are shown in Figures 7.2 and 7.3, respectively.

```
--================================--
ipbus_cb_ctrl_stat : entity work.ipbus_syncreg_v
  --================================--
  generic map (
      SWAP_ORDER =>  true,
    n_ctrl => 20,
    n_stat => 34
  )
  port map (
    clk     => ipb_clk_i,
    rst     => ipb_rst_i,
    ipb_in  => ipb_w_array(N_SLV_CAPTURE_BLOCK_CTRL_STAT),
    ipb_out => ipb_r_array(N_SLV_CAPTURE_BLOCK_CTRL_STAT),
    slv_clk => clk_daq_i,
    d       => ipb_regs_stat_cb,
    q       => ipb_regs_ctrl_cb,
    qmask   => (   0 => X"00000001",
                   1 => X"F00007FF",
                   2 => X"0FFFFFFF",
                   3 => X"0FFFFFFF",
                   4 => X"0FFFFFFF",
                   5 => X"0FFFFFFF",
                   6 => X"0FFFFFFF",
                   7 => X"0FFFFFFF",
                   8 => X"0FFFFFFF",
                   9 => X"0FFFFFFF",
                  10 => X"0FFFFFFF",
                  11 => X"0FFFFFFF",
                  12 => X"0FFFFFFF",
                  13 => X"0FFFFFFF",
                  14 => X"0FFFFFFF",
                  15 => X"0FFFFFFF",
                  16 => X"0FFFFFFF",
                  17 => X"0FFFFFFF",
                  18 => X"0FFFFFFF",
                  19 => X"0FFFFFFF"
                  ),
    stb     => open,
    rstb    => open
  );
```

Figure 7.1. IPbus register fabric.

```
ipb_regs_stat_cb(0) <= per_cb_stat_o.capture_block_sr;                    -- capture block status (idle and sticky)
ipb_regs_stat_cb(1) <= per_cb_stat_o.hdr_crc_error_counter(31 downto 0);  -- hdr crc econd 3 downto 0, 8 bit per econd
ipb_regs_stat_cb(2) <= per_cb_stat_o.hdr_crc_error_counter(63 downto 32); -- hdr crc econd 7 downto 4, 8 bit per econd
ipb_regs_stat_cb(3) <= per_cb_stat_o.hdr_crc_error_counter(95 downto 64); -- hdr crc econd 11 downto 8, 8 bit per econd

per_cb_config_i.reset                       <= ipb_regs_ctrl_cb(0)(0);
per_cb_config_i.econ_header_i               <= ipb_regs_ctrl_cb(1)( 8 downto 0);
disable_backpressure                        <= ipb_regs_ctrl_cb(1)(9);
is_BX_mismatch_passthrough_i                <= ipb_regs_ctrl_cb(1)(10);
per_cb_config_i.ECOND_CONFIG.ElinkConfig2.EconNum <= ipb_regs_ctrl_cb(1)( 31 downto 28); -- no of econds

econ_memory : for econ_num in 0 to 11 generate
  per_cb_config_i.ECOND_CONFIG.ElinkConfig3(econ_num).StartAddr <= ipb_regs_ctrl_cb(econ_num + 2)(13 downto 0);  -- start
  per_cb_config_i.ECOND_CONFIG.ElinkConfig3(econ_num).EndAddr   <= ipb_regs_ctrl_cb(econ_num + 2)(27 downto 14); -- end
end generate econ_memory;

elink_to_econ0_to6 : for elink_num in 0 to 6 generate
  per_cb_config_i.ECOND_CONFIG.ElinkConfig(elink_num).EconId <= ipb_regs_ctrl_cb(14)(elink_num * 4 + 3 downto elink_num * 4); -- start
end generate elink_to_econ0_to6;

elink_to_econ7_to13 : for elink_num in 7  to 13 generate
  per_cb_config_i.ECOND_CONFIG.ElinkConfig(elink_num).EconId <= ipb_regs_ctrl_cb(15)((elink_num - 7) * 4 + 3 downto (elink_num - 7) * 4); -- start
end generate elink_to_econ7_to13;
```

Figure 7.2. Capture Block IPbus configuration registers.

```
-- Autorecovery IPBUS Signals
-- Control
  ar_routing_map(23 downto 0)  <= ipb_regs_ctrl_cb(16)(23 downto 0);
  ar_routing_map(47 downto 24) <= ipb_regs_ctrl_cb(17)(23 downto 0);
  ar_disable_econds            <= ipb_regs_ctrl_cb(18)(11 downto 0); -- Econ-D disable FC trigger
  ar_fifo_ren                  <= ipb_regs_ctrl_cb(19)(0)        ; -- FIFO read enable (0 --> 1 to read)

-- Status
   -- Error counters
  ipb_regs_stat_cb(4)  <= "00" &  ar_counters(0)(29 downto 0 );
  ipb_regs_stat_cb(5)  <= "00" &  ar_counters(0)(59 downto 30);
  ipb_regs_stat_cb(6)  <= "00" &  ar_counters(1)(29 downto 0 );
  ipb_regs_stat_cb(7)  <= "00" &  ar_counters(1)(59 downto 30);
  ipb_regs_stat_cb(8)  <= "00" &  ar_counters(2)(29 downto 0 );
  ipb_regs_stat_cb(9)  <= "00" &  ar_counters(2)(59 downto 30);
  ipb_regs_stat_cb(10)  <= "00" &  ar_counters(3)(29 downto 0 );
  ipb_regs_stat_cb(11)  <= "00" &  ar_counters(3)(59 downto 30);
  ipb_regs_stat_cb(12)  <= "00" &  ar_counters(4)(29 downto 0 );
  ipb_regs_stat_cb(13)  <= "00" &  ar_counters(4)(59 downto 30);
  ipb_regs_stat_cb(14) <= "00" &  ar_counters(5)(29 downto 0 );
  ipb_regs_stat_cb(15) <= "00" &  ar_counters(5)(59 downto 30);
  ipb_regs_stat_cb(16) <= "00" &  ar_counters(6)(29 downto 0 );
  ipb_regs_stat_cb(17) <= "00" &  ar_counters(6)(59 downto 30);
  ipb_regs_stat_cb(18) <= "00" &  ar_counters(7)(29 downto 0 );
  ipb_regs_stat_cb(19) <= "00" &  ar_counters(7)(59 downto 30);
  ipb_regs_stat_cb(20) <= "00" &  ar_counters(8)(29 downto 0 );
  ipb_regs_stat_cb(21) <= "00" &  ar_counters(8)(59 downto 30);
  ipb_regs_stat_cb(22) <= "00" &  ar_counters(9)(29 downto 0 );
  ipb_regs_stat_cb(23) <= "00" &  ar_counters(9)(59 downto 30);
  ipb_regs_stat_cb(24) <= "00" &  ar_counters(10)(29 downto 0 );
  ipb_regs_stat_cb(25) <= "00" &  ar_counters(10)(59 downto 30);
  ipb_regs_stat_cb(26) <= "00" &  ar_counters(11)(29 downto 0 );
  ipb_regs_stat_cb(27) <= "00" &  ar_counters(11)(59 downto 30);

  -- FIFO FC output
  ipb_regs_stat_cb(28) <= ar_fifo_fc(31 downto 0)   ;
  ipb_regs_stat_cb(29) <= ar_fifo_fc(63 downto 32)  ;

  -- FIFO status
  ipb_regs_stat_cb(30)      <= "000000" & ar_fifo_info(25 downto 0)  ;
  ipb_regs_stat_cb(31)      <= "000000" & ar_fifo_info(51 downto 26) ;
  ipb_regs_stat_cb(32)      <= "000000" & ar_fifo_info(77 downto 52) ;
  ipb_regs_stat_cb(33)      <= "000000" & ar_fifo_info(103 downto 78);
```

Figure 7.3. Autorecovery Block IPbus control and status registers.

7.2.2. New Address Tables

To accommodate the enhanced Autorecovery Block functionality, the IPbus XML address tables were updated in alignment with the modified firmware hierarchy. The addressing path follows the module structure from the top-level `top_emp` down through `emp_payload`, `hgc_daq`, and `payload_per_lpgbt_pair`, terminating at the `capture_block_ctrl_stat` node. The updated register definitions associated with the Autorecovery Block were included within the `capture_block_ctrl_stat.xml` file.

In this update, new register fields were allocated to accommodate the FIFO control and status signals from the enhanced Autorecovery Block. These include fields for enabling FIFO read, reading output values, and observing status signals such as full, empty, and data count same as in Chapter 6. The base addresses and address widths were revised accordingly to integrate these fields without conflicting with existing al-

locations.

7.3. Packet Structures

The packet structures employed follow the definitions previously detailed in Chapter 3. The format of the ECON-D packet is illustrated in Figure 3.5. The packet structure includes an event packet header, a payload section, and a mandatory `IDLE` word. The lpGBT input files are constructed in full compliance with this format specification.

Similarly, the format of the Capture Block event packet is shown in Figure 3.6. Each Capture Block packet begins with a 64-bit header that encapsulates the L1A information (22 bits) along with status indicators for each ECON-D (3 bits per ECON-D). This packet represents an L1A event for which all participating ECON-Ds are required to share a consistent L1A value.

Both the Capture Block and the ECON-Ds maintain independent L1A counters, and these values are compared as part of the data validation process. The Capture Block performs checks on the CRC and header fields of each received ECON-D packet, verifying that BX, event numbers, and orbit numbers from the ECON-D align with the corresponding local L1A values. This validation determines the status assigned to each ECON-D within the Capture Block. The set of possible status values is given in Table 3.1.

7.4. LpGBT Input Generation and Firmware Tests with TCDS2 Emulator

The lpGBT input text files for the ECON-D data are constructed to comply with the TCDS2 fast commands, ensuring that they generate appropriate error flags from the Event Builder within the Capture Block, which are subsequently sent to the Autorecovery Block.

DAQ system comprises Capture Blocks which receives data from two lpGBT links. Together, these two lpGBTs provide 14 e-link channels, and these channels may carry data from 1 to 12 ECON-Ds, depending on the configuration specified via IPbus registers. Each IPbus register controlling this mapping contains a 4-bit value, where values from 0x0 to 0xB indicate the ECON-D number assigned to a given e-link. A value of 0xF is used to denote that the corresponding e-link is not connected to any ECON-D. The e-link-to-ECON-D connections utilized in this test are summarized in Table 7.1.

Table 7.1. e-link to ECON-D connections for Capture Blocks 1 and 2.

Capture Block 1		*Capture Block 2*	
e-link	*ECON-D*	*e-link*	*ECON-D*
e-link00	0xf	e-link00	0x0
e-link01	0x0	e-link01	0xf
e-link02	0xf	e-link02	0x1
e-link03	0x1	e-link03	0xf
e-link04	0xf	e-link04	0x2
e-link05	0x2	e-link05	0xf
e-link06	0xf	e-link06	0x3
e-link07	0x3	e-link07	0xf
e-link08	0xf	e-link08	0x4
e-link09	0x4	e-link09	0xf
e-link10	0xf	e-link10	0x5
e-link11	0x5	e-link11	0xf
e-link12	0xf	e-link12	0x6
e-link13	0x6	e-link13	0xf

For this Stage-1 firmware, input channel 50 receives data from e-links 1, 3, 5, 7, 9, 11, and 13, while channel 51 receives data from e-links 0, 2, 4, 6, 8, 10, and 12. Since each e-link is connected to a dedicated ECON-D, these e-link pairs are interpreted as ECON-D 1 through ECON-D 7 within the firmware implementation. Therefore, this

test does not explicitly address variations in e-link connectivity or different capture block configurations.

The input sequence begins with `IDLE` words, represented by the hexadecimal value 0xff000000ccccff08. This is followed by the first 32 bits of the packet header for each e-link, then the second 32 bits of the header, after which the payload arrives, corresponding to the length indicated in the header.

The flag thresholds and priorities configured for this test within the Autorecovery Block are illustrated in Figure 7.4. Each flag threshold is set to 32, enabling simultaneous fast command generation for different ECON-Ds during this test. The priorities for the `econd_rr` and `no_idle` flags are explicitly set to zero, as the Event Builder does not yet produce these flags; consequently, no action is triggered in the Autorecovery Block for these cases.

```
entity Autorecovery is
  generic (
    cb_counter_matrix   : cb_counter_matrix_t      := ( others => (others =>"0000100000")); -- Each Econ-D each flag is triggered at 32(0001000000).
    --! Priority of the flags for each  ECOND (row), for flags(columns) 0 means no action , 1st has the highest priority
    cb_priority_matrix : cb_priority_matrix_t := (  (0, 1, 0, 2, 4, 3),
                                                    (0, 1, 0, 2, 4, 3),
                                                    (0, 1, 0, 2, 4, 3),
                                                    (0, 1, 0, 2, 4, 3),
                                                    (0, 1, 0, 2, 4, 3),
                                                    (0, 1, 0, 2, 4, 3),
                                                    (0, 1, 0, 2, 4, 3),
                                                    (0, 1, 0, 2, 4, 3),
                                                    (0, 1, 0, 2, 4, 3),
                                                    (0, 1, 0, 2, 4, 3),
                                                    (0, 1, 0, 2, 4, 3),
                                                    (0, 1, 0, 2, 4, 3));
```

Figure 7.4. Flag thresholds and priorities of Autorecovery Block used in Stage-1 firmware tests.

The routing map defining the assignment of ECON-Ds to FCCs for both Capture Blocks is provided in Table 7.2. Since only ECON-D 1 to ECON-D 7 are utilized in this test, the mapping of the remaining ECON-Ds is not relevant for the current configuration. Consequently, FCCs 5, 6, and 7 are expected to remain empty for this test.

Table 7.2. Routing map for Stage-1 test.

ECON-D	*FCC*
1	4
2	3
3	3
4	2
5	2
6	1
7	8
8	7
9	7
10	6
11	6
12	5

7.4.1. Good Packets

The first 64-bit lpGBT input header used in the simulation was configured as `0xaa02b08001d04cc6`. According to the data format specification, this header encodes the following parameters:

- Header marker: 154
- Payload length: 10
- BX number: 29
- L1A event number: 1
- Orbit number: 1

The CRC field was computed following the guidelines specified in the ECON-D specification document. The remaining payload data were populated with pseudo-random values ranging sequentially from 0 to 9 since payload processing is beyond the scope of this test. Input file used in the test is illustrated in Figure 7.5.

Link	050	051
Frame 0000	0000 00000000ccccff08	0000 00000000ccccff08
Frame 0001	0000 00000000ccccff08	0000 00000000ccccff08
Frame 0002	0000 00000000ccccff08	0000 00000000ccccff08
Frame 0003	0000 00000000ccccff08	0000 00000000ccccff08
Frame 0004	0000 00000000ccccff08	0000 00000000ccccff08
Frame 0005	0000 00000000ccccff08	0000 00000000ccccff08
Frame 0006	0000 00000000ccccff08	0000 00000000ccccff08
Frame 0007	0101 FF000000ccccff08	0101 FF000000ccccff08
Frame 0008	0001 0F000000ccccff08	0001 0F000000ccccff08
Frame 0009	0001 0F000000ccccff08	0001 0F000000ccccff08
Frame 0010	0001 0F000000ccccff08	0001 0F000000ccccff08
Frame 0011	0001 0F000000ccccff08	0001 0F000000ccccff08
Frame 0012	0001 0F000000ccccff08	0001 0F000000ccccff08
Frame 0013	0001 0F000000ccccff08	0001 0F000000ccccff08
Frame 0014	0000 00000000ccccff08	0000 00000000ccccff08
Frame 0015	0101 FF000000aa02b080	0101 FF000000aa02b088
Frame 0016	0001 0F000000aa02b081	0001 0F000000aa02b089
Frame 0017	0001 0F000000aa02b082	0001 0F000000aa02b08a
Frame 0018	0001 0F000000aa02b083	0001 0F000000aa02b08b
Frame 0019	0001 0F000000aa02b084	0001 0F000000aa02b08c
Frame 0020	0001 0F000000aa02b085	0001 0F000000aa02b08d
Frame 0021	0001 0F000000aa02b086	0001 0F000000aa02b08e
Frame 0022	0000 00000000aa02b086	0000 00000000aa02b08e
Frame 0023	0101 FF00000001d04cc6	0101 FF00000001d04cc6
Frame 0024	0001 0F00000001d04cc6	0001 0F00000001d04cc6
Frame 0025	0001 0F00000001d04cc6	0001 0F00000001d04cc6
Frame 0026	0001 0F00000001d04cc6	0001 0F00000001d04cc6
Frame 0027	0001 0F00000001d04cc6	0001 0F00000001d04cc6
Frame 0028	0001 0F00000001d04cc6	0001 0F00000001d04cc6
Frame 0029	0001 0F00000001d04cc6	0001 0F00000001d04cc6
Frame 0030	0000 0000000001d04cc6	0000 0000000001d04cc6
Frame 0031	0101 FF00000000000000	0101 FF00000000000000
Frame 0032	0001 0F00000000000000	0001 0F00000000000000
Frame 0033	0001 0F00000000000000	0001 0F00000000000000
Frame 0034	0001 0F00000000000000	0001 0F00000000000000
Frame 0035	0001 0F00000000000000	0001 0F00000000000000
Frame 0036	0001 0F00000000000000	0001 0F00000000000000
Frame 0037	0001 0F00000000000000	0001 0F00000000000000
Frame 0038	0000 0000000000000000	0000 0000000000000000
Frame 0039	0101 FF00000000000001	0101 FF00000000000001
Frame 0040	0001 0F00000000000001	0001 0F00000000000001
Frame 0041	0001 0F00000000000001	0001 0F00000000000001
Frame 0042	0001 0F00000000000001	0001 0F00000000000001
Frame 0043	0001 0F00000000000001	0001 0F00000000000001
Frame 0044	0001 0F00000000000001	0001 0F00000000000001
Frame 0045	0001 0F00000000000001	0001 0F00000000000001
Frame 0046	0000 0000000000000001	0000 0000000000000001
Frame 0047	0101 FF00000000000002	0101 FF00000000000002
Frame 0048	0001 0F00000000000002	0001 0F00000000000002

Figure 7.5. Snippet of input lpGBT data file showing the configured 64-bit header and payload data used in the simulation for "Good Packets".

To achieve a status indicating "Good Packet," the TCDS2 emulator's YAML configuration file was prepared to dispatch two specific fast commands for each cycle, as shown in Figure 7.6. Fast commands are transmitted at BX + 4 relative to their specified BX values in the YAML file. Accordingly:

- An Event Counter Reset (ECR) command configured at BX = 10 is sent at BX = 14.
- An L1A trigger configured at BX = 25 is sent at BX = 29.

```
0:
  bx: 10
  commands:
  - ECR
  index: 0
  orbit: 0
1:
  bx: 25
  commands:
  - L1A
  index: 1
  orbit: 0
```

Figure 7.6. TCDS2 YAML configuration used to send fast commands (ECR and L1A) at predefined BX positions.

Given the alignment of the lpGBT input BX number and the configured L1A trigger BX number, the Event Builder correctly recognizes and validates the BX number. Furthermore, the Event-ID is cross-validated. Since the lpGBT input event number is configured as 1, each loop initiates with the first event. Importantly, the YAML configuration includes the transmission of an ECR command preceding the L1A trigger in each loop. Consequently, the local event counter resets at the beginning of every cycle, ensuring that the first event is consistently expected and correctly matched, resulting in a validated "Good Packet" status.

The output generated by the Capture Block is verified as 0xfe03a0bfffe00000. This output signifies that all seven connected ECON-Ds have successfully generated valid packets, and further confirms consistency with the original ECON-D input data parameters (BC=29, EC=1). The resulting Capture Block output data captured in the output buffers is shown in Figure 7.7.

```
Frame 0025    00000 fe03a0bfffe00000  00000 fe03a0bfffe00000
Frame 0026    00000 fe03a0bfffe00000  00000 fe03a0bfffe00000
Frame 0027    00001 fe03a0bfffe00000  00000 fe03a0bfffe00000
Frame 0028    00001 aa02b08001d04cc6  00000 fe03a0bfffe00000
Frame 0029    00001 0000000000000001  00000 fe03a0bfffe00000
Frame 0030    00001 0000000200000003  00000 fe03a0bfffe00000
Frame 0031    00001 0000000400000005  00000 fe03a0bfffe00000
Frame 0032    00001 0000000600000007  00000 fe03a0bfffe00000
Frame 0033    00001 0000000800000009  00000 fe03a0bfffe00000
Frame 0034    00001 aa02b08101d04cc6  00000 fe03a0bfffe00000
Frame 0035    00001 0000000000000001  00000 fe03a0bfffe00000
Frame 0036    00001 0000000200000003  00000 fe03a0bfffe00000
Frame 0037    00001 0000000400000005  00000 fe03a0bfffe00000
Frame 0038    00001 0000000600000007  00000 fe03a0bfffe00000
Frame 0039    00001 0000000800000009  00000 fe03a0bfffe00000
Frame 0040    00001 aa02b08201d04cc6  00000 fe03a0bfffe00000
Frame 0041    00001 0000000000000001  00000 fe03a0bfffe00000
Frame 0042    00001 0000000200000003  00000 fe03a0bfffe00000
Frame 0043    00001 0000000400000005  00000 fe03a0bfffe00000
Frame 0044    00001 0000000600000007  00000 fe03a0bfffe00000
Frame 0045    00001 0000000800000009  00000 fe03a0bfffe00000
Frame 0046    00001 aa02b08301d04cc6  00000 fe03a0bfffe00000
Frame 0047    00001 0000000000000001  00000 fe03a0bfffe00000
Frame 0048    00001 0000000200000003  00000 fe03a0bfffe00000
Frame 0049    00001 0000000400000005  00000 fe03a0bfffe00000
Frame 0050    00001 0000000600000007  00000 fe03a0bfffe00000
Frame 0051    00001 0000000800000009  00000 fe03a0bfffe00000
Frame 0052    00001 aa02b08401d04cc6  00000 fe03a0bfffe00000
Frame 0053    00001 0000000000000001  00000 fe03a0bfffe00000
Frame 0054    00001 0000000200000003  00000 fe03a0bfffe00000
Frame 0055    00001 0000000400000005  00000 fe03a0bfffe00000
Frame 0056    00001 0000000600000007  00000 fe03a0bfffe00000
Frame 0057    00001 0000000800000009  00000 fe03a0bfffe00000
Frame 0058    00001 aa02b08501d04cc6  00000 fe03a0bfffe00000
Frame 0059    00001 0000000000000001  00000 fe03a0bfffe00000
Frame 0060    00001 0000000200000003  00000 fe03a0bfffe00000
Frame 0061    00001 0000000400000005  00000 fe03a0bfffe00000
Frame 0062    00001 0000000600000007  00000 fe03a0bfffe00000
Frame 0063    00001 0000000800000009  00000 fe03a0bfffe00000
Frame 0064    00001 aa02b08601d04cc6  00000 fe03a0bfffe00000
Frame 0065    00001 0000000000000001  00000 fe03a0bfffe00000
Frame 0066    00001 0000000200000003  00000 fe03a0bfffe00000
Frame 0067    00001 0000000400000005  00000 fe03a0bfffe00000
Frame 0068    00001 0000000600000007  00000 fe03a0bfffe00000
Frame 0069    00001 0000000800000009  00000 fe03a0bfffe00000
Frame 0070    00000 0000000800000009  00000 fe03a0bfffe00000
Frame 0071    00000 0000000800000009  00001 fe03a0bfffe00000
Frame 0072    00000 0000000800000009  00001 aa02b08801d04cc6
Frame 0073    00000 0000000800000009  00001 0000000000000001
Frame 0074    00000 0000000800000009  00001 0000000200000003
Frame 0075    00000 0000000800000009  00001 0000000400000005
Frame 0076    00000 0000000800000009  00001 0000000600000007
Frame 0077    00000 0000000800000009  00001 0000000800000009
```

Figure 7.7. Snippet of the Capture Block output showing valid ECON-D data with correct BC and EC fields, and confirming a “Good Packet” status for all connected ECON-Ds.

Finally, through IPbus register access commands, it is possible to obtain the FIFO status information, and by setting the read-enable register to 1, FIFO outputs can be actively read and examined. Analysis of the status registers shows that the FIFO empty flags are set high, and consequently, FIFO outputs contain no data. This is precisely the expected behavior under conditions where all packets have been verified as ”Good Packets,” thereby requiring no Autorecovery Block fast commands. The

FIFO status registers and output data are presented in Figure 7.8. The Python script responsible for reading the Autorecovery Block FIFO status and output data has been updated accordingly with the new IPbus register addresses. The resulting CSV log file generated by the script is illustrated in Figure 7.9.

```
192.168.2.5 beamtest >reg_access -d $conid payload.daq.per_lpgbt_pair.capture_block_ctrl_stat.ctrl_stat.stat.fifo_info_00
payload.daq.per_lpgbt_pair.capture_block_ctrl_stat.ctrl_stat.stat.fifo_info_00 = 16779264 : 0x1000800
172.17.0.1
192.168.2.5 beamtest >reg_access -d $conid payload.daq.per_lpgbt_pair.capture_block_ctrl_stat.ctrl_stat.stat.fifo_info_01
payload.daq.per_lpgbt_pair.capture_block_ctrl_stat.ctrl_stat.stat.fifo_info_01 = 16779264 : 0x1000800
172.17.0.1
192.168.2.5 beamtest >reg_access -d $conid payload.daq.per_lpgbt_pair.capture_block_ctrl_stat.ctrl_stat.stat.fifo_info_02
payload.daq.per_lpgbt_pair.capture_block_ctrl_stat.ctrl_stat.stat.fifo_info_02 = 16779264 : 0x1000800
172.17.0.1
192.168.2.5 beamtest >reg_access -d $conid payload.daq.per_lpgbt_pair.capture_block_ctrl_stat.ctrl_stat.stat.fifo_info_03
payload.daq.per_lpgbt_pair.capture_block_ctrl_stat.ctrl_stat.stat.fifo_info_03 = 16779264 : 0x1000800
172.17.0.1
192.168.2.5 beamtest >
172.17.0.1
192.168.2.5 beamtest >reg_access -d $conid payload.daq.per_lpgbt_pair.capture_block_ctrl_stat.ctrl_stat.ctrl.fifo_ren -w --val 1
172.17.0.1
192.168.2.5 beamtest >reg_access -d $conid payload.daq.per_lpgbt_pair.capture_block_ctrl_stat.ctrl_stat.stat.fifo_data_00
payload.daq.per_lpgbt_pair.capture_block_ctrl_stat.ctrl_stat.stat.fifo_data_00 = 0 : 0x0
172.17.0.1
192.168.2.5 beamtest >reg_access -d $conid payload.daq.per_lpgbt_pair.capture_block_ctrl_stat.ctrl_stat.stat.fifo_data_01
payload.daq.per_lpgbt_pair.capture_block_ctrl_stat.ctrl_stat.stat.fifo_data_01 = 0 : 0x0
```

Figure 7.8. FIFO status and output registers indicating no fast command activity, consistent with error-free ECON-D input.

timestamp	fc_channel	fcc_fifo_out	full	empty	valid	data_count
10:41.9	1	0x0	0	1	0	0
10:41.9	2	0x0	0	1	0	0
10:41.9	3	0x0	0	1	0	0
10:41.9	4	0x0	0	1	0	0
10:41.9	5	0x0	0	1	0	0
10:41.9	6	0x0	0	1	0	0
10:41.9	7	0x0	0	1	0	0
10:41.9	8	0x0	0	1	0	0
10:42.9	1	0x0	0	1	0	0
10:42.9	2	0x0	0	1	0	0
10:42.9	3	0x0	0	1	0	0
10:42.9	4	0x0	0	1	0	0
10:42.9	5	0x0	0	1	0	0
10:43.0	6	0x0	0	1	0	0
10:43.0	7	0x0	0	1	0	0
10:43.0	8	0x0	0	1	0	0
10:44.0	1	0x0	0	1	0	0
10:44.0	2	0x0	0	1	0	0
10:44.0	3	0x0	0	1	0	0
10:44.0	4	0x0	0	1	0	0
10:44.0	5	0x0	0	1	0	0
10:44.0	6	0x0	0	1	0	0
10:44.0	7	0x0	0	1	0	0
10:44.0	8	0x0	0	1	0	0
10:45.0	1	0x0	0	1	0	0
10:45.0	2	0x0	0	1	0	0
10:45.0	3	0x0	0	1	0	0
10:45.0	4	0x0	0	1	0	0
10:45.0	5	0x0	0	1	0	0
10:45.0	6	0x0	0	1	0	0
10:45.0	7	0x0	0	1	0	0
10:45.0	8	0x0	0	1	0	0

Figure 7.9. Snippet of the CSV output generated by the updated FIFO reader script, showing real-time FIFO status and fast command entries.

7.4.2. Event-ID Mismatch

The generated 64-bit `lpGBT` input header used in this test was configured as `0xaa02b08001d08c7b`. This header encodes the following parameters:

- Header marker: 154
- Payload length: 10
- BX number: 29
- L1A event number: 2
- Orbit number: 1

The input file used in this configuration is illustrated in Figure 7.10.

Link	050	051
Frame 0000	0000 00000000ccccff08	0000 00000000ccccff08
Frame 0001	0000 00000000ccccff08	0000 00000000ccccff08
Frame 0002	0000 00000000ccccff08	0000 00000000ccccff08
Frame 0003	0000 00000000ccccff08	0000 00000000ccccff08
Frame 0004	0000 00000000ccccff08	0000 00000000ccccff08
Frame 0005	0000 00000000ccccff08	0000 00000000ccccff08
Frame 0006	0000 00000000ccccff08	0000 00000000ccccff08
Frame 0007	0101 FF000000ccccff08	0101 FF000000ccccff08
Frame 0008	0001 0F000000ccccff08	0001 0F000000ccccff08
Frame 0009	0001 0F000000ccccff08	0001 0F000000ccccff08
Frame 0010	0001 0F000000ccccff08	0001 0F000000ccccff08
Frame 0011	0001 0F000000ccccff08	0001 0F000000ccccff08
Frame 0012	0001 0F000000ccccff08	0001 0F000000ccccff08
Frame 0013	0001 0F000000ccccff08	0001 0F000000ccccff08
Frame 0014	0000 00000000ccccff08	0000 00000000ccccff08
Frame 0015	0101 FF000000aa02b080	0101 FF000000aa02b088
Frame 0016	0001 0F000000aa02b081	0001 0F000000aa02b089
Frame 0017	0001 0F000000aa02b082	0001 0F000000aa02b08a
Frame 0018	0001 0F000000aa02b083	0001 0F000000aa02b08b
Frame 0019	0001 0F000000aa02b084	0001 0F000000aa02b08c
Frame 0020	0001 0F000000aa02b085	0001 0F000000aa02b08d
Frame 0021	0001 0F000000aa02b086	0001 0F000000aa02b08e
Frame 0022	0000 00000000aa02b086	0000 00000000aa02b08e
Frame 0023	0101 FF00000001d08c7b	0101 FF00000001d08c7b
Frame 0024	0001 0F00000001d08c7b	0001 0F00000001d08c7b
Frame 0025	0001 0F00000001d08c7b	0001 0F00000001d08c7b
Frame 0026	0001 0F00000001d08c7b	0001 0F00000001d08c7b
Frame 0027	0001 0F00000001d08c7b	0001 0F00000001d08c7b
Frame 0028	0001 0F00000001d08c7b	0001 0F00000001d08c7b
Frame 0029	0001 0F00000001d08c7b	0001 0F00000001d08c7b
Frame 0030	0000 0000000001d08c7b	0000 0000000001d08c7b
Frame 0031	0101 FF0000000000000	0101 FF0000000000000
Frame 0032	0001 0F0000000000000	0001 0F0000000000000
Frame 0033	0001 0F0000000000000	0001 0F0000000000000
Frame 0034	0001 0F0000000000000	0001 0F0000000000000
Frame 0035	0001 0F0000000000000	0001 0F0000000000000
Frame 0036	0001 0F0000000000000	0001 0F0000000000000
Frame 0037	0001 0F0000000000000	0001 0F0000000000000
Frame 0038	0000 0000000000000000	0000 0000000000000000
Frame 0039	0101 FF0000000000001	0101 FF0000000000001
Frame 0040	0001 0F0000000000001	0001 0F0000000000001
Frame 0041	0001 0F0000000000001	0001 0F0000000000001
Frame 0042	0001 0F0000000000001	0001 0F0000000000001
Frame 0043	0001 0F0000000000001	0001 0F0000000000001
Frame 0044	0001 0F0000000000001	0001 0F0000000000001
Frame 0045	0001 0F0000000000001	0001 0F0000000000001
Frame 0046	0000 0000000000000001	0000 0000000000000001
Frame 0047	0101 FF0000000000002	0101 FF0000000000002
Frame 0048	0001 0F0000000000002	0001 0F0000000000002

Figure 7.10. Snippet of the configured 64-bit `lpGBT` input header corresponding to BX = 29 and Event-ID = 2.

In this test scenario, the ECR command is intentionally removed from the YAML configuration file in order to create a controlled mismatch. The modified YAML file is illustrated in Figure 7.11. As a result, although the BX numbers remain correctly aligned between the `lpGBT` input and the TCDS2 emulator configuration, the event numbers do not match. Consequently, the Event Builder classifies the received packet as a "Good Packet with Event-ID Mismatch". These packets are still deemed valid, as the BX alignment is correct. Therefore, they are forwarded to the readout state, despite the Event-ID mismatch condition.

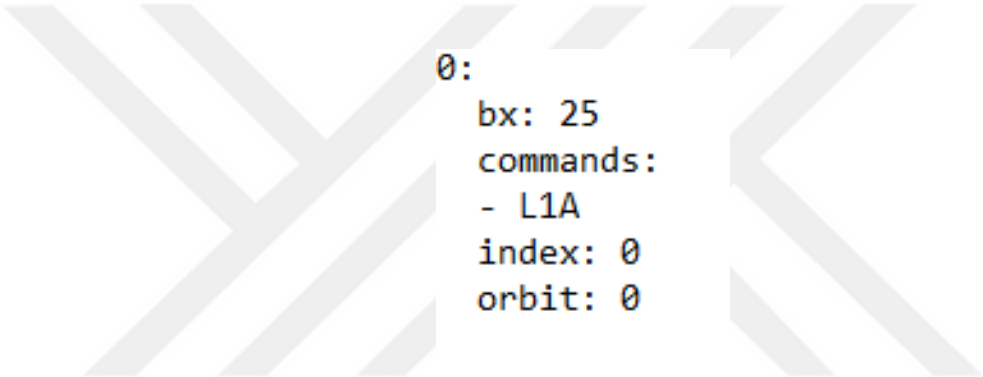


Figure 7.11. Modified TCDS2 emulator YAML file without the ECR command, leading to Event-ID mismatch.

Given that only 7 e-links are connected in this configuratio and they are treated as first 7 ECON-Ds in the firmware, and considering the defined routing map, the expected filled fast command FIFOs are channels 1, 2, 3, 4, and 8. These channels correspond to the fast command destinations for the connected ECON-Ds. Figure 7.12 confirms this behavior: only the fast command FIFOs linked to active ECON-Ds assert their full flags, while the remaining channels remain empty. The output fast command in all active FIFOs is `0xD2`, which correctly corresponds to the configured fast command for an Event-ID mismatch condition.

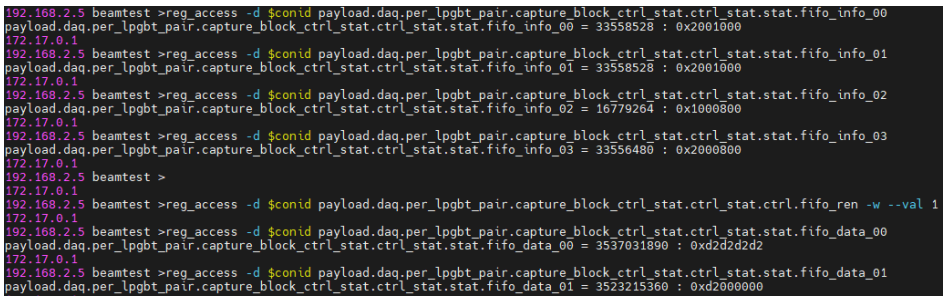


Figure 7.12. FIFO status output showing full flags asserted for FCCs 1, 2, 3, 4, and 8 with correct fast command outputs.

The fast command FIFO contents are also read and logged into a CSV file using the Python-based FIFO reader script. The resulting CSV output is shown in Figure 7.13, confirming the correctness of fast command `0xD2` for each of the active channels.

timestamp	fc_channel	fcc_fifo_out	full	empty	valid	data_count
49:48.3	1	0xd2	1	0	0	0
49:48.3	2	0xd2	1	0	0	0
49:48.3	3	0xd2	1	0	0	0
49:48.3	4	0xd2	1	0	0	0
49:48.3	5	0x0	0	1	0	0
49:48.3	6	0x0	0	1	0	0
49:48.3	7	0x0	0	1	0	0
49:48.3	8	0xd2	1	0	0	0
49:49.3	1	0xd2	1	0	0	0
49:49.3	2	0xd2	1	0	0	0
49:49.3	3	0xd2	1	0	0	0
49:49.3	4	0xd2	1	0	0	0
49:49.3	5	0x0	0	1	0	0
49:49.3	6	0x0	0	1	0	0
49:49.3	7	0x0	0	1	0	0
49:49.3	8	0xd2	1	0	0	0
49:50.3	1	0xd2	1	0	0	0
49:50.3	2	0xd2	1	0	0	0
49:50.3	3	0xd2	1	0	0	0
49:50.3	4	0xd2	1	0	0	0
49:50.3	5	0x0	0	1	0	0
49:50.3	6	0x0	0	1	0	0
49:50.3	7	0x0	0	1	0	0
49:50.3	8	0xd2	1	0	0	0
49:51.4	1	0xd2	1	0	0	0
49:51.4	2	0xd2	1	0	0	0
49:51.4	3	0xd2	1	0	0	0
49:51.4	4	0xd2	1	0	0	0
49:51.4	5	0x0	0	1	0	0
49:51.4	6	0x0	0	1	0	0
49:51.4	7	0x0	0	1	0	0
49:51.4	8	0xd2	1	0	0	0

Figure 7.13. Snippet of the CSV output of FIFO readout script confirming expected FCCs and fast command value for Event-ID mismatch.

7.4.3. Bunch Crossing Mismatch

ECON-D packets exhibiting a BX mismatch condition go to `s5_StartReadout` state only if the BX mismatch passthrough feature is explicitly enabled via software, as detailed in Table 3.1. In the current test configuration, the BX mismatch passthrough is disabled.

If no packets go to `s5_StartReadout` state, the Event Builder, upon reaching the `s4_Header` state, transmits the backend header to the Event Buffer and subsequently returns to the `s0_WaitForL1A` state. At this point, the `error_flags_valid` signal transitions back to a logic low state. To enable the transmission of valid BX mismatch error flags from the Event Builder to Autorecovery Block, the system must transition into the `s5_StartReadout` state.

If all incoming ECON-D packets contain incorrect BX numbers, the resulting backend header indicates a "BX mismatch between Capture Block and ECON-D" condition; however, continuous error flag propagation to the Autorecovery Block does not occur under these circumstances. Consequently, the test scenario requires a combination of correct BX numbered and BX mismatched packets to ensure progression into the readout state, thereby facilitating consistent error flag transmission.

To fulfill this requirement, an input file combining both Event-ID mismatch packets (considered good due to correct BX) and BX mismatch packets was generated. The lpGBT input file is presented in Figure 7.14. Specifically:

- ECON-Ds 1 to 4 utilized the header `0xaa02b08001d08c7b`, matching the previous Event-ID mismatch configuration (BX=29).
- ECON-Ds 5 to 7 employed the header `0xaa02b08001f08cbc`, corresponding to a BX number of 31 and an L1A event number of 2, with the remaining packet configuration identical, including a correctly calculated CRC.

```
      Link              050                   051
Frame 0000    0000 00000000ccccff08  0000 00000000ccccff08
Frame 0001    0000 00000000ccccff08  0000 00000000ccccff08
Frame 0002    0000 00000000ccccff08  0000 00000000ccccff08
Frame 0003    0000 00000000ccccff08  0000 00000000ccccff08
Frame 0004    0000 00000000ccccff08  0000 00000000ccccff08
Frame 0005    0000 00000000ccccff08  0000 00000000ccccff08
Frame 0006    0000 00000000ccccff08  0000 00000000ccccff08
Frame 0007    0101 FF000000ccccff08  0101 FF000000ccccff08
Frame 0008    0001 0F000000ccccff08  0001 0F000000ccccff08
Frame 0009    0001 0F000000ccccff08  0001 0F000000ccccff08
Frame 0010    0001 0F000000ccccff08  0001 0F000000ccccff08
Frame 0011    0001 0F000000ccccff08  0001 0F000000ccccff08
Frame 0012    0001 0F000000ccccff08  0001 0F000000ccccff08
Frame 0013    0001 0F000000ccccff08  0001 0F000000ccccff08
Frame 0014    0000 00000000ccccff08  0000 00000000ccccff08
Frame 0015    0101 FF000000aa02b080  0101 FF000000aa02b088
Frame 0016    0001 0F000000aa02b081  0001 0F000000aa02b089
Frame 0017    0001 0F000000aa02b082  0001 0F000000aa02b08a
Frame 0018    0001 0F000000aa02b083  0001 0F000000aa02b08b
Frame 0019    0001 0F000000aa02b084  0001 0F000000aa02b08c
Frame 0020    0001 0F000000aa02b085  0001 0F000000aa02b08d
Frame 0021    0001 0F000000aa02b086  0001 0F000000aa02b08e
Frame 0022    0000 00000000aa02b086  0000 00000000aa02b08e
Frame 0023    0101 FF00000001d08c7b  0101 FF00000001d08c7b
Frame 0024    0001 0F00000001d08c7b  0001 0F00000001d08c7b
Frame 0025    0001 0F00000001d08c7b  0001 0F00000001d08c7b
Frame 0026    0001 0F00000001d08c7b  0001 0F00000001d08c7b
Frame 0027    0001 0F00000001f08cbc  0001 0F00000001f08cbc
Frame 0028    0001 0F00000001f08cbc  0001 0F00000001f08cbc
Frame 0029    0001 0F00000001f08cbc  0001 0F00000001f08cbc
Frame 0030    0000 0000000001f08cbc  0000 0000000001f08cbc
Frame 0031    0101 FF00000000000000  0101 FF00000000000000
Frame 0032    0001 0F00000000000000  0001 0F00000000000000
Frame 0033    0001 0F00000000000000  0001 0F00000000000000
Frame 0034    0001 0F00000000000000  0001 0F00000000000000
Frame 0035    0001 0F00000000000000  0001 0F00000000000000
Frame 0036    0001 0F00000000000000  0001 0F00000000000000
Frame 0037    0001 0F00000000000000  0001 0F00000000000000
Frame 0038    0000 0000000000000000  0000 0000000000000000
Frame 0039    0101 FF00000000000001  0101 FF00000000000001
Frame 0040    0001 0F00000000000001  0001 0F00000000000001
Frame 0041    0001 0F00000000000001  0001 0F00000000000001
Frame 0042    0001 0F00000000000001  0001 0F00000000000001
Frame 0043    0001 0F00000000000001  0001 0F00000000000001
Frame 0044    0001 0F00000000000001  0001 0F00000000000001
Frame 0045    0001 0F00000000000001  0001 0F00000000000001
Frame 0046    0000 0000000000000001  0000 0000000000000001
Frame 0047    0101 FF00000000000002  0101 FF00000000000002
Frame 0048    0001 0F00000000000002  0001 0F00000000000002
Frame 0049    0001 0F00000000000002  0001 0F00000000000002
Frame 0050    0001 0F00000000000002  0001 0F00000000000002
Frame 0051    0001 0F00000000000002  0001 0F00000000000002
Frame 0052    0001 0F00000000000002  0001 0F00000000000002
```

Figure 7.14. Snippet of the lpGBT input file containing both Event-ID mismatch (BX=29, Event number=2) and BX mismatch (BX=31, Event number=2) packet configurations used in the test scenario.

The TCDS2 emulator configuration was set to dispatch L1A and ECR commands at BX number 25, which, accounting for software-induced delays, resulted in command issuance at BX number 29. The corresponding YAML configuration file is shown in Figure 7.15. With these inputs and configurations, the Autorecovery Block was expected to generate corresponding fast commands for both Event-ID mismatch and BX mismatch conditions.

```
0:
  bx: 25
  commands:
  - ECR
  index: 0
  orbit: 0
1:
  bx: 25
  commands:
  - L1A
  index: 1
  orbit: 0
```

Figure 7.15. TCDS2 emulator YAML configuration file set to send ECR and L1A commands at BX=25, resulting in actual command dispatch at BX=29.

Analysis of the Fast Command FIFO outputs reveals the following:

- FCC-1 output: `0x95` (BX mismatch)
- FCC-2, FCC-3, and FCC-4 outputs: `0xd2` (Event-ID mismatch)
- FCC-8 output: `0x95` (BX mismatch)

These outputs are consistent with expectations. Furthermore, ECON-D priority handling was verified. Specifically, ECON-D 4 and ECON-D 5 were both routed to FCC-2. Despite the higher priority of the BX mismatch flag compared to the Event-ID mismatch flag, the ECON-D priority (ECON-D 4 having higher priority over ECON-D 5) prevailed, resulting in the FIFO output containing the fast command associated with ECON-D 4. The corresponding terminal output of the FIFO is shown in Figure 7.16, and the logged CSV file is presented in Figure 7.17.

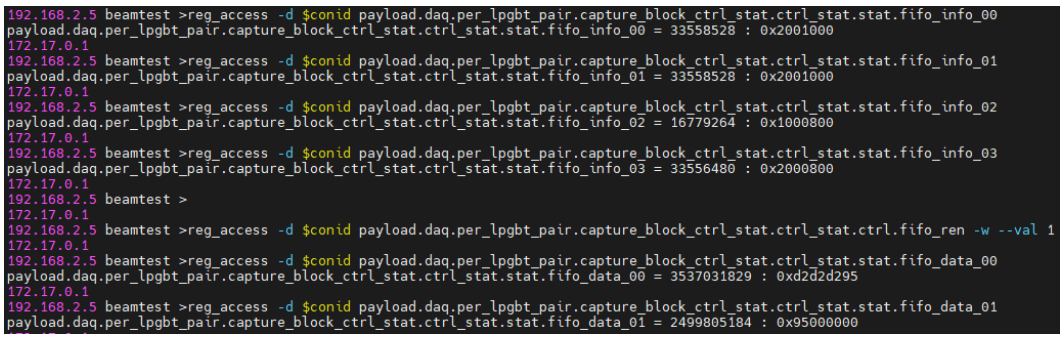


Figure 7.16. Terminal snapshot displaying the Fast Command FIFO outputs, clearly indicating FCC channel assignments and corresponding fast commands triggered due to BX mismatch (`0x95`) and Event-ID mismatch (`0xd2`).

timestamp	fc_channel	fcc_fifo_out	full	empty	valid	data_count
54:13.2	1	0x95	1	0	0	0
54:13.2	2	0xd2	1	0	0	0
54:13.2	3	0xd2	1	0	0	0
54:13.2	4	0xd2	1	0	0	0
54:13.2	5	0x0	0	1	0	0
54:13.2	6	0x0	0	1	0	0
54:13.2	7	0x0	0	1	0	0
54:13.2	8	0x95	1	0	0	0
54:14.2	1	0x95	1	0	0	0
54:14.2	2	0xd2	1	0	0	0
54:14.2	3	0xd2	1	0	0	0
54:14.2	4	0xd2	1	0	0	0
54:14.2	5	0x0	0	1	0	0
54:14.2	6	0x0	0	1	0	0
54:14.2	7	0x0	0	1	0	0
54:14.3	8	0x95	1	0	0	0
54:15.3	1	0x95	1	0	0	0
54:15.3	2	0xd2	1	0	0	0
54:15.3	3	0xd2	1	0	0	0
54:15.3	4	0xd2	1	0	0	0
54:15.3	5	0x0	0	1	0	0
54:15.3	6	0x0	0	1	0	0
54:15.3	7	0x0	0	1	0	0
54:15.3	8	0x95	1	0	0	0

Figure 7.17. CSV log file generated by the Python script, documenting FIFO status, data counts, and output data, confirming the expected fast command entries and FIFO states during the test execution.

7.4.4. Timeout

The timeout condition arises when no valid ECON-D packet header is received within the expected timeframe. This typically occurs due to CRC errors, causing the Packet Former to omit the generation of the *ECON-D packet detected* signal, which is essential for proper Event Builder operation. The input file configuration is depicted in Figure 7.19

	Link	050		051	
Frame	0000	0000	00000000ccccff08	0000	00000000ccccff08
Frame	0001	0000	00000000ccccff08	0000	00000000ccccff08
Frame	0002	0000	00000000ccccff08	0000	00000000ccccff08
Frame	0003	0000	00000000ccccff08	0000	00000000ccccff08
Frame	0004	0000	00000000ccccff08	0000	00000000ccccff08
Frame	0005	0000	00000000ccccff08	0000	00000000ccccff08
Frame	0006	0000	00000000ccccff08	0000	00000000ccccff08
Frame	0007	0101	FF000000ccccff08	0101	FF000000ccccff08
Frame	0008	0001	0F000000ccccff08	0001	0F000000ccccff08
Frame	0009	0001	0F000000ccccff08	0001	0F000000ccccff08
Frame	0010	0001	0F000000ccccff08	0001	0F000000ccccff08
Frame	0011	0001	0F000000ccccff08	0001	0F000000ccccff08
Frame	0012	0001	0F000000ccccff08	0001	0F000000ccccff08
Frame	0013	0001	0F000000ccccff08	0001	0F000000ccccff08
Frame	0014	0000	00000000ccccff08	0000	00000000ccccff08
Frame	0015	0101	FF000000aa02b080	0101	FF000000aa02b088
Frame	0016	0001	0F000000aa02b081	0001	0F000000aa02b089
Frame	0017	0001	0F000000aa02b082	0001	0F000000aa02b08a
Frame	0018	0001	0F000000aa02b083	0001	0F000000aa02b08b
Frame	0019	0001	0F000000aa02b084	0001	0F000000aa02b08c
Frame	0020	0001	0F000000aa02b085	0001	0F000000aa02b08d
Frame	0021	0001	0F000000aa02b086	0001	0F000000aa02b08e
Frame	0022	0000	00000000aa02b086	0000	00000000aa02b08e
Frame	0023	0101	FF00000001d08c2e	0101	FF00000001d08c2e
Frame	0024	0001	0F00000001d08c2e	0001	0F00000001d08c2e
Frame	0025	0001	0F00000001d08c2e	0001	0F00000001d08c2e
Frame	0026	0001	0F00000001d08c2e	0001	0F00000001d08c2e
Frame	0027	0001	0F00000001d08c7b	0001	0F00000001d08c7b
Frame	0028	0001	0F00000001d08c7b	0001	0F00000001d08c7b
Frame	0029	0001	0F00000001d08c7b	0001	0F00000001d08c7b
Frame	0030	0000	0000000001d08c7b	0000	0000000001d08c7b
Frame	0031	0101	FF00000000000000	0101	FF00000000000000
Frame	0032	0001	0F00000000000000	0001	0F00000000000000
Frame	0033	0001	0F00000000000000	0001	0F00000000000000
Frame	0034	0001	0F00000000000000	0001	0F00000000000000
Frame	0035	0001	0F00000000000000	0001	0F00000000000000
Frame	0036	0001	0F00000000000000	0001	0F00000000000000
Frame	0037	0001	0F00000000000000	0001	0F00000000000000
Frame	0038	0000	0000000000000000	0000	0000000000000000
Frame	0039	0101	FF00000000000001	0101	FF00000000000001
Frame	0040	0001	0F00000000000001	0001	0F00000000000001
Frame	0041	0001	0F00000000000001	0001	0F00000000000001
Frame	0042	0001	0F00000000000001	0001	0F00000000000001
Frame	0043	0001	0F00000000000001	0001	0F00000000000001
Frame	0044	0001	0F00000000000001	0001	0F00000000000001
Frame	0045	0001	0F00000000000001	0001	0F00000000000001
Frame	0046	0000	0000000000000001	0000	0000000000000001

Figure 7.18. The lpGBT input file configuration used in the timeout scenario. Headers with deliberately induced CRC errors for ECON-D 1–4 and Event-ID mismatch headers for ECON-D 5–7 are indicated.

In this test scenario, two distinct ECON-D packet headers were configured:

- For ECON-Ds 5, 6, and 7, the previous Event-ID mismatch header (`0xaa02b08001d08c7b`) is used.
- For ECON-Ds 1 through 4, a deliberate CRC error is introduced by altering the original 8-bit CRC segment, resulting in the header `0xaa02b08001d08c2e`.

The YAML file responsible for generating L1A and ECR signals remains identical to the configuration shown previously in Figure 7.15.

Link	050	051
Frame 0000	0000 00000000ccccff08	0000 00000000ccccff08
Frame 0001	0000 00000000ccccff08	0000 00000000ccccff08
Frame 0002	0000 00000000ccccff08	0000 00000000ccccff08
Frame 0003	0000 00000000ccccff08	0000 00000000ccccff08
Frame 0004	0000 00000000ccccff08	0000 00000000ccccff08
Frame 0005	0000 00000000ccccff08	0000 00000000ccccff08
Frame 0006	0000 00000000ccccff08	0000 00000000ccccff08
Frame 0007	0101 FF000000ccccff08	0101 FF000000ccccff08
Frame 0008	0001 0F000000ccccff08	0001 0F000000ccccff08
Frame 0009	0001 0F000000ccccff08	0001 0F000000ccccff08
Frame 0010	0001 0F000000ccccff08	0001 0F000000ccccff08
Frame 0011	0001 0F000000ccccff08	0001 0F000000ccccff08
Frame 0012	0001 0F000000ccccff08	0001 0F000000ccccff08
Frame 0013	0001 0F000000ccccff08	0001 0F000000ccccff08
Frame 0014	0000 00000000ccccff08	0000 00000000ccccff08
Frame 0015	0101 FF000000aa02b080	0101 FF000000aa02b088
Frame 0016	0001 0F000000aa02b081	0001 0F000000aa02b089
Frame 0017	0001 0F000000aa02b082	0001 0F000000aa02b08a
Frame 0018	0001 0F000000aa02b083	0001 0F000000aa02b08b
Frame 0019	0001 0F000000aa02b084	0001 0F000000aa02b08c
Frame 0020	0001 0F000000aa02b085	0001 0F000000aa02b08d
Frame 0021	0001 0F000000aa02b086	0001 0F000000aa02b08e
Frame 0022	0000 00000000aa02b086	0000 00000000aa02b08e
Frame 0023	0101 FF00000001d08c2e	0101 FF00000001d08c2e
Frame 0024	0001 0F00000001d08c2e	0001 0F00000001d08c2e
Frame 0025	0001 0F00000001d08c2e	0001 0F00000001d08c2e
Frame 0026	0001 0F00000001d08c2e	0001 0F00000001d08c2e
Frame 0027	0001 0F00000001d08c7b	0001 0F00000001d08c7b
Frame 0028	0001 0F00000001d08c7b	0001 0F00000001d08c7b
Frame 0029	0001 0F00000001d08c7b	0001 0F00000001d08c7b
Frame 0030	0000 0000000001d08c7b	0000 0000000001d08c7b
Frame 0031	0101 FF00000000000000	0101 FF00000000000000
Frame 0032	0001 0F00000000000000	0001 0F00000000000000
Frame 0033	0001 0F00000000000000	0001 0F00000000000000
Frame 0034	0001 0F00000000000000	0001 0F00000000000000
Frame 0035	0001 0F00000000000000	0001 0F00000000000000
Frame 0036	0001 0F00000000000000	0001 0F00000000000000
Frame 0037	0001 0F00000000000000	0001 0F00000000000000
Frame 0038	0000 0000000000000000	0000 0000000000000000
Frame 0039	0101 FF00000000000001	0101 FF00000000000001
Frame 0040	0001 0F00000000000001	0001 0F00000000000001
Frame 0041	0001 0F00000000000001	0001 0F00000000000001
Frame 0042	0001 0F00000000000001	0001 0F00000000000001
Frame 0043	0001 0F00000000000001	0001 0F00000000000001
Frame 0044	0001 0F00000000000001	0001 0F00000000000001
Frame 0045	0001 0F00000000000001	0001 0F00000000000001
Frame 0046	0000 0000000000000001	0000 0000000000000001

Figure 7.19. The lpGBT input file configuration used in the timeout scenario. Headers with deliberately induced CRC errors for ECON-D 1–4 and Event-ID mismatch headers for ECON-D 5–7 are indicated.

The observed results indicate the expected fast command outputs in the FCCs:

- FCC-1 and FCC-8 output the fast command `0xd2`, corresponding to an Event-ID mismatch.
- FCC-2, FCC-3, and FCC-4 output the fast command `0x1d`, indicating a timeout condition.

Due to simultaneous threshold triggering for ECON-D 4 and ECON-D 5, both connected to FCC-2 but with differing error conditions, the priority resolution mechanism takes effect. Consequently, despite the concurrent timeout and Event-ID mismatch signals, the higher-priority ECON-D (ECON-D 4) determines the fast command, resulting in the timeout command (`0x1d`) appearing at the FCC-2 FIFO output. The FIFO register outputs demonstrating these outcomes are provided in Figure 7.20, and the corresponding Python-generated CSV data log is illustrated in Figure 7.21.

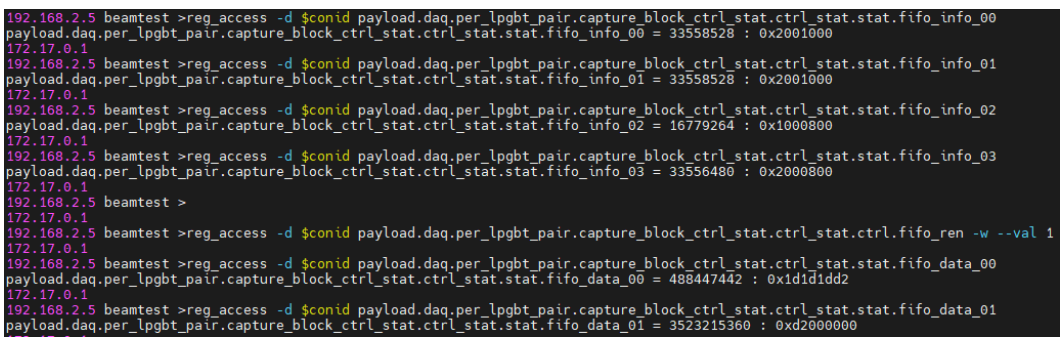


Figure 7.20. Terminal snapshot of FIFO register outputs showing correct FCC allocations for the Event-ID mismatch (`0xd2`) and Timeout (`0x1d`) conditions.

timestamp	fc_channel	fcc_fifo_out	full	empty	valid	data_count
41:16.9	1	0xd2	1	0	0	0
41:16.9	2	0x1d	1	0	0	0
41:16.9	3	0x1d	1	0	0	0
41:16.9	4	0x1d	1	0	0	0
41:16.9	5	0x0	0	1	0	0
41:16.9	6	0x0	0	1	0	0
41:16.9	7	0x0	0	1	0	0
41:16.9	8	0xd2	1	0	0	0
41:17.9	1	0xd2	1	0	0	0
41:17.9	2	0x1d	1	0	0	0
41:17.9	3	0x1d	1	0	0	0
41:17.9	4	0x1d	1	0	0	0
41:18.0	5	0x0	0	1	0	0
41:18.0	6	0x0	0	1	0	0
41:18.0	7	0x0	0	1	0	0
41:18.0	8	0xd2	1	0	0	0
41:19.0	1	0xd2	1	0	0	0
41:19.0	2	0x1d	1	0	0	0
41:19.0	3	0x1d	1	0	0	0
41:19.0	4	0x1d	1	0	0	0
41:19.0	5	0x0	0	1	0	0
41:19.0	6	0x0	0	1	0	0
41:19.0	7	0x0	0	1	0	0
41:19.0	8	0xd2	1	0	0	0

Figure 7.21. CSV file output generated by the FIFO reading Python script, providing timestamped status and data from FIFO registers confirming the expected fast command responses.

8. CONCLUSION

Following the significant upgrade of the Large Hadron Collider (LHC) during Long Shutdown 2, the detector electronics and trigger systems of the CMS experiment are undergoing replacement as part of the Phase-2 upgrade program.

This dissertation has presented the enhancement, verification, EMP framework integration, and hardware validation of the Autorecovery Block, a critical firmware module within the HGCAL BE DAQ system.

Behavioral verification demonstrated that the Autorecovery Block's design achieved complete coverage across all functional aspects. These results confirm that the verification environment successfully stimulated the entire design, validating its nominal functionality as well as its robustness under edge-case conditions.

The integration of a fast command FIFO, along with EMP framework adaptation and IPbus-controlled register interfacing, enabled thorough validation of the Autorecovery Block on Serenity hardware. Tests confirmed correct functionality of the module when deployed in the intended operational environment.

The integration of the Autorecovery Block into the Stage-1 beam test firmware ensures readiness for efficient debugging and performance validation of both the DAQ system and the Autorecovery Block using real hardware in future beam tests. Moreover, the comprehensive tests conducted using the TCDS2 emulator verified that the Autorecovery Block performs reliably and correctly within the Capture Block in preparation for experimental operations.

REFERENCES

1. The CMS Collaboration, "The CMS Experiment at the CERN LHC", *Journal of Instrumentation*, Vol. 3, No. 8, p. S08004, 2008.

2. PICMG, "AdvancedTCA Overview", `https://www.picmg.org/openstandards/advancedtca/`, accessed on May 26, 2025.

3. EMP Collaboration, "EMP Framework Guide", `https://serenity.web.cern.ch/serenity/emp-fwk/`, accessed on May 28, 2025.

4. Aad, G., E. Abat, J. Abdallah, A. A. Abdelalim, A. Abdesselam, O. Abdinov, B. A. Abi, M. Abolins, H. Abramowicz, E. Acerbi, B. S. Acharya, R. Achenbach, M. Ackers, D. L. Adams, F. Adamyan, T. N. Addy, M. Aderholz, C. Adorisio, P. Adragna, M. Aharrouche, S. P. Ahlen, F. Ahles, A. Ahmad, H. Ahmed, G. Aielli, P. F. Åkesson, T. Åkesson, S. M. Alam, J. Albert, S. Albrand, M. Aleksa, I. N. Aleksandrov, M. Aleppo, F. Alessandria, C. Alexa, G. Alexander, T. Alexopoulos, G. Alimonti, M. Aliyev, P. P. Allport, S. E. Allwood-Spiers, A. Aloisio, J. Alonso, R. Alves, M. G. Alviggi, K. Amako, P. Eerola, V. Hedberg, G. Jarlskog, O. Smirnova, V. Zychacek and A. Collaboration, "The ATLAS Experiment at the CERN Large Hadron Collider", *Journal of Instrumentation*, Vol. 3, No. 8, p. S08003, 2008.

5. The ALICE Collaboration, "The ALICE Experiment at the CERN LHC", *Journal of Instrumentation*, Vol. 3, No. 8, p. S08002, 2008.

6. The LHCb Collaboration, "The LHCb Detector at the LHC", *Journal of Instrumentation*, Vol. 3, No. 8, p. S08005, 2008.

7. Schindl, K., "The Injector Chain for the LHC", *9th LEP Performance Workshop*, pp. 47–52, Chamonix, France, 1999.

8. Evans, L. and P. Bryant, "LHC Machine", *Journal of Instrumentation*, Vol. 3, No. 08, p. S08001, 2008.

9. Vidal, X. C. and R. C. Manzano, "RF Cavities", `https://www.lhc-closer.es/taking_a_closer_look_at_lhc/0.rf_cavities/`, accessed on May 26, 2025.

10. Neutelings, I., "CMS Coordinate System", `https://tikz.net/axis3d_cms/`, accessed on May 26, 2025.

11. CMS Collaboration, "Tracking", `https://cms.cern/detector/identifying-tracks/`, accessed on May 26, 2025.

12. CMS Collaboration, *CMS Physics: Technical Design Report Volume 1: Detector Performance and Software*, Technical Design Report. CMS, CERN, Geneva, 2006.

13. Brüning, O., H. Burkhardt and S. Myers, "The Large Hadron Collider", *Progress in Particle and Nuclear Physics*, Vol. 67, No. 3, pp. 705–734, 2012.

14. CMS Collaboration, "The Phase-2 Upgrade of the CMS Endcap Calorimeter", `https://cds.cern.ch/record/2293646/`, accessed on May 26, 2025.

15. Akgün, B., "An Overview of the CMS High Granularity Calorimeter", *Particles*, Vol. 8, No. 1, p. 4, 2025.

16. Soós, C., S. Détraz, L. Olanterä, C. Sigaud, J. Troska, F. Vasey and M. Zeiler, "Versatile Link PLUS Transceiver Development", *JINST*, Vol. 12, No. 03, p. C03068, 2017.

17. Caratelli, A., S. Bonacini, K. Kloukinas, A. Marchioro, P. Moreira, R. De Oliveira and C. Paillard, "The GBT-SCA, A Radiation Tolerant ASIC for Detector Control and Monitoring Applications in HEP Experiments", *JINST*, Vol. 10, No. 03, p. C03034, 2015.

18. Moreira, P., S. Baron, S. Biereigel, J. Carvalho, B. Faes, M. Firlej, T. Fiutowski, J. Fonseca, R. Francisco, D. Gong, N. Guettouche, P. Gui, D. Guo, D. Hernandez, M. Idzik, I. Kremastiotis, T. Kugathasan, S. Kulis, P. Leitao, P. Leroux, E. Mendes, J. Mendez, J. Moron, N. Paulino, D. Porret, J. Prinzie, A. Pulli, Q. Sun, K. Swientek, K. Wyllie, D. Yang, J. Ye, T. Zhang and W. Zhou, "LpGBT Documentation", 2024, `https://cds.cern.ch/record/2809058/`, accessed on May 26, 2025.

19. Rose, A., D. Parker, G. Iles, O. Sahin, P.-A. Bausson, A. Tsirou, G. Fedi, P. Verdini, L. Ardila, M. Balzer, T. Schuh, T. Williams, A. Thea, K. Harder, S. Dugad, R. Shukla and I. Mirza, "Serenity: An ATCA Prototyping Platform for CMS Phase-2", *TWEPP 2018 Topical Workshop on Electronics for Particle Physics*, p. 115, Antwerpen, Belgium, 2019.

20. Hegeman, J. G., "First Measurements with the CMS DAQ and Timing Hub Prototype-1", *TWEPP 2019 Topical Workshop on Electronics for Particle Physics*, p. 6, Santiago De Compostela, Spain, 2019.

21. Mallios, S., P. Dauncey, A. David, P. Vichoudis and on behalf of the CMS collaboration, "Firmware Architecture of the Back End DAQ System for the CMS High Granularity Endcap Calorimeter Detector", *Journal of Instrumentation*, Vol. 17, No. 04, p. C04007, 2022.

22. Rosado, M., A. Şarkışla, S. Mallios, B. Akgün, A. David, N. Roma, P. Tomás and P. Vichoudis, "Back-End DAQ System Prototype Testing and Integration on a Full Detector Test System for the CMS HGCAL Detector", *Journal of Instrumentation*, Vol. 20, No. 03, p. C03028, 2025.

23. Larrea, C. G., K. Harder, D. Newbold, D. Sankey, A. Rose, A. Thea and T. Williams, "IPbus: A Flexible Ethernet-Based Control System for xTCA Hardware", *Journal of Instrumentation*, Vol. 10, No. 02, p. C02019, 2015.

24. SAMTEC, "FireFly Optical Transceivers", `https://www.samtec.com/optics/`

systems/firefly/, accessed on May 28, 2025.

25. IPbus Collaboration, "IPbus Software Repository", https://github.com/ipbus/ipbus-software, accessed on April 4, 2025.

26. AMD, "AMD Vivado Design Suite", https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html, accessed on May 28, 2025.

27. FOSSi Foundation, "Cocotb: Coroutine-Based Co-Simulation Library for Writing VHDL and Verilog Testbenches in Python", https://github.com/cocotb/cocotb, accessed on May 26, 2025.

28. Siemens, "QuestaSim Simulator", https://eda.sw.siemens.com/en-US/ic/questa/, accessed on May 28, 2025.

29. AMD, "FIFO Generator v13.2 Product Guide (PG057)", 2017, https://docs.amd.com/v/u/en-US/pg057-fifo-generator, accessed on May 28, 2025.

APPENDIX A: SUPPLEMENTAL VHDL CODES

A.1. Event Builder State Machine VHDL Code

VHDL source code of the Event Builder state machine, which constitutes the core of the DAQ system.

```
even_builder_fsm: process(clk_i)

variable econd: integer range 0 to max_econ-1 := 0; --
   processed ECOND packet
variable delay: integer range 0 to mb_READ_LATENCY_B + 3:= 0;
variable findelay: integer range 0 to mb_READ_LATENCY_B + 1:=
   0;
variable packet_size: integer range 0 to 511:= 0;
variable payload_lenght: integer range 0 to 511:= 0;
begin


if rising_edge(clk_i) then

  l1a_rd_ack_o     <= '0';
  pkt_info_rd_o    <= (others => '0');
  mb_rden_o        <= (others => '0');
  ev_wren          <= '0';
  timeout_cnt_rst  <= '1';
  ev_data_o <= (others => '0');
  rst_ev_cnt <= '0';

  CASE readout_state is

    WHEN s0_WaitforL1a =>              -- wait for L1A, read
       timestamp and start timeout counter (l1a_fifo must be in
```

```
      FWFT mode)
24
25      mb_addr_o          <= addr_array_i(0);
26      econd_pkt_good     <= (others => '0');
27      be_header          <= (others => '1');
28      error_flags_valid  <= '0';
29      error_flags_array  <= (others => (others => '0'));
30
31      if l1a_empty_i = '0' and ev_full_i = '0' then --l1a
          arrived and event buffer is not full
32
33        l1a_ocid <= l1a_i( 2 downto 0); -- get l1a orbit number
34        l1a_evid <= l1a_i( 8 downto 3); -- get l1a event number
35        l1a_bcid <= l1a_i(20 downto 9); -- get l1a bunch number
36        --
37        l1a_rd_ack_o     <= '1';         -- l1a read acknowledge
            (l1a_fifo must be in FWFT mode)
38        --
39        readout_state <= s1_WaitforPackets;
40
41      else
42
43        readout_state <= s0_WaitforL1a;
44
45      end if;
46
47    -- wait for all ECOND packets to appear in the main buffer
        OR until timeout
48    -- when timeout, readout only the available packets.
49    WHEN s1_WaitforPackets =>
50
51      econd:= 0;
52
```

```
53      if pkt_info_empty_i(0 to econ_num-1) = ALLZEROES(0 to
            econ_num-1) then
54        econd_pkt_det <= (others => '1');
55        readout_state <= s2_GetHdrBX;
56
57      elsif timeout = '1' then
58        econd_pkt_det <= pkt_ready;
59        readout_state     <= s2_GetHdrBX;
60
61      else
62        timeout_cnt_rst <= '0';
63        readout_state     <= s1_WaitforPackets;
64
65      end if;
66
67    WHEN s2_GetHdrBX => -- get econd header BX
68
69      if econd = econ_num then
70        be_header(56 downto 36) <= l1a_bcid & l1a_evid &
              l1a_ocid; -- add l1a timestamp to the event be
              header register
71        readout_state   <= s4_Header;
72
73
74      else
75        if econd_pkt_det(econd) = '1' then
76          mb_addr_o <= addr_array_i(econd);
77          delay:= 0;
78          readout_state <= s3_Timestamp_Check;
79        else
80          be_header(3*econd+2 downto 3*econd) <= "100"; -- no
                packet, timeout
81          error_flags_array(econd + 1).timeout <= '1';
```

```
82              econd:= econd + 1;
83              readout_state <= s2_GetHdrBX;
84            end if;
85
86          end if;
87
88        WHEN s3_Timestamp_Check => -- read econd header bx and
             compare ECOND BX with local BX
89
90          if delay = mb_READ_LATENCY_B then
91
92            if mb_data_i(31 downto 20) = l1a_bcid then --and
                mb_data_i(13 downto 11) = l1a_ocid then --uncomment
                to add orbit counter check
93
94              econd_pkt_good(econd) <= '1';
95              pkt_info_rd_o(econd) <= '1';
96
97              if mb_data_i(19 downto 14) = l1a_evid then
98                be_header(3*econd+2 downto 3*econd) <= "000"; --
                     good packet
99              else
100               be_header(3*econd+2 downto 3*econd) <= "011"; --
                     good packet with event id mismatch
101               error_flags_array(econd + 1).ev_mismatch <= '1';
102             end if;
103
104           else
105
106             if pkt_info_i(2*econd+1) = '1' then --check the
                   buffer_full flag
107               econd_pkt_good(econd) <= '0';
108               be_header(3*econd+2 downto 3*econd) <= "110"; --
```

```
                no packet, buffer overflow
109             error_flags_array(econd + 1).buf_overflow <= '1';
110             pkt_info_rd_o(econd) <= '1';
111           elsif pkt_info_i(2*econd) = '1' then --check the
                 large_packet flag
112             econd_pkt_good(econd) <= '0';
113             be_header(3*econd+2 downto 3*econd) <= "001"; --
                   no packet, large packet
114             error_flags_array(econd + 1).buf_overflow <= '1';
115             pkt_info_rd_o(econd) <= '1';
116           else
117             if (is_BX_mismatch_passthrough_i) then --
                   software controlable passthrough
118               econd_pkt_good(econd) <= '1';
119               pkt_info_rd_o(econd) <= '1';
120             else
121               econd_pkt_good(econd) <= '0';
122               pkt_info_rd_o(econd) <= '0';
123             end if;
124
125           -- econd_pkt_good(econd) <= '0';
126           -- pkt_info_rd_o(econd) <= '0';
127             be_header(3*econd+2 downto 3*econd) <= "101"; --
                   no packet, timestamp mismatch
128             error_flags_array(econd + 1).bx_mismatch <= '1';
129           end if;
130         end if;
131
132         econd:= econd + 1;
133         readout_state <= s2_GetHdrBX;
134
135       else
136
```

```
137           delay := delay + 1;
138           readout_state <= s3_Timestamp_Check;
139
140         end if;
141
142
143       WHEN s4_Header =>  -- send BE Event Header to event fifo
144
145         delay := 0;
146         econd := 0;
147         ev_data_o  <= be_header;
148         ev_wren  <= '1';
149         error_flags_valid <= '1';
150
151         if econd_pkt_good(0 to econ_num-1) = ALLZEROES(0 to
              econ_num-1) then
152           rst_ev_cnt <= '1';
153           readout_state <= s0_WaitforL1a;
154         else
155           readout_state <= s5_startReadout;
156         end if;
157
158
159       WHEN s5_startReadout => -- read econd packets starting
            from the header again
160
161         if econd = econ_num then  --finished reading all econd
              packets
162           rst_ev_cnt <= '1';
163           readout_state <= s0_WaitforL1a;
164         else
165
166           if econd_pkt_good(econd) = '1' then  --packet has a
```

```
            header
167           mb_addr_o <= addr_array_i(econd); --header address
168           mb_rden_o(econd) <= '1';
169           delay:= 0;
170           readout_state <= s6_readHeader;
171         else  -- no packet header ,skip
172           econd:= econd+1;
173           mb_rden_o(econd) <= '0';
174           readout_state <= s5_startReadout;
175         end if;
176
177       end if;
178
179     WHEN s6_readHeader =>
180       -- mb_READ_LATENCY_B because of main buffer delay, +1
            is due to mb_addr_o requires 1 cycles to change, but
            -1 because we are changing it in previous cycle
181       if delay = mb_READ_LATENCY_B then --we have the header
            now, set the size, write header to ev fifo
182         payload_lenght:= to_integer(unsigned(mb_data_i(54
              downto 46))) ;
183         packet_size := (payload_lenght + 1 )  ; -- lenght
              doesnt include header, if the lenght is one we are
              reading it in this cycle
184         packet_size := packet_size /2 ; -- right shift to
              divide by two (no of 32b words - 2 word header)
185         ev_data_o <= mb_data_i(63 downto 0);
186         ev_wren <= '1';
187         delay := 0;
188         mb_addr_o <= addr_array_i(econd); --address is header
              +1
189
190         if payload_lenght = 0 then --truncated packet ,
```

```
            increment econd and read next header
191             econd := econd+1;
192             readout_state <= s5_startReadout;
193           else  --  normal packet read the payload amount
194             readout_state <= s7_ReadPayload;
195             mb_rden_o(econd) <= '1'; -- confirm reading of
                   address header +1
196           end if;
197
198         else
199           delay := delay+1;
200           readout_state <= s6_readHeader;
201         end if;
202
203       WHEN s7_ReadPayload => -- Read Payload
204         -- mb_READ_LATENCY_B because of main buffer delay, +1
              is due to mb_addr_o requires 1 cycles to change, but
              -1 because we are changing it in previous cycle
205         -- set delay and event fifo read, +1 is due to delay of
               read ack
206
207         if delay /= mb_READ_LATENCY_B + 1 then
208           delay := delay+1;
209         else
210           delay := delay;
211           ev_data_o <= mb_data_i(63 downto 0);
212           ev_wren <= '1';
213         end if;
214
215         if packet_size = 0 then --  only payload in the
              pipeline remains
216           readout_state <= s8_FinishPayload;
217           findelay := 0;
```

```
218           econd := econd+1;
219           if econd /= econ_num then -- there are more econds to
                  be read
220             mb_addr_o <= addr_array_i(econd);   -- we want
                   header of the next econd to arrive after the last
                    data of the current econd
221           end if;
222         else  -- there is still payload to be read
223           readout_state <= s7_ReadPayload;
224           if packet_size /= 1 then --remove extra read ack
225             mb_rden_o(econd) <= '1';
226           end if;
227           packet_size := packet_size -1;
228           mb_addr_o <= addr_array_i(econd);
229         end if;
230
231
232       WHEN s8_FinishPayload => -- Wait for all on the fly
             packets to be read-out
233         -- address read in the main buffer is the header of the
              next econd, if econd /=econ_num  when findelay=0
234
235         -- mb_READ_LATENCY_B because of main buffer delay, +1
             is due to mb_addr_o requires 1 cycles to change,
236         -- but -1 because we are changing it in previous cycle,
              one more -1 because we dont want to data of the next
              header we want to last data of the current econd
237         if findelay = mb_READ_LATENCY_B - 1 then -- last data
             of the current econd is arrived
238           ev_data_o <= mb_data_i(63 downto 0);
239           ev_wren <= '1';
240
241           if econd = econ_num then -- all econs are read
```

```
242                 rst_ev_cnt <= '1';
243                 readout_state <= s0_WaitforL1a;
244               else  -- next data is the next header
245                 if econd_pkt_good(econd) = '1' then --packet has
                       a header  that will arrive in the next cycle
246                   mb_rden_o(econd) <= '1';
247                   delay := mb_READ_LATENCY_B;
248                   readout_state <= s6_readHeader;
249                 else -- no packet header ,skip
250                   econd:= econd+1;
251                   mb_rden_o(econd) <= '0';
252                   readout_state <= s5_startReadout;
253                 end if;
254               end if;
255
256             else -- waiting for last payload
257               readout_state <= s8_FinishPayload;
258               findelay := findelay+1;
259               if delay /= mb_READ_LATENCY_B + 1 then -- payload
                     is in the pipeline
260                 delay := delay+1;
261               else  -- payload arrived
262                 delay := delay;
263                 ev_data_o <= mb_data_i(63 downto 0);
264                 ev_wren <= '1';
265               end if;
266             end if;
267
268
269
270   end CASE;
271   --
272
```

```
273     if rstn_i = '0' then readout_state <= s0_WaitforL1a;
274     end if;
275
276 end if;
277 end process;
```

A.2. Autorecovery Declaration VHDL Code

VHDL source code of the Autorecovery Block declaration package.

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4
5  package autorecovery_package is
6
7    constant NO_OF_ECONDs : integer := 12; --Number of ECONDs
8    constant NO_OF_FC_CHs : integer := 8; --Number of Fast
       command channels in the output
9
10   constant FC_ACTION_BIT_SIZE : integer := 8;
11
12   -- Adding a new error flag,
13   --  *increse the number ERROR_FLAG_COUNT by 1
14   --  *add Error Flag to cb_flag_t record
15   --  *add "flags.Error Flag" to ser in serialize_flags
       function
16
17   -- Adding a new fast control command,
18   --  * add fast control command to fc_cmds_t record
19   --  * add fast control command to constant fcc with its
       corresponding hex
20
```

```
21  constant ERROR_FLAG_COUNT : integer := 6;
22  type cb_flag_t is record
23    econd_rr      : std_logic; -- 2-bit reset request coming
        directly from the ECONDs. Different meaning depending on
        the selected error conditions at the ECOND.We don't have
        to count but act on this as soon as we can.
24    -- econd_rr      : std_logic_vector(1 downto 0);
25    timeout       : std_logic; -- There was no ECOND packet
        received for a given L1A, this indicates packet lost due
        to crc errors
26    no_idle       : std_logic; -- There was no IDLE seen within
         a time threshold (indicates an error in the ECOND-lpGBT
        link)
27    bx_mismatch   : std_logic; -- The BC number of the ECOND
        packet did not match the expected BE BCR value. This
        indicates either a missing event or a spurious event.
28    ev_mismatch   : std_logic; -- This is not an error but a
        warning flag and probably it will not generate a recovery
         action. This indicates that the packet received from the
         ECOND has the wrong event number.
29    buf_overflow  : std_logic; -- The ECOND packet arrived but
        was truncated due to BE buffer overflow.
30  end record cb_flag_t;
31
32  --fast control commands
33  type fc_cmds_t is record
34    IDLE          : std_logic_vector(FC_ACTION_BIT_SIZE-1 downto
        0) ; -- Buffer reset (HGROC + ECOND)
35    EBR           : std_logic_vector(FC_ACTION_BIT_SIZE-1 downto
        0) ; -- Buffer reset (HGROC + ECOND)
36    BCR           : std_logic_vector(FC_ACTION_BIT_SIZE-1 downto
        0) ; -- Bunch counter reset (HGROC + ECOND)
37    ECR           : std_logic_vector(FC_ACTION_BIT_SIZE-1 downto
```

```
        0) ; -- Event Counter reset (HGROC + ECOND)
38    BCR_OCR      : std_logic_vector(FC_ACTION_BIT_SIZE-1 downto
        0) ; -- Bunch and Orbit counter reset
39    chipSync     : std_logic_vector(FC_ACTION_BIT_SIZE-1 downto
        0) ; -- Hard reset (HGROC + ECOND)
40    linkRstEconD : std_logic_vector(FC_ACTION_BIT_SIZE-1 downto
        0) ; -- ECOND-BE link reset (ECOND)
41    linkRstRocD  : std_logic_vector(FC_ACTION_BIT_SIZE-1 downto
        0) ; -- ECOND-BE link reset (ROC)
42  end record fc_cmds_t;
43
44  constant fcc          : fc_cmds_t := (IDLE         => x"36",
45                                        EBR          => x"D1",
46                                        BCR          => x"1D",
47                                        ECR          => x"A9",
48                                        BCR_OCR      => x"95",
49                                        chipSync     => x"D2",
50                                        linkRstEconD => x"B8",
51                                        linkRstRocD  => x"9A");
52
53  -- Arbiter
54  type cb_counter_arr_t is array (0 to ERROR_FLAG_COUNT-1) of
      integer;
55  type cb_counter_matrix_t is array (1 to NO_OF_ECONDs) of
      cb_counter_arr_t;
56
57  type cb_priority_arr_t is array (0 to ERROR_FLAG_COUNT-1) of
       integer range 0 to ERROR_FLAG_COUNT;
58  type cb_priority_matrix_t is array (1 to NO_OF_ECONDs) of
      cb_priority_arr_t ;
59
60  type fcc_arr_t is array (0 to ERROR_FLAG_COUNT-1) of
      std_logic_vector(FC_ACTION_BIT_SIZE-1 downto 0);
```

```
61    type fcc_matrix_t is array (1 to NO_OF_ECONDs) of  fcc_arr_t;
62
63    type cb_flag_array_t  is array (1 to NO_OF_ECONDs) of
         cb_flag_t;
64    function serialize_flags (flags :  cb_flag_t) return
         std_logic_vector ;
65
66    -- Router
67    type fast_command_array_t is array(natural range <>) of
         std_logic_vector(FC_ACTION_BIT_SIZE-1 downto 0);
68    type routing_map_t is array(natural range <>) of integer
         range 0 to NO_OF_FC_CHs;
69    type fast_command_array_3d_t is array(natural range <>) of
         fast_command_array_t;
70    type routing_fc is array(natural range <>) of
         std_logic_vector(3 downto 0);
71
72    -- FIFO
73    type fifo_info is array(natural range <>) of std_logic_vector
         (12 downto 0);
74
75  end autorecovery_package;
76
77  package body autorecovery_package is
78    function serialize_flags (flags : cb_flag_t) return
         std_logic_vector is
79      variable ser : std_logic_vector(ERROR_FLAG_COUNT - 1 downto
           0);
80    begin
81      ser := flags.econd_rr &  flags.timeout &  flags.no_idle &
           flags.bx_mismatch & flags.ev_mismatch &  flags.
           buf_overflow     ;
82      return ser;
```

```
83    end;
84 end package body;
```

APPENDIX B: PYTHON SCRIPTS

B.1. Routing Map Control and Flag Error Counter Read Script

Python script for configuring the routing map via control registers and retrieving error flag counters via status registers.

```
import uhal

manager = uhal.ConnectionManager("file:///home/cmx/Berkay/
    my_connections.xml")
hw = manager.getDevice("x0")

node = hw.getNode("payload").getNode("ipregtest")

# Read ECON-D Status Registers to print current error count

# ECON-D 1
econd1_rr = node.getNode("econd1_rr").read()
econd1_t = node.getNode("econd1_t").read()
econd1_i = node.getNode("econd1_i").read()
econd1_bx = node.getNode("econd1_bx").read()
econd1_ev = node.getNode("econd1_ev").read()
econd1_of = node.getNode("econd1_of").read()

# ECON-D 2
econd2_rr = node.getNode("econd2_rr").read()
econd2_t = node.getNode("econd2_t").read()
econd2_i = node.getNode("econd2_i").read()
econd2_bx = node.getNode("econd2_bx").read()
econd2_ev = node.getNode("econd2_ev").read()
econd2_of = node.getNode("econd2_of").read()

```

```
26 # ECON-D 3
27 econd3_rr = node.getNode("econd3_rr").read()
28 econd3_t = node.getNode("econd3_t").read()
29 econd3_i = node.getNode("econd3_i").read()
30 econd3_bx = node.getNode("econd3_bx").read()
31 econd3_ev = node.getNode("econd3_ev").read()
32 econd3_of = node.getNode("econd3_of").read()
33
34 # ECON-D 4
35 econd4_rr = node.getNode("econd4_rr").read()
36 econd4_t = node.getNode("econd4_t").read()
37 econd4_i = node.getNode("econd4_i").read()
38 econd4_bx = node.getNode("econd4_bx").read()
39 econd4_ev = node.getNode("econd4_ev").read()
40 econd4_of = node.getNode("econd4_of").read()
41
42 # ECON-D 5
43 econd5_rr = node.getNode("econd5_rr").read()
44 econd5_t = node.getNode("econd5_t").read()
45 econd5_i = node.getNode("econd5_i").read()
46 econd5_bx = node.getNode("econd5_bx").read()
47 econd5_ev = node.getNode("econd5_ev").read()
48 econd5_of = node.getNode("econd5_of").read()
49
50 # ECON-D 6
51 econd6_rr = node.getNode("econd6_rr").read()
52 econd6_t = node.getNode("econd6_t").read()
53 econd6_i = node.getNode("econd6_i").read()
54 econd6_bx = node.getNode("econd6_bx").read()
55 econd6_ev = node.getNode("econd6_ev").read()
56 econd6_of = node.getNode("econd6_of").read()
57
58 # ECON-D 7
```

```
econd7_rr = node.getNode("econd7_rr").read()
econd7_t = node.getNode("econd7_t").read()
econd7_i = node.getNode("econd7_i").read()
econd7_bx = node.getNode("econd7_bx").read()
econd7_ev = node.getNode("econd7_ev").read()
econd7_of = node.getNode("econd7_of").read()

# ECON-D 8
econd8_rr = node.getNode("econd8_rr").read()
econd8_t = node.getNode("econd8_t").read()
econd8_i = node.getNode("econd8_i").read()
econd8_bx = node.getNode("econd8_bx").read()
econd8_ev = node.getNode("econd8_ev").read()
econd8_of = node.getNode("econd8_of").read()

# ECON-D 9
econd9_rr = node.getNode("econd9_rr").read()
econd9_t = node.getNode("econd9_t").read()
econd9_i = node.getNode("econd9_i").read()
econd9_bx = node.getNode("econd9_bx").read()
econd9_ev = node.getNode("econd9_ev").read()
econd9_of = node.getNode("econd9_of").read()

# ECON-D 10
econd10_rr = node.getNode("econd10_rr").read()
econd10_t = node.getNode("econd10_t").read()
econd10_i = node.getNode("econd10_i").read()
econd10_bx = node.getNode("econd10_bx").read()
econd10_ev = node.getNode("econd10_ev").read()
econd10_of = node.getNode("econd10_of").read()

# ECON-D 11
econd11_rr = node.getNode("econd11_rr").read()
```

```
econd11_t = node.getNode("econd11_t").read()
econd11_i = node.getNode("econd11_i").read()
econd11_bx = node.getNode("econd11_bx").read()
econd11_ev = node.getNode("econd11_ev").read()
econd11_of = node.getNode("econd11_of").read()

# ECON-D 12
econd12_rr = node.getNode("econd12_rr").read()
econd12_t = node.getNode("econd12_t").read()
econd12_i = node.getNode("econd12_i").read()
econd12_bx = node.getNode("econd12_bx").read()
econd12_ev = node.getNode("econd12_ev").read()
econd12_of = node.getNode("econd12_of").read()

hw.dispatch()

# Printing the results
print("ECOND 1: rr={}, t={}, i={}, bx={}, ev={}, of={}".format(
    econd1_rr, econd1_t, econd1_i, econd1_bx, econd1_ev,
    econd1_of))
print("ECOND 2: rr={}, t={}, i={}, bx={}, ev={}, of={}".format(
    econd2_rr, econd2_t, econd2_i, econd2_bx, econd2_ev,
    econd2_of))
print("ECOND 3: rr={}, t={}, i={}, bx={}, ev={}, of={}".format(
    econd3_rr, econd3_t, econd3_i, econd3_bx, econd3_ev,
    econd3_of))
print("ECOND 4: rr={}, t={}, i={}, bx={}, ev={}, of={}".format(
    econd4_rr, econd4_t, econd4_i, econd4_bx, econd4_ev,
    econd4_of))
print("ECOND 5: rr={}, t={}, i={}, bx={}, ev={}, of={}".format(
    econd5_rr, econd5_t, econd5_i, econd5_bx, econd5_ev,
    econd5_of))
print("ECOND 6: rr={}, t={}, i={}, bx={}, ev={}, of={}".format(
```

```
    econd6_rr, econd6_t, econd6_i, econd6_bx, econd6_ev,
    econd6_of))
115 print("ECOND 7: rr={}, t={}, i={}, bx={}, ev={}, of={}".format(
    econd7_rr, econd7_t, econd7_i, econd7_bx, econd7_ev,
    econd7_of))
116 print("ECOND 8: rr={}, t={}, i={}, bx={}, ev={}, of={}".format(
    econd8_rr, econd8_t, econd8_i, econd8_bx, econd8_ev,
    econd8_of))
117 print("ECOND 9: rr={}, t={}, i={}, bx={}, ev={}, of={}".format(
    econd9_rr, econd9_t, econd9_i, econd9_bx, econd9_ev,
    econd9_of))
118 print("ECOND 10: rr={}, t={}, i={}, bx={}, ev={}, of={}".format
    (econd10_rr, econd10_t, econd10_i, econd10_bx, econd10_ev,
    econd10_of))
119 print("ECOND 11: rr={}, t={}, i={}, bx={}, ev={}, of={}".format
    (econd11_rr, econd11_t, econd11_i, econd11_bx, econd11_ev,
    econd11_of))
120 print("ECOND 12: rr={}, t={}, i={}, bx={}, ev={}, of={}".format
    (econd12_rr, econd12_t, econd12_i, econd12_bx, econd12_ev,
    econd12_of))
121
122 # Set the routing map for each ECON-D
123
124 node.getNode("ctrl1").write(1)
125 print("written")
126 hw.dispatch()
127
128 node.getNode("ctrl2").write(3)
129 print("written")
130 hw.dispatch()
131
132 node.getNode("ctrl3").write(4)
133 print("written")
```

```
134 hw.dispatch()
135
136 node.getNode("ctrl4").write(6)
137 print("written")
138 hw.dispatch()
139
140 node.getNode("ctrl5").write(5)
141 print("written")
142 hw.dispatch()
143
144 node.getNode("ctrl6").write(5)
145 print("written")
146 hw.dispatch()
147
148 node.getNode("ctrl7").write(5)
149 print("written")
150 hw.dispatch()
151
152 node.getNode("ctrl8").write(2)
153 print("written")
154 hw.dispatch()
155
156 node.getNode("ctrl9").write(7)
157 print("written")
158 hw.dispatch()
159
160 node.getNode("ctrl10").write(7)
161 print("written")
162 hw.dispatch()
163
164 node.getNode("ctrl11").write(2)
165 print("written")
166 hw.dispatch()
```

```
167
168 node.getNode("ctrl12").write(8)
169 print("written")
170 hw.dispatch()
171
172 # Disable desired ECON-D
173
174 for i in range(1, 13):
175     if i == 5:
176         node.getNode("disable{}".format(i)).write(1)
177     else:
178         node.getNode("disable{}".format(i)).write(0)
179
180 print("written")
181 hw.dispatch()
```

B.2. FIFO Data Capture Start Script

Python script to assert the data capture register and initiate data streaming through the firmware.

```
1  import uhal
2
3  manager = uhal.ConnectionManager("file:///home/cmx/Berkay/
       my_connections.xml")
4  hw = manager.getDevice("x0")
5
6  node = hw.getNode("payload").getNode("ipregtest")
7
8  node.getNode("capturer").write(1)
9  print("written")
10 hw.dispatch()
```

B.3. FIFO Read to CSV Script

Python script for reading Fast Command FIFO status and outputs and logging the results into a CSV file during EMP framework testing.

```
import uhal
import csv
import time
from datetime import datetime

manager = uhal.ConnectionManager("file:///home/cmx/Berkay/
    my_connections.xml")
hw = manager.getDevice("x0")

node = hw.getNode("payload").getNode("ipregtest")

def extract_bits(fifo_stat):
    full = (fifo_stat >> 12) & 1
    empty = (fifo_stat >> 11) & 1
    data_count = fifo_stat & 0x3FF
    return full, empty, data_count

with open('fifo_data.csv', mode='w', newline='') as csvfile:
    fieldnames = ['timestamp', 'fc_channel', 'fcc_fifo_out', '
        full', 'empty', 'data_count']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()

    while True:
        try:
            # Trigger FIFO read operation
            node.getNode("fifo_read").write(0)
            hw.dispatch()
```

```
28              node.getNode("fifo_read").write(1)
29              hw.dispatch()
30
31              for i in range(8):  # 8 FC FIFOs
32                  fifo_stat_key = f"fifo_{i+1}_stats"
33                  fifo_out_key = f"fifo_{i+1}_out"
34
35                  # Read FIFO status
36                  fifo_stat = node.getNode(fifo_stat_key).read()
37                  hw.dispatch()
38
39                  # Extract full, empty, and data_count bits
40                  full, empty, data_count = extract_bits(
                        fifo_stat)
41
42                  # Read FIFO output
43                  fifo_out = node.getNode(fifo_out_key).read()
44                  hw.dispatch()
45
46                  # Log the data with a timestamp
47                  writer.writerow({
48                      'timestamp': datetime.fromtimestamp(time.
                            time()).strftime('%Y-%m-%d %H:%M:%S.%f'),
49                      'fc_channel': i + 1,
50                      'fcc_fifo_out': hex(fifo_out),
51                      'full': full,
52                      'empty': empty,
53                      'data_count': data_count
54                  })
55
56                  # Print to console if wanted
57                  print(f"FIFO {i + 1} | Output: {hex(fifo_out)}
                        | Full: {full} | Empty: {empty} | Data Count:
```

```
                    {data_count}")
58
59              time.sleep(1)
60
61          except KeyboardInterrupt:
62              print("Terminated by user")
63              break
64          except Exception as e:
65              print(f"Error: {e}")
66              break
```