PixelIBL Testing Note

Pixel Operation Team

January 19, 2020

Contents

Chapter 1

Introduction

The idea behind this document is to describe in detail the procedures needed to validate the following items:

- HW setup in SR1 (ROD, BOC, Rx/Tx Plugin, Opto-boards and modules)
- Latest FW releases (both ROD and BOC) that will be produced during Run 3
- ROD Master PowerPC (PPC) and ROD Slave Micro Blaze (MB) software
- Host software running in fit farms or other server machines.

The test procedures can be divided in four main functional blocks, depending of what is the main functionality used to perform the tests:

- Stand alone application tests.
- Data taking with emulator.
- Data taking with real detector modules.
- Calibration with real detector modules.

Each of these blocks can have an immediate result or might require some post-processing analysis. The idea is to run some of these tests regularly to validate single parts of the read-out system and execute, if necessary, post-processing analysis. The purpose and the implementation of these analyses might change with time if further and/or better checks are expected. For this reason, the two parts should be developed separately and the test architecture should be flexible enough for future adjustment.

In Chapter 2, we will give a technical description of each single check, explaining in which conditions each test should be run and defining the expected values of the relevant registers/counters.

In Chapter 3, we will describe instead the logical implementation of more complex tests, that are typically a motivated sequence of more simple tasks as the one's describe in Chapter 2.

In Chapter 4, a schematic view of the implemented testing infrastructure will be presented, giving some details about the main test parameters and explaining the flexibility of such architecture.

Finally, in Chapter 5, a motivated guideline (based on Run 1 and Run 2 operational experience) of when and which tests are to be run before and during Run 3 operation is given.

Chapter 2

Technical description of the tests

2.1 Basic DAQ checks

2.1.1 ROD Register checks

From the directory */daq/RodDaq/IblUtils/HostCommandPattern/bin* , we can execute the following command:

```
./dumpRodRegisters –rodName ROD_C1_S7
```

to check that the main ROD registers are correctly set during data taking and/or calibration. Here below you can find a description of the of the main registers that are present in the output of the dumpRodRegisters command. Some of the registers are fixed and can be directly compared with a reference/expected value whilst some others depend on the ROD, the RunConfig, the moduleConfig or even the running conditions we are in (running, trigger rate, holding trigger,..).

ROD Master registers

- **RrifCtrlReg** Configuration register that should match the expected value (different between calibration and data taking).
- **SprCtrlReg** Configuration register that should match the expected value.
- **FeCmdMask0LsbReg/FeCmdMask1LsbReg** These registers represent the serial port masks respectively in the North and South ROD FPGAs.
- **EcrCntInReg** This counter records the amount of ECR received by the ROD Master since the begin of the run; it should match the number present in TimStatus and the one in the most significant part (8 bits) of the “most recent LVL1ID” field in the IS ROS dump. The counter wraps around since it has only 8 bits.
- **CalL1Id0Reg** This counter records the number of triggers received after the last ECR was received by the ROD Master; it should match the LVLID in TimStatus and also the lowest significant part (24 bits) of the ”most recent LVL1ID” field in the IS ROS dump.

- **CalL1Id1Reg** This counter records the number of triggers received by the ROD Master whilst the ROD was busy. It should always be "0" if the RODBUSY signal is correctly propagate from the TIM to the LTP.

- **CalPulseCount0/CalPulseCount1** This counter records the number of Calibration Pulse received by the ROD Master and forwarded to the North and South FPGA. This number should be always "0" during Data Taking; during Calibration should be equal to the number of trigger received (CalL1Id0Reg.. see above).

- **EfbMaskReg** This mask represents the EFBs that need to be mask since they don't have any channel enable (derived from the FmtLinkEnable_A and FmtLinkEnable_B masks). As a consequence, this register is module-mask dependent and could be very different between SR1 and the PIT or between different RODs.

- **BusytimeAfterEcrReg, VetoAfterEcrReg, ResetAfterEcrReg** Special configuration registers that should match the values in the Ref. document.

- **EcrVetoedAfterEcrReg, L1AVetoedAfterEcrReg** These counters should be always "0" otherwise the entire ROD is desynchronized.

- **BcrVetoedAfterEcrReg** This counter should be different from "0"; no impact on data taking is expected from that.

- **EcrVetoedAfterEcrReg, BcrDroppedReg, L1ADroppedReg** This counter should be always "0" otherwise some problem with the sequence of the TTC commands coming from the TIM is present.

ROD Slave registers

- **FmtLinkEnable**_A/B Check that all the modules that are part of the PixDisable are correctly enable in this register. Depending of the adopted readout-speed this mask could look like different.

- **SlaveId**_A/B It should be "0" for the Slave North FPGA and "1" for the Slave South FPGA.

- **CalibrationMode**_A/B It should be "0" during data taking and "1" during calibration (it will switch off the SLINK fragment production for the latter case).

- **SourceId**_A/B. This "Hex" number should match the ROB_ID of the corresponding ROL of the SLINK0 connected to the North FPGA (check the connectivity file).

- **TriggerCountNumber**_A/B This register should match the readout window size that is selected inside the RunConfig. This variable is also reported as "Consecutive LVL1" or "Number of frame to read-out" depending on the location. It is typically "1" in p-p data taking, it can be > 1 for Cosmic data taking (3 for IBL and 5 for Pixel) and Calibration(typically 16 frames to be readout.)

- **RunNumber**_A/B This number should match the RunNumber present in the DAQSlice UI.

- **CodeVersion**_A/B This parameter should be "0xfe13" for Pixel RODs and "0xfe14" for IBL RODs.

Register Name	Default Value (from FW)	Reference Value (0xFF otherwise)	Conditions
RrifCtrlReg ””		0x88001527 0x88001528	DataTaking CalibraTion
SprCtrlReg		0x878	
FeCmdMask0LsbReg FeCmdMask1LsbReg		0xFF 0xFF	
EcrCntInReg		0xFF	Hold-Trigger
CalL1Id0Reg		0xFFFFFF	Hold-Trigger
CalL1Id1Reg		0x0	
CalPulseCount0		0xFFFFFFFF	Hold-Trigger
CalPulseCount1		0xFFFFFFFF	Hold-Trigger
EfbMaskReg		0xF	
BusytimeAfterEcrReg		0x2328	
VetoAfterEcrReg		0x1fa4	
ResetAfterEcrReg		0x1fa4	
EcrVetoedAfterEcrReg		0x0	
BcrVetoedAfterEcrReg		0xFFFFFFFF	
L1AVetoedAfterEcrReg		0x0	
EcrDroppedReg		0x0	
BcrDroppedReg		0x0	
L1ADroppedReg		0x0	

Table 2.1: Description of the Rod Master registers that needs to be checked right after the configuration or during running. The expected value of the register is also given togheter with the condition for which the value should be checked.

- **FmtReadoutTimeoutLimit**_A/B**, FmtDataOverflowLimit**_A/B**, FmtHeaderTrailerLimit**_A/B**, FmtRodBusyLimit**_A/B**, FmtTrailerTimeoutLimit**_A/B All of these registers should match the expected values that might/should be different for Pixel vs IBL RODs.

- **Fmt0OccupancyReg**_A/B**, Fmt1OccupancyReg**_A/B**, Fmt2OccupancyReg**_A/B**, Fmt3OccupancyReg**_A/B All of these registers should be "0" whenever we are holding a trigger and dumping the content; if this counters are >0 it means that some junk is stucked in the FMT FIFO.

Register Name	Default Value (from FW)	Reference Value (0xFF otherwise)	Conditions
FmtLinkEnable_A/B		0xFFFF	
SlaveId_A/B		0x0/0x1	
CalibrationMode_A/B		0	Data Taking
""		1	Calibration
SourceId_A/B		0xFFFFFF	
TriggerCountNumber_A/B		0xF	
CodeVersion_A/B		0xfe13	Pixel ROD
""		0xfe14	IBL ROD
FmtReadoutTimeoutLimit_A/B		0x801	Pixel ROD
""			IBL ROD
FmtDataOverflowLimit_A/B		0x801	
""			IBL ROD
FmtHeaderTrailerLimit_A/B		0x801	
""			IBL ROD
FmtRodBusyLimit_A/B		0x3c1	
""			IBL ROD
FmtTrailerTimeoutLimit_A/B		0xf50	
""			IBL ROD
Fmt0OccupancyReg_A/B		0x0	
Fmt1OccupancyReg_A/B		0x0	
Fmt2OccupancyReg_A/B		0x0	
Fmt3OccupancyReg_A/B		0x0	

Table 2.2: Description of the Rod Slave registers that needs to be checked right after the configuration or during running. The expected value of the register is also given togheter with the condition for which the value should be checked.

2.1.2 BOC Register checks

In order to check some basic BOC registers/counters, we need first to access the library that is used to interact directly with the BOC. One should execute the following commands:

```
cd /home/mbindi/daq/RodDaq/IblUtils/Boc2
```

```
source ./setup.sh
```

and finally dump the registers values:

```
dumpRegisters --ip BOC_IP
```

BOC BCF registers

Here are the relevant registers to check in the BOC BCF:

- **BOC clock source:** PLL buffered 40 MHz from VME
- **Global Idelay:** 0
- **VME Clock Frequency:** 40077521 Hz

The BOC during data taking should get its clock from the TIM (via VME back plane). In case *"Internal"* is present, the BOC is not correctly configured for data taking. The clock value should be $\sim$ 40 MHz $\pm$ 100 kHz.

BOC BMF registers

Most of the BMF North and South registers are exactly the same; the only thing that differs is the mapping of the channels that depends on the ROD/BOC slot under test (SR1 vs PIT, IBL vs Pixel). We will proceed describing the checks to be performed on the BMF registers/counters, separating them based on their functional block: General and SLINK, Tx Channel and Rx Channel.

General or SLINK registers

- **BMF RX speed :** the first thing that needs to be checked in case we are testing a Pixel BOC is the RX readout speed settings. These are the tipycal configuration settings for the PIT:
  - 40 Mbit/s used only during Calibration (in all the layers),
  - 80 Mbit/s used for Ly2, Disk1 and Disk 3 during data taking,
  - 2x80 Mbit/s used for Disk2, Layer 1 and B-Layer during data taking.

  For testing porposes, these value can be different from what indicated above. In any case, whatever is in the BOC BMF dump should match the RunConfig parameters in IS.
  The "BMF RX speed" variable is not present in the dump of the IBL BOC since it will be always running at 160 Mb/s.
- **SLINK0/1 Control:** For a Pixel BOC, only SLINK0 Control should be checked; it should be always "0x90" that means it is ignoring completely the status of the FTK link (both "link LDown" and "link Xoff" conditions are ignored).
- **SLINK0/1 Status:** For a Pixel BOC, only SLINK0 Status should be checked; it should be "0xe4" or "0xf4" depending of the status of the running conditions. Typically, when the link is active and transferring data, the "link active" status bit is set to 1. This differentiates "0xe4" ("Slink non

Functional block	**Register Name**	**Default Value (from FW)**	**Reference Value (0xFF otherwise)**	**Conditions**
General	BMF RX speed	40	40, 80, 160	Relevant only for Pixel
Slink	SLINK0/1 Control	0x60	0x90	
Slink	SLINK0/1 Status	0xa2	0xe4/0xf4	Hold-trigger/triggering
Slink	SLINK0/1 Hola0 Xoff			
Slink	SLINK0/1 Hola0 Ldown			
Slink	SLINK0/1 Hola1 Xoff			if FTK inlcuded
Slink	SLINK0/1 Hola1 Ldown			if FTK included
TxChannel	Configuration		40Mbit/s	
TxChannel	”		BPM enabled	
TxChannel	”		ROD input	Data taking
TxChannel	”		Tx FIFO	Configuration
TxChannel	Trigger count		0xFFFFFFF	Hold-trigger
TxChannel	ECR count		0xFFFFFFF	Hold-trigger
TxChannel	BCR count		0xFFFFFFF	
TxChannel	CAL count		0xFFFFFFF	Hold-trigger
TxChannel	SYNC count		0xFFFFFFF	Hold-trigger
TxChannel	Slow command count		0xFFFFFFF	Hold-trigger
TxChannel	Corrupted command count		0	
TxChannel	ECR Enabled			IBL only
TxChannel	ECR Busy			IBL only
TxChannel	Collision Flag		0	IBL only
TxChannel	Veto from QS			IBL only
RxChannel	Rx	Off	On	Enabled channels
	”		Off	Disabled channels
RxChannel	Selected input	Optical		Real modules
”	”	FE-Emulator		FE-Emulator
RxChannel	ROD-Output		1	
RxChannel	Inverted		1	
RxChannel	FIFO-Output			
RxChannel	Received Frames		0xFFFFFFF	
RxChannel	Locked		YES	
RxChannel	Synced		YES	
RxChannel	Decoding errors		NO	IBL only
RxChannel	Disparity errors		NO	IBL only
RxChannel	Frame errors		NO	IBL only
RxChannel	Dec. err Counter		0xFFFFFFF	IBL only
RxChannel	Disp. err Counter		0xFFFFFFF	IBL only
RxChannel	Frame err Counter		0xFFFFFFF	IBL only

Table 2.3: Description of the BOC BMF registers that needs to be checked right after the configuration or during running. The expected value of the register is also given togheter with the condition for which the value should be checked.

active") from the "0xf4" ("Slink is active"). Any other value for the SLink status register might indicate a mulfunctioning on the link communication (problem on one QSFP plugin, ROL fiber disconnected/broken, ROS Robin Fw not fully configured for data taking, ...)

- **SLINK0/1 Hola0/1 XOff/LDown:** the number of "Xoff" or "LDown" occurrencies can be a number different from "0" but it should be a fixed number (incremented typically during the start up of the data taking) and should never increase during a run.

Tx Channels

Each TX channel can be configured independently to take input data from the ROD serial lines or from the TX FIFO; each Tx channel has also a monitoring mechanism that can be activated to keep track of all the commands (fast commnands like Trigger/ECR/BCR/CAL pulse or slow commands like the configuration stream) that are propagated to the modules(FEs) or to the Emulators. Here below we present a description of the various configuration registers and monitoring counters.

If we want to check the status of the TXchannel and the propagation of fast commands (like ECR, BCR or L1A) or slow commands (like configuration commands), we need to enable the TXFIFO Monitoring tool using one of the the BOClib functions:

```
IblBocMon (or PixBocMon) --ip BOC_IP
```

Select *"txmon_enable"* once you are in "CONNECTED", right before the start of the Run. After some running time, `\Hold the trigger!"`. Finally, select *"txmon_show"* in order to check the counter values.

Each TX channel should have the same amount of Slow and Fast commands except for the BCRs counters that will keep increasing despite the trigger beeing on-hold. The number of trigger sent should match the one present in the DAQ slice. The number of ECR sent should match the one present in the IS of the DAQ Slice (variable *RunParams/ECRCounter*). The other counters like ECRcounter 8 bit should match instead the EcrCntInReg that is part of the ROD Master registers checks.

A very important task of the TX in the BOC is to perform regular reconfiguration actions during the ECR time window (2 ms). Typically this task is executed by the PPC. The important check to perform in this case is that the "Collision Flag" value is FALSE (or 0) meaning that there were no conflict between commands sent by the ROD and the one created inside the BOC FIFO to reconfigure the module/FEs.

ECR Reconfiguration Status: Enabled 0 Busy: 0 Collision Flag: 0 Veto from QS Reconfig Active: 1

In case the monitoring counters are all 0s... it means that the monitoring mechanism was likely not activated.

In case some problems are detected into the TX data stream, the *"Corrupted command count"* will be different from "0". This is a strong indication that something upstream is not properly working!

Rx Channels

Each RX channel can be configured independently to receive data internally to the BOC (from the MCC/FE Emulator) or externally via SNAP12 RX Plugin (from the Optical Channel) that is connected to the detector module/FE via optical fiber; each Rx channel has also frame and error counters that allow to monitor the data flow and understand if there is some problem with the tuning of the optical link. IBL and Pixel BOC differ each other mostly for a different Rx channel instance. Here below and example for

both cases. In the case of IBL, decoding and disparity errors are present since the IBL 8b-10b encoding done from the FE-I4 allows to check these type of errors.

2.1.3 TIM Register checks

In order to check some basic TIM counters/registers we can check via RodMon the values published in IS. Here is the list of variables that should be checked **after the start of the Run and while holding the trigger**:

- **ECR counter value :** same as ROD Master register value (see ROD section)
- **RodBusyDuration (seconds):** it should be very small number, and similar for all the crates under test. Typically we set artificially in the ROD Master a busy time $<$ 1ms at each ECR.
- **RodBusyDuration** value should be $<\sim (ECRreceived) * 1$ ms
- **TimBcidOffset:** it should be equal to "13"
- **TimTriggerDelay:** this number should match what is specified in the RunConfig
- **TTCrxFineDelay:** this number should match what is specified in the RunConfig
- **TTCrxCoarseDelay:** this number should match what is specified in the RunConfig

Chapter 3

Logical description of the tests

This chapter describes in detail the logic behind the scripting of each of the tests written to validate the items mentioned in Chapter 1.

The following tests are located in the path */pixeldaqtest/testlibrary/*.

3.1 TestBusy

- Input: Reads IS Variables
- Script Logic:
  - The first part of the code searches for the busy signal pertaining to the ROD selected and checks its value. If the value is greater than "0", the ROD is said to be in BUSY state.
  - Once it is established that a ROD is busy, we further aim to generate a list of signals that cause the ROD to be busy. For this, we open a pre-generated list from the Information Service (IS) monitor with all signals having the keyword "BUSY". Then we parse through the list to check their respective values. The signals with values greater than "0" are then listed out as the busy causing signals for the specific ROD.
- Output: Saved in file "busy_out.txt". Prints the BUSY status of the ROD and busy causing signals.

3.2 TestFragError

- Input: Reads IS variables
- Script logic:
  - In the first part of the test we generate a list called "rol_en", which stands for "Readout link enabled". Depending on which ROD is being tested, the length and contents of rol_en list is decided.

- In the second part of the script, we perform our first check. This check confirms whether the number of triggers sent are equal to the fragments received. The number of triggers sent, and the number of fragments received are both obtained from the IS monitor.
- In the next check, we read the IS for other fragment related errors like- Fragment size mismatch, fragment format error, fragment marker error etc. These errors are displayed for each ROD that is selected.
- The next check involves checking whether the Tx errors are in sync. Synchronization of Tx errors can be understood as the following:
  * Tx error1 = Fragment Framing Error + Fragment Size Mismatch Error + Fragment Marker Error
  * Tx error2 = Fragments Corrupted – Fragments Rejected
  * When Tx error1 = Tx error2, the Tx errors are said to be in sync, else a error is raised.
- The last check is called the Xoff check. The number of xoff per ROD is determined at the first hold of the trigger. This value must remain constant at all subsequent holds. To check this, a list called "xoff.txt" that is generated at the first hold is opened. The values present in this list is then compared with the xoff values at each hold of the trigger. In case of any discrepancies, an error is recorded.

- Output: Saved in JSON format as "frag_out.json". Prints fragment error name along with the numbered of errored fragments.

3.3 TestRodMon

- Input: Reads IS Variables
- Script logic:
  - In the first part of the script, we create 3 lists. The first list consists of the headers used for the final result, the second tables consists of keywords that are used throughout the script to look for related errors and the third list consists of default values corresponding to the keywords from list2.
  - In the next step, we open a pre-saved list that consists of all signals corresponding to all RODS. The list is then formatted for easy parsing. A new list called "res" is formed here that only has signals corresponding to the ROD under test.
  - In the function "general_func", we use the list "res" to look for errored signals. We parse through the list looking for the keywords from list2 (point (i)) and compare their values with the default values from list3. This function return the list of errored signals with their values.
  - The next section of the script is a check to see all the modules that are enabled in the ROD under test. For this, we first create a list called "filtered_list" with the keyword "FMTSTAT". For each module enabled in the ROD, the FMTSTAT value would be "E" (enabled). This count is saved under the variable name "fmtstat".
  - In the next step, we check if the DAQEnabled for each ROD matched the FMTSTAT value calculated in the previous step. DAQEnabled also depends on the modules enabled and hence these values must match. This count is saved under the variable name "daq_en".

- The function "rand" is used to check if the ROD under test is AVAILABLE. We use the keyword "AVAILABLE" to look for the signal and check if its value is "0". A zero value means that the ROD is available. NOTE: The list created using the keyword "AVAILABLE" must only contain one element as each ROD only has one of these signals.
- The function "num_gen" is just a number generating function that assigns the parameter values to other functions like "general_func".

- Output: Saved in JSON format as "RodMon_out.json". Prints the various checks mentioned above with their corresponding values.

3.4 TestTimeout

- Input: Reads IS variables
- Script logic:
  - In the first part of the code, we open a pre-saved list called "timeout_list.txt". We then parse through this list to search for the timeout signal pertaining to the ROD selected and checks its value. If the value is greater than "0", the ROD is said to be in TIMEOUT state.
- Output: Saved in a file "timeout_out.txt". Print the timeout status of a ROD.

3.5 TestRodMaster

- Input: Reads ROD dump
- Script logic:
  - In the first part of the script, we obtain the ROD dump and save it as a csv file. This file is used for all analysis in this script.
  - As in the test "TestFragError", we generate a list called "rol_en", which stands for "Readout link enabled". Depending on which ROD is being tested, the length and contents of rol_en list is decided.
  - We then obtain the values from the IS monitor that must match the ROD master registers under check. These values are the most significant bit(msb) and least significant bit(lsb) of the "mostRecentL1ID".
  - In the next step we create a csv with the expected values of each of the ROD master registers under check. This csv will be used against the values obtained from the ROD dump to check for errors.
  - For easy comparison of register values, we use a python3 pandas package. This package can convert csv to a pandas dataframe and allows manipulation of the dataframes under test. As this package needs python3, the next steps are saved in a python3 file saved in the path */testlibrary/py3_scripts/rod_master_py3.py*.
  - In the script "rod_master_py3.py", we first extract all relevant ROD master registers and form a csv with actual values. We then create two dataframes from the available csv files. One is the Expected dataframe and the other is the Actual dataframe.

- The function "reg_error" is used to compare the two dataframes and create a third dataframe with the register values that did not match the expected.

- Output: Saved in a file "Rod_Master_out.txt". Prints the ROD masters register values that do not match the expected values.

3.6 TestRodSlave

- Input: Reads ROD dump

- Script logic:

  - In the first part of the script, we obtain the ROD dump and save it as a csv file. This file is used for all analysis in this script.
  - As in the test "TestFragError", we generate a list called "rol_en", which stands for "Readout link enabled". Depending on which ROD is being tested, the length and contents of rol_en list is decided.
  - We then obtain the values from the IS monitor that must match the ROD slave registers under check. This is the run number value which is saved under the variable name "run".
  - As the ROD slave register values are different for PIXEL vs IBL, we first save the default values for these registers for Pixel and IBL.
  - After this, we create a csv with the expected values of the ROD slave registers for Slaves A and B. We now direct to the python3 script for further analysis. The script is located at */testlibrary/py3_scripts/rod_slave_py3.py*.
  - In the script "rod_slave_py3.py", we first extract all relevant ROD slave A registers and form a csv with actual values. We then create two dataframes from the available csv files. One is the Expected dataframe and the other is the Actual dataframe.
  - The function "reg_error" is used to compare the two dataframes and create a third dataframe with the register values that did not match the expected. The same steps are repeated for Slave B.

- Output: Saved in a file "Rod_Slave_out.txt". Prints the ROD slave A and B register values that do not match the expected values.

3.7 TestBocBcf

- Input: Reads BOC dump

- Script logic:

  - NOTE: In this script it is assumed that the BOC dump is available and stored in the path */testlibrary/dumps/boc_dump.csv*. Please ensure the filename "boc_dump.csv" is also updated in the script */testlibrary/py3_scripts/bcf_py3.py*.
  - In the script "bcf_py3.py", we first parse through the BOC dump and look for the keyword "Internal". If the keyword "Internal" is present, it means that the BOC is not correctly configured for data taking and this raises an error.

- The next check is for the frequency of the VME clock. Ideally, the frequency must lie within +/-100KHz of 40MHz. iv. We create two dataframes with actual and expected values of the BOC BCF registers and compare them using the function "reg_error".

- Output: Saved in a file "BOC_BCF_out.txt". Prints the BOC BCF register values that do not match the expected values.

3.8 TestBocBmf

- Input: Reads BOC dump

- Script logic:

  - NOTE: In this script it is assumed that the BOC dump is available and stored in the path */testlibrary/dumps/boc_dump.csv*. Please ensure the filename "boc_dump.csv" is also updated in the script */testlibrary/py3_scripts/bmf_py3.py*.
  - In the script "bmf_py3.py", we first create lists using keyword search for all the BMF registers to be checked. The function "bmf_N_S" is used to separate the registers for North and South side.
  - We define the default values of the registers and then create actual and expected dataframes for comparison. This is done for both BMF North and BMF South registers.

- Output: Saved in a file "BOC_BMF_out.txt". Prints the BOC BMF register values that do not match the expected values.

3.9 TestBocTx

- Input: Reads BOC dump

- Script logic:

  - NOTE: In this script it is assumed that the BOC dump is available and stored in the path */testlibrary/dumps/boc_dump.csv*. Please ensure the filename "boc_dump.csv" is also updated in the script */testlibrary/py3_scripts/tx_py3.py*.
  - In the script "tx_py3.py", we create lists using keyword search for the registers to be checked for each Tx line. The function "tx_N_S" is used to separate the registers for the North and South Tx lines.
  - We create two lists that contain the name and expected values of the Tx contents respectively.
  - The function "count_error" is used to compare the actual and expected values of the registers in the Tx lines. This function outputs the registers whose values do not the match the expected values.

- Output: Saved in a file "BOC_TX_out.txt". Prints the BOC TX line register values that do not match the expected values.

Chapter 4

Testing architecture

Chapter 5

Guidelines for tests during Run 3 operation

Appendices

Bibliography