ATLAS NOTE

November 20, 2015

CMX review - software documentation and CMX FW logic tests

CMX project team - main author Duc Bao Ta, firmware: Wojtek Fedorko, Pawel Plucinski.

Abstract

This document describes the software that is being used for the ATLAS L1Calo CMX either for the operation online, for the online simulation or for diagnostic purposes. This document also describes the test that were performed with the help of the diagnostic software for the different CMX flavours.

© Copyright 2015 CERN for the benefit of the ATLAS Collaboration.
Reproduction of this article or parts of it is allowed as specified in the CC-BY-3.0 license.

Contents

1 Introduction

This document describes the software that is being used for the ATLAS L1Calo CMX either for the operation online, for the online simulation or for diagnostic purposes. This document also describes the test that were performed with the help of the diagnostic software for the different CMX flavours. The CMX comes in 6 different flavours, crate-type and system-type with JET, CP and ENERGY summing functionality. The CP CMX will have two different TOB object inputs (and hence two different sets of thresholds), EM and TAU TOBs. The CP (=EM or TAU) TOBs will come from CPMs, while the JET TOBs will be provided by JEMs. The main functionality of the CMX is to count the number of TOBs over programmable thresholds (cluster multiplicity counts, thresholds counts) in certain detector regions and add them to the total counts. In case of the ENERGY CMX the energy is summed over the all detector regions and compared to the thresholds. The whole detector is covered by 4 CP crates each with 14 CPMs and each with two CMX (for EM and TAU). The coverage is roughly $\eta = -2.5 - -2.5$ in 50 η-slices with a granularity in most areas of $\eta \times \phi = 0.1 \times 0.1$. Jets and energy values are found in 2 JET/ENERGY crates each with 16 JEMs and two CMX (one CMX is reponsible for the JET thresholding and summing, the other for the energy summing). The coverage is roughly $\eta = -4.9 - -4.9$ in 32 η-slices with a granularity in most areas of $\eta \times \phi = 0.2 \times 0.2$. Figures 1 and 2 show the coverage of the modules as obtained from the `l1calomap.sh` tool. A more detailed description of the CMX can be found here [1, 2, 3, 4, 5, 6]. Most of the functionality of the system in run-1 is described here [7, 8, 9, 10, 11, 12] and some algorithms are similar for the new system. The reference also give a rough overview of the L1Calo system.

The software concerning the CMX modules can be mainly found in the `cmxServices`, `cmxSim` and `cmxTets` packages. Some general interfaces and classes can be found in the `infraL1Calo` (mainly `Bitcoder.h`) and in the `linkSim` packages. Nightly builds of the L1Calo software can be found here [13]. Other notable dependencies are the database interfaces which can be found in `dbL1Calo`. The description of the L1Calo databases can be found here [14].

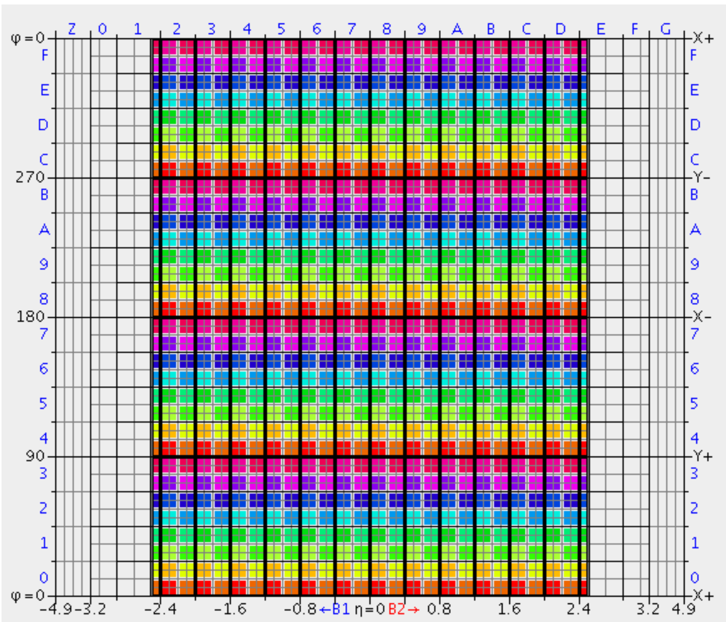


Figure 1: Schematic representation of the detector coverage. The gridlines are the calorimeter cells, the thick outlines are the coverage areas of the CPM modules and the colour-filled areas represent the different areas covered by different presence bits.

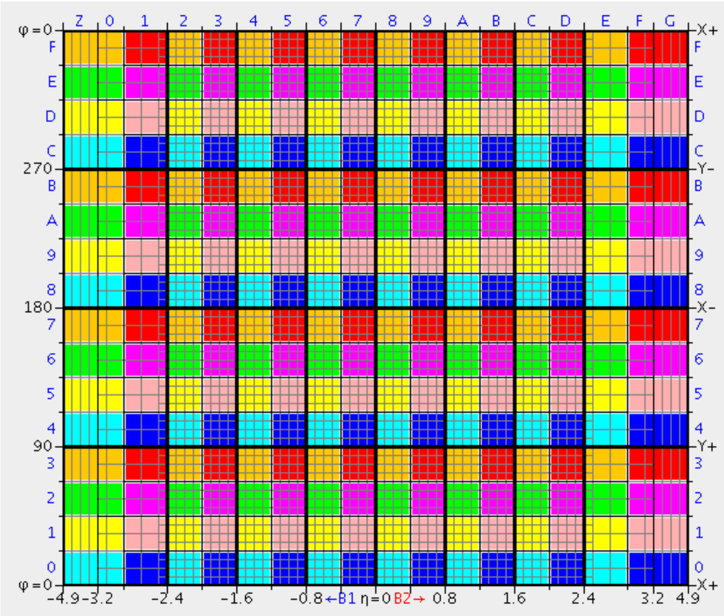


Figure 2: Schematic representation of the detector coverage. The gridlines are the calorimeter cells, the thick outlines are coverage areas of the JEM modules and the colour-filled areas represent the different areas covered by different presence bits.

2 Software documentation

The software documentation is meant as an overview of all software parts that the CMX specific software uses. An overview over the classes and functions are given, as well as instructions how to run standalone codes. The online software and online simulation is part of the L1Calo software framework, that is not described in this document, but here [15]. Also the base classes from which the classes in the online software derives is only described, if the functionality has been adopted for the CMX. Code that is deprecated is not mentioned.

2.1 Online software - L1CaloCmx class

The online software consists of the software package `cmxServices` within the L1Calo software. Its main functionality lies in the `L1CaloCmx::DaqModule` class. This class is usually instantiated if access to the CMX module is needed, e.g. by the standalone code described below or by the ATLAS run control (RC) software. In conjunction with the L1Calo database it will automatically assign the correct VME addresses for the registers (depending on position in the crate, see [16]). The code is commented using the doxygen documentation generator.

2.1.1 General helper functions

Several functions are available to determine the type and location (crate and position in the crate) of the CMX. The type/flavour and location (geographical location, see Table 7) are based on the database setting. The flavour of the CMX firmware that is loaded can be checked by registers in the CMX (the location defines the flavour of CMX, but firmware not matching to the location of the CMX can be loaded). If there is a mismatch, the hardware setting is preferred. This is useful when messages regarding a certain CMX are printed or published to the ATLAS information system (ERS). Other functions convert values, read/write memory registers or get the status of ATLAS/ATLAS RC and LHC. Here is the list of general helper functions:

```
const DbCmx *getDbCmx() const // get a pointer to the database.
unsigned int myCrate() // returns the crate number from the database.
unsigned int myLeftRight() // returns the 0 for left and 1 for right position in the crate as indicate in the
    database.
std :: string getCMXIDstring( int details =-1 ) // returns a string as default 'CMX crate_number leftright '.
uint32_t getVersionCommon() // returns the common version of the CMX firmware.
uint32_t getVersionFlavourCommon() // returns the common flavour version of the CMX firmware.
uint32_t getVersionFlavourLocal () // returns the local flavour version of the CMX firmware.
uint32_t convertDBAddr2CFAddr( uint32_t addr ) // converts the database address to the CF slot.
uint32_t convertDBAddr2GeoAddr( uint32_t addr ) // converts the DB address to the geographical address used in
    the hardware.
uint32_t convertGeoAddr2DBAddr( uint32_t addr ) // converts the geographical address to the DB address.
uint32_t convertGeoAddr2CFAddr( uint32_t addr ) // converts the geographical address into the CF slot number.
uint32_t convertCFAddr2CrateSystem( uint32_t addr ) // converts the CF slot number to left / right position of the
    CMX.
CmmFirmwareType convertCFAddr2Firmwaretype( uint32_t addr ) // converts the CF slot number to the
    CmmFirmwareType.
std :: string convertFWtype2String( CmmFirmwareType cmmfwtype ) // converts the firmware type into a readable
    string .
CmmFirmwareType myFirmwareType() // returns the 'CmmFirmwareType'.
bool mySystemLevel() // returns true for a system type CMX.
//
int foldDelay( const int K ) // converts the delay 'K' value into the 'mn' delay value used for DS1 (deskew-1)
    and DS2 (deskew-2) fine delays on the TTCDec card.
int unfoldDelay( const int mn ) // converts the 'mn' value back into the 'K' value.
//
float convert_sysmon_temp( unsigned int adc ) // converts the ADC value from the system monitoring into a
    temperature value.
float convert_sysmon_volt ( unsigned int adc ) // converts the ADC value from the system monitoring into a
    voltage value.
//
uint32_t read32bitword( ModuleMemory16 * memory, int offset=0 ) // read a 32-bit word ( least significane word (
    LSW), then HSW) at position ' offset ' (counting the number of 32-bit word) in a 'ModuleMemory16'.
void write32bitword ( ModuleMemory16 * memory, int offset, uint32_t value ) // write a 32-bit word (LSW, then
    HSW) at position ' offset ' in a 'ModuleMemory16'.
//
float rounding( float number, unsigned int digits ) // convenience function to do rounding of numbers.
//
std :: string getLHCstatus() // get the string that signals the LHC status (e.g. 'RAMPING', 'FLAT TOP', etc.).
void getATLASstatus( std :: string &AtlasRunState, int &RunNumber, int &LB ) // get the ATLAS RC status (e.g. '
    RUNNING', 'CONFIGURE', etc.) and run number.
std :: string getATLASrcState() // only get the ATLAS RC status.
std :: string timestamp() // get a string with the timestamp.
bool writeandverifyRegister (ModuleRegister16 * register , uint32 value) // writes and verifies the value written
    to the register . On a long run, this should replace all the code that writes to registers .
```

The fine delay of the clocks in the TTCDec card [17] in K values is described in Section 3.3.2. `CmmFirmwareType` refers to the definitions for the CMM, see [7]. The CF slot is described in Section 2.6.

2.1.2 CMX FSM transitions

Most of the functions that relate to the state changes of the CMX are inherited from the base class. Here only functions are mentioned that were changed for the CMX. Figures 3 and 4 show the ATLAS RC panel for the L1CaloStandalone partition as an illustration of the L1Calo segment and the FSM state transitions.

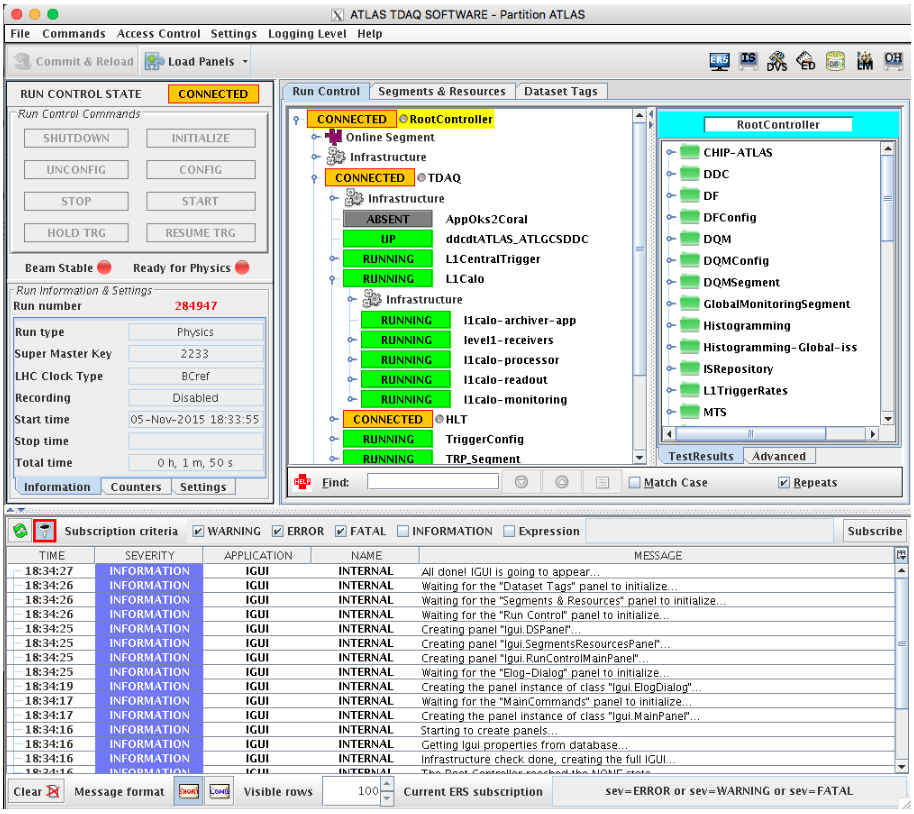


Figure 3: ATLAS run control panel for the L1CaloStandalone partiton

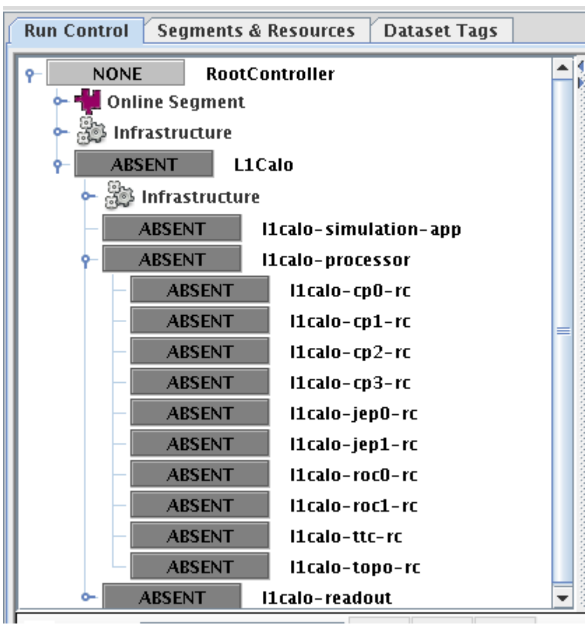


Figure 4: The L1Calo segment in the run control tree.

The following functions are called when the CMX is going though state changes in the ATLAS run control:

- `initModule()`: all registers are set to a default value (at the moment this function does no do anything, so the initial values are the remaining value from the last run).
- `load()`: INITIAL → CONFIGURE
  - First the clock settings on the TTCDec card are set such that the correct input clock is taken. The checks on the PLL ready flags (all flags in the module status register about the locking of the PLLs) and various other status flags are done (`checkGeneralStatus()`, checking SystemAce status, BF configuration done, TTCRx ready, SFP DAQ/ROI PLL lock status and GTX/RX ready status) provided by the board support FPGA (BSPT) [18].
  - A check is made on the OKS database (`geoForceAddr`, `geoAddr`), if the geographical address is forced. In that case a reload of the firmware through the reset of the SystemAce is requested. Every 20 s the SystemAce status and the `BFConfigDone` flag is checked. Within 2 min these flags must be set and the CMX flavour in the flavour register must match the requested flavour (conversion of geographical address to CMX flavour). If the flags are good, but the flavour is different the reset is repeated until success or the 2 min have expired. ERS warnings are shown when a new firmware is loaded and ERS fatales are thrown in case of failure.
  - Afterwards the data polarity of the links to L1Topo are set (`setTopoPolarity()`), the TTC fine delay for the clocks (DS1 and DS2) are set from the database (see 2.1.3, `setClocks()`) and also the TTC coarse delay of the clocks is checked and set to zero.
  - The clock manager is reset at this stage for the first time (`resetClockMananger()`), followed by a reset of the TopoGTX links (`resetTopoGTX()`).
  - Checks on the firmware versions (`checkBSPTFWversion()`), on the clock settings (`checkClockSettings()`) on the PLL ready flags and various other status flags are checked again. ERS warnings are shown if the firmware does not match any known version. Fatales are only thrown if the firmware versions are too old (i.e. checked against hardcoded lists of valid firmware versions). In case of failure of the checks, ERS fatales are thrown.
  - Finally all counters and error counters are reset (`resetAllCounters()`, `resetErrorCounters()`).
- `configure()`: INITIAL → CONFIGURE
  - All spy memories are set to spy mode and the memory writing is un-inhibited (`systemSpyMemory ->setDPRmode(GenericCMXSpyMemory::DPR_CONTROL_SPY)`, `unsetSpyMemoryInhibit()`).
  - Various items are configured from the database: TTC clock delay phases, backplane fine delays (i.e. delays on the data and forwarded clock lines on the backplane), relative readout pointers for each line in the readout (offset per glink pin minus offset of the backplane data), DAQ offset (offset of backplane data), number of readout slices, BC offset, disabled masked for the backplane and (for system-type CMX) the mask for the RTM cables. An abbreviated excerpt of the configuration procedure is shown below.
  - The trigger thresholds are set according to the flavour of the CMX.
  - The clock manager is reset and the status is checked again.
- `prepareForRun()`: CONFIGURE → RUN
  - All counters, error counters and rate counters are reset.

Here is an excerpt of the code that configures the CMX:

```
// set TTC clock delay phases
setDeskew1(db−>getTtcrxPhase1());
setDeskew2(db−>getTtcrxPhase2());
// set quiet/force flag
writeandverifyRegister ( p_quietForce , 1 , "quietForce");
// set backplane delays
for (int n=0; n<16; n++){
    std :: vector < unsigned int > delays = db−>getBackplaneInputDelays(n);
    int count=0;
    for (auto elem: delays) {
        if (count>24) break;
        setIDELAY(n,count,elem);
        count++;
    }
}
// relative readout pointers
for (int n=0;n<20; n++){
    unsigned int calcualtedoffset =( ( db−>getDaqOffsetForPin(n)− db−>getDaqOffsetBpData() ) & 0xff);
    if (n!=0) writeDAQRAMrelativeOffset(n−1, calcualtedoffset );
}
// DAQ slice, DAQ offset, BC offset
writeandverifyRegister (p_DAQSlice, (db−>getNumDaqSlices() −1 )/2, "DAQslices" );
int DAQRAMOffset = db−>getDaqOffsetBpData();
int BCOffset = db−>getBcOffset();
writeandverifyRegister (p_DAQRAMOffset,DAQRAMOffset,"DAQRAMOffset");
writeandverifyRegister (p_BCIDResetVal,BCOffset,"BCOffset");
// Pipedelay
setPipelineDelay (db−>getPipeDelay());
// disabled masks
Word16 disabledMask = ~(db−>getBackplaneInputMask());
p_backplaneInputmask−>write(disabledMask);
if (mySystemLevel()){
    writertmMask( ~db−>getCableInputMask() );
} else {
    writertmMask(0);
}
// firmware configuration
if (firmwaretype == CmmFirmwareType::CmmCp ) {
    configure_cp () ;
}
else
\\ [...]
m_resync_state =−1;
checkGeneralStatus (1) ;
```

Internally a `m_inrun` is used to flag if the CMX is in the run. This is set at `prepareForRun()` and `resume()`, and removed at `pause()` and `stopFrontEnd()`.

2.1.3 CMX database value

The following variables and values are in the L1Calo database:

A common column to many database entries is the `ChannelId` which helps to identify database values for a particular CMX. The format is a hex number with `0xABCDEFG` where `A` is the crate number (starting from 8 for CP crates, 12 for JEP crates), `B` is the module type (`8` is for CMX), `C` indicates left or right position. For the fine delay `E` is the channel number and `G` is the pin number. Figures 5– 9 show some screenshots of the L1Calo database structure.

0:/TRIGGER/L1Calo/V1/Calibration/CmxCalib

Until	ModuleId	ErrorCode	CmxDeskew	CmxDeskew	ttcrxPhase1	ttcrxPhase2	pipeDelay
ValidityKe…	0	0	0	0	30	0	0
ValidityKe…	0	0	0	0	50	0	0
ValidityKe…	0	0	0	0	30	0	0
ValidityKe…	0	0	0	0	60	0	0
ValidityKe…	0	0	0	0	20	0	0
ValidityKe…	0	0	0	0	60	0	0
ValidityKe…	0	0	0	0	30	175	33
ValidityKe…	0	0	0	0	70	215	33
ValidityKe…	0	0	0	0	180	0	0
ValidityKe…	0	0	0	0	120	0	0
ValidityKe…	0	0	0	0	180	85	49
ValidityKe…	0	0	0	0	140	45	33

Figure 5: L1Calo database content related to the CMX, here the CMX clock phases.

1:/TRIGGER/L1Calo/V1/Calibration/CmxInputDelayCalib

ChannelI	Since	Until	ModuleId	ErrorCode	CmxInputTim	signal00	signal01
8A00100	2014-09-1…	ValidityKe…	0	0	0	0	0
8A00101	2014-09-1…	ValidityKe…	0	0	0	0	0
8A00102	2014-09-1…	ValidityKe…	0	0	0	0	0
8A00103	2014-09-1…	ValidityKe…	0	0	0	0	0
8A00104	2014-09-1…	ValidityKe…	0	0	0	0	0

Figure 6: L1Calo database content related to the CMX, here the backplane delays.

2:/TRIGGER/L1Calo/V2/Configuration/ReadoutConfig

ChannelI	Since	Until	description	baselinePoin	numFadcSlic	l1aFadcSlice	numLutSlices	l1aLutSlice
1	2015-10-1…	ValidityKe…	Default	63	5	2	1	0
2	2015-09-2…	ValidityKe…	Extended	63	15	7	1	0
3	2015-09-2…	ValidityKe…	Reduced	63	5	2	1	0
4	2015-10-1…	ValidityKe…	Expert	63	7	3	1	0
5	2015-09-2…	ValidityKe…	Def80MHz	63	5	2	1	0
6	2015-09-2…	ValidityKe…	Ext80MHz	63	15	7	1	0

Figure 7: L1Calo database content related to the CMX, here the readout configuration.

2:/TRIGGER/L1Calo/V2/Configuration/ReadoutConfig

latencyCpCn	latencyJetCn	latencyJetCn	latencyJetCn	latencyJetCn	latencyJetCn	latencyJetCn	latencyEnerg	latencyEnerg
33	28	29	32	33	28	27	24	25
33	28	29	32	33	28	27	24	25
33	28	29	32	33	28	27	24	25
33	28	29	32	33	28	27	24	25
33	28	29	32	33	28	27	24	25
33	28	29	32	33	28	27	24	25

UInt32

Figure 8: L1Calo database content related to the CMX, here another view on the readout configuration.

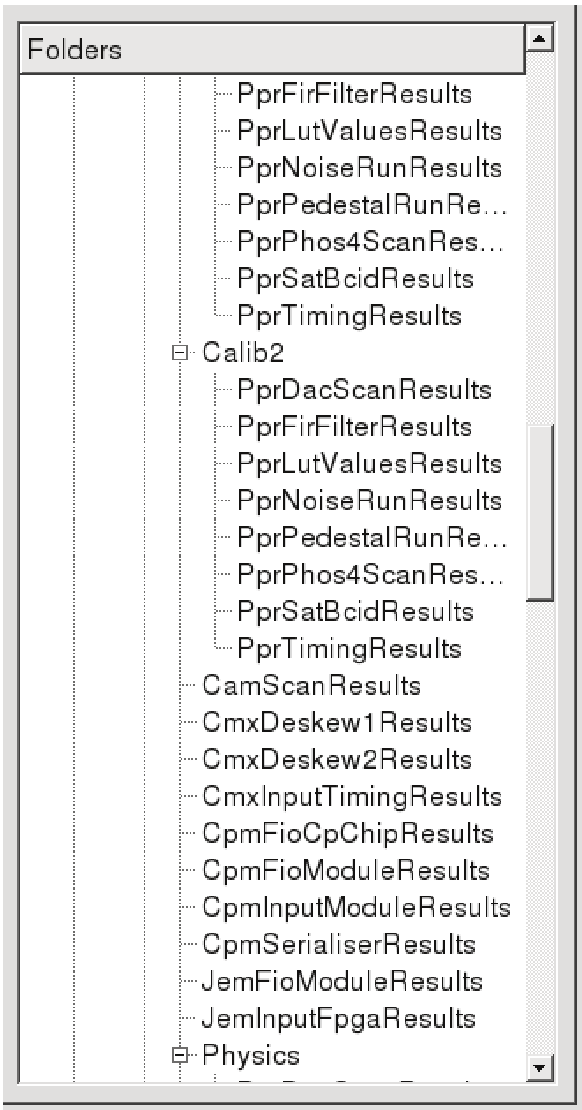


Figure 9: L1Calo database folders related to the CMX.

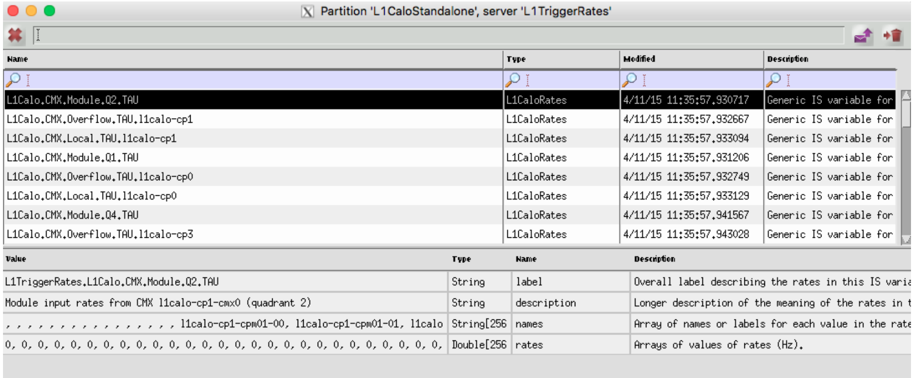


Figure 10: Screenshot of the IS for the CMX rate metering.

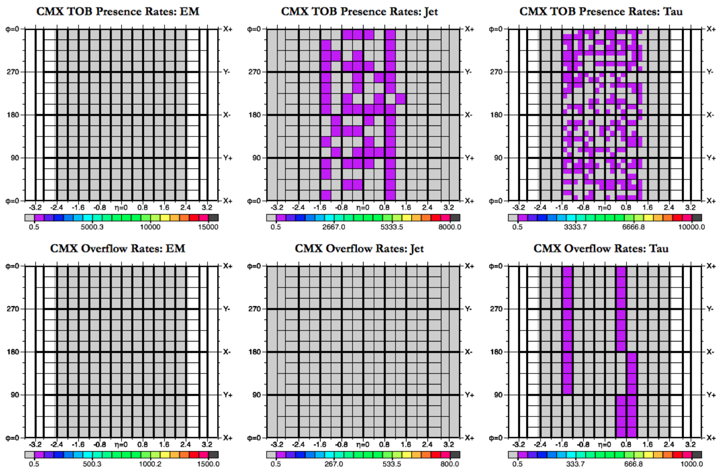


Figure 11: Screenshot of the L1Calo on-call page for the CMX rate metering, graphical representation of the TOB locations.

2.1.4 CMX rate metering

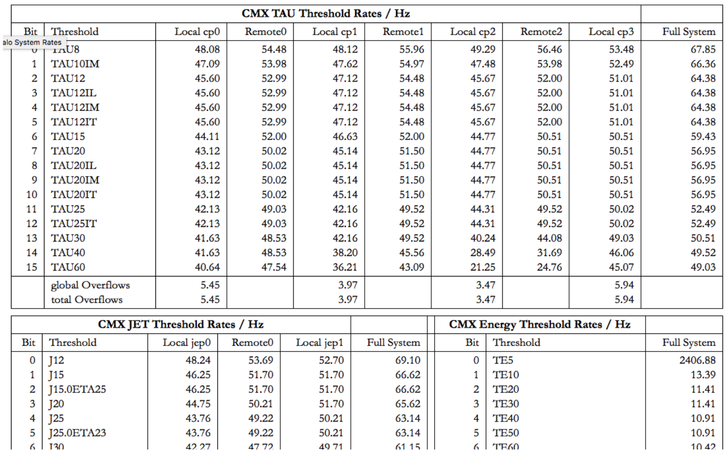
Regularly the rates of the CMX rate monitoring are published to the information system (IS) in the `updateFullStatistics()` function. The rates are calculated for all thresholds and TOBs simultaneously by holding the counters (`p_rateCounterInhibit`) while reading the counters and calculating the rates (see Table 2). Rates are availble for each each presence bit, local, remote and total threshold counters and overflows. In case the normalisation counter (`p_normalisationCounter`) is zero, no rates are published and a warning is given in ERS.

The rates can be found in IS under `L1TriggerRates` with the value names `L1Calo.CMX.*X*.*Y*` where `*X*` can be `Modul` for the module rates, `Overflow` for overflow rate and `Local` for local rates. `*X*` designate the quadrant and type (TAU, EM, JET, ENERGY) of rate. A screenshot can be seen in Figure 10. The rates in numbers and as a map of L1Calo can also be found on the L1Calo on-call page [19] as seen in Figures 11 and 12.

2.1.5 CMX status panel

In regular intervals also the status of the CMX is reported in the L1Calo status panel (see Figure 13).

The function `updateModuleStatus()` updates the status of the backplane and the cable inputs in the panel by reading the channels masks from the registers (`p_backplaneInputmask` and `p_rtmMask`).



		CMX TAU Threshold Rates / Hz							
Bit	Threshold	Local cp0	Remote0	Local cp1	Remote1	Local cp2	Remote2	Local cp3	Full System
0	TAU8	48.08	54.48	48.12	55.96	49.29	56.46	53.48	67.85
1	TAU10IM	47.09	53.98	47.62	54.97	47.48	53.98	52.49	66.36
2	TAU12	45.60	52.99	47.12	54.48	45.67	52.00	51.01	64.38
3	TAU12IL	45.60	52.99	47.12	54.48	45.67	52.00	51.01	64.38
4	TAU12IM	45.60	52.99	47.12	54.48	45.67	52.00	51.01	64.38
5	TAU12IT	45.60	52.99	47.12	54.48	45.67	52.00	51.01	64.38
6	TAU15	44.11	52.00	46.63	52.00	44.77	50.51	50.51	59.43
7	TAU20	43.12	50.02	45.14	51.50	44.77	50.51	50.51	56.95
8	TAU20IL	43.12	50.02	45.14	51.50	44.77	50.51	50.51	56.95
9	TAU20IM	43.12	50.02	45.14	51.50	44.77	50.51	50.51	56.95
10	TAU20IT	43.12	50.02	45.14	51.50	44.77	50.51	50.51	56.95
11	TAU25	42.13	49.03	42.16	49.52	44.31	49.52	50.02	52.49
12	TAU25IT	42.13	49.03	42.16	49.52	44.31	49.52	50.02	52.49
13	TAU30	41.63	48.53	42.16	49.52	40.24	44.08	49.03	50.51
14	TAU40	41.63	48.53	38.20	45.56	28.49	31.69	46.06	49.52
15	TAU60	40.64	47.54	36.21	43.09	21.25	24.76	45.07	49.03
	global Overflows	5.45		3.97		3.47		5.94	
	total Overflows	5.45		3.97		3.47		5.94	

		CMX JET Threshold Rates / Hz			
Bit	Threshold	Local jep0	Remote0	Local jep1	Full System
0	J12	48.24	53.69	52.70	69.10
1	J15	46.25	51.70	51.70	66.62
2	J15.0ETA25	46.25	51.70	51.70	66.62
3	J20	44.75	50.21	51.70	65.62
4	J25	43.76	49.22	50.21	63.14
5	J25.0ETA23	43.76	49.22	50.21	63.14
6	J30	42.27	47.72	49.71	61.15

	CMX Energy Threshold Rates / Hz	
Bit	Threshold	Full System
0	TE5	2406.88
1	TE10	13.39
2	TE20	11.41
3	TE30	11.41
4	TE40	10.91
5	TE50	10.91
6	TE60	10.42

Figure 12: Screenshot of the L1Calo on-call page for the CMX rate metering.

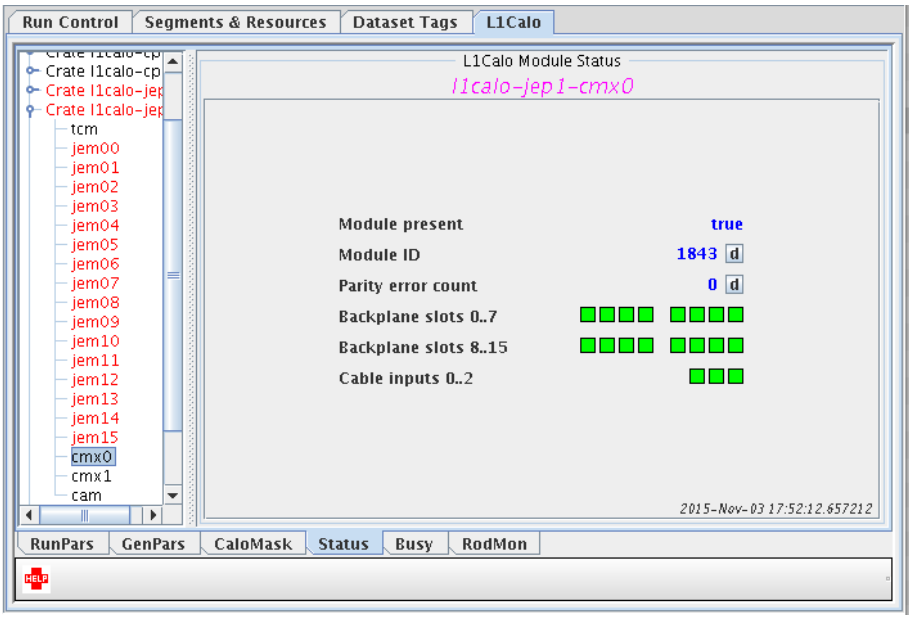


Figure 13: L1Calo status panel.

Channels that are not applicable (RTM inputs for crate-type CMXs, backplane channels 0 and 15 for CP CMXs) either show a good status or are greyed out. The error mask for a backplane channel is set if the clock counter for a particular channel (`p_clockDetectCounter`) is not changing in two consecutive reads. The error mask of a backplane or RTM channel is set, if a parity error has been detected (`p_parityErrorCounter` and `p_inputSpyMemRTMParityErrorCounter`). The status panel for this particular channel will in both cases turn red and a shifter can easily spot the problematic channel. If the parity error counter overflow (32-bit integer overflow), the counters are reset. Messages about parity errors are published to ERS.

The status of the `p_bcResetErrorCounterCounter` and the `p_clockDiffCounter` (duration in clock ticks and ratchet counters of the minimum and maximum clock difference of the clock difference as well: `p_clockDiffDuration`, `p_clockDiffRachetUp` and `p_clockDiffRachetDown`) are checked in the same function to detect a loss of the clock (via the functions `check_bcResetErrorCounter()` and `check_clockDiffCounters()`). If the counters have changed, the time is remembered and only if the difference is still persistent after 10 s and ERS warning is published. In case a `RESYNC` (see below) has been issued and the counters have been reset, the `p_bcResetErrorCounterCounter` and `p_clockDiffCounter` will have returned to zero.

The system monitor of the FPGA is also read in the same function (`check_systemonitor()` via `p_systemonitor`). The calculated voltage and the temperature are printed out to `std::cout` every 60 s.

2.1.6 CMX resync

Signals from the expert system are handled by the `userCommand()` function. One signal `RESYNC` indicates a clock switch between ATLAS internal and LHC clock in steps from 0 to 3. Step 0 is issued before the clock switch, so that the CMX inputs (backplane and RTM) are all disabled. The current disable mask is saved. At step 1 the clock manager is reset and the links to L1Topo are reset as well. Step 2 is issued approximately 5 s after step 1, so that sending of alignment characters to L1Topo is forced for at least 5 s and L1Topo can reset its link at this step. The inputs are enabled again at step 3 using the previously saved masks. The `p_bcResetErrorCounterCounter` and the `p_clockDiffCounter` are shown together with an INFO message on ERS about the reset and the waiting period.

Internally the resync step is stored in `m_resync_state`, so that executing a step twice does not have any effect.

2.1.7 Configuration of CMX thresholds

The thresholds are set according to the flavour of the CMX. There are functions that directly configure the thresholds (`configure_*_old()`) from the database into the registers. The preferred method is now to read the database and store the thresholds in `std::vector` structures (using `get_*_configuration()`). With the functions `configure_*()` the values are written into the registers. There is also the possibility to read the thresholds back from the registers (`read_*_configuration()`). In this case it should be noted that the energy configuration is ambiguous if the scale is not explicitly given. A wrong scale can lead to inconsistent simulation results.

The format of the thresholds is compatible with the format used in the CMX simulation:

- For the jet thresholds the 10-bit thresholds are stored for each η-slice (32 slices) and each of the 25 thresholds (=800 thresholds): `threshold[eta] [thresholdno]`. The thresholds are mapped onto the registers (`p_JetThreshold`), such that the first 25 values are the thresholds for one η slice, etc. For the convenient calculation of the η-slice in the firmware (calculation of the eta slice for JEMs 8-15 which cover the same η-slices in another quadrant), the thresholds are mirrored from the relative address 800 on within the 1600 16-bit memory space. The threshold value already includes the jet size in the highest bit. A sketch of the threshold organisation is shown in Figure 14.

	vector[...][0]	...	vector[...][23]
vector[0]	threshold 0, slice 0	...	threshold 23, slice 0
...	...	...	...
vector[31]	threshold 0, slice 31	...	threshold 23, slice 31

	vector[...][0]		vector[...][23]
vector[0]	0	...	31
...	...	...	...
vector[31]	736	...	799
vector[0]	800	...	831
...	...	...	...
vector[31]	1536	...	1599

Figure 14: Scheme of the jet thresholds in the threshold vector and in the register. The number in the lower table indicate the relative address in the threshold memory.

- Depending on the CMX position (physical or `geoAddr` forced) TAU or EM 8-bit thresholds are loaded. The threshold value already includes the isolation in the highest 5 bits. The thresholds are stored for each η-slice (50 slices) and each of the 16 thresholds: `threshold[eta][thresholdno]`. The thresholds are mapped onto the registers (`p_JetThreshold`), such that the first 16 values are the thresholds for one η slice, etc. Each CPM channel is forced to contain 4 η-slices, even the CPMs at the edge of the η coverage (actually only the inner η slice has detector coverage). So the first three and last three η-slices contain zero thresholds in the CMX registers (so that in total 56×16 words used for 50×16 thresholds, the first and last 48 words are unused). A sketch of the threshold organisation is shown in Figure 15.

- The energy configuration has different registers than the JET/CP thresholds. The first element of the threshold vector contains the $E_{\mathrm{T}}^{\mathrm{miss}}$thresholds (XE: 8 thresholds), followed by the restricted $E_{\mathrm{T}}^{\mathrm{miss}}$thresholds, total energy thresholds (TE: 8 thresholds), restricted total energy thresholds and the $E_{\mathrm{T}}^{\mathrm{miss}}$significance thresholds (XS: 8 thresholds). The 6-th vector contains the scale, the offset, xeMin, xeMax, teSqrtMin, teSqrtMax, restricted $E_{\mathrm{T}}^{\mathrm{miss}}$mask and restricted total energy mask. The (restricted) $E_{\mathrm{T}}^{\mathrm{miss}}$, xeMin and xeMax values are squared and stored as 32-bit words. The xeMin and xeMax values are duplicated for each $E_{\mathrm{T}}^{\mathrm{miss}}$threshold. The (restricted) total energy are stored as 16-bit words, while the teSqrtMin and teSqrtMax (convention in database) values are squared and stored as 16-bit words as well. The teSqrtMin and teSqrtMax are duplicated for each threshold as well. The thresholds for the $E_{\mathrm{T}}^{\mathrm{miss}}$significance thresholds are stored in a special way to ease the thresholding in the firmware. The thresholds from the database are divided by 10, then scaled by 1000, the offset is scaled by 1000 as well. Stored in the registers are the thresholds multiplied by the scale (32-bit word). The offset is squared and stored as a 16-bit value. All numbers are calculated as floats (also to avoid limitations of 32/64-bit integers) and only when written to the registers, they are truncated to integers. For the (detector) restricted thresholds two extra registers are used (`p_etMISSMask` and `p_sumMETMask` for the restricted $E_{\mathrm{T}}^{\mathrm{miss}}$and restricted total energy thresholds respectively). Each bit corresponds to a JEM channel (bit number corresponds to the channel number) and 1 means that this input channel is included in the restricted TE/XE calculation. Due to the coverage of two quadrants in one crate, the mask should repeat after bit 8. An overview is shown in Table 3 and a graphical representation is shown in Figure 16.

	vector[...][0]	...	vector[...][15]
vector[0]	threshold 0, slice 0	...	threshold 15, slice 0
...	...	...	...
vector[49]	threshold 0, slice 49	...	threshold 15, slice 49

	vector[...][0]	...	vector[...][15]
vector[0]	48	...	63
...	...	...	...
vector[49]	780	...	863

Figure 15: Scheme of the CP thresholds in the threshold vector and in the register. The number in the lower table indicate the relative address in the threshold memory.

	vector[...][0]	vector[...][1]		vector[...][7]	
vector[0]	XE0	XE1	...	XE7	sqr -> 32bit words
vector[1]	rXE0	rXE1	...	rXE0	sqr -> 32bit words
vector[2]	TE0	TE1	...	TE7	16bit words
vector[3]	rTE0	rTE1	...	rTE7	16bit words
vector[4]	XS0	XS1	...	XS7	
vector[5]	scale	offset	...		

xeMin	sqr -> 32bit words	
xeMax	sqr -> 32bit words	
teSqrtMin	sqr -> 16bit words	
teSqrtMax	sqr -> 16bit words	
XS/10 * scale /1000		32bit word
offset / 1000		16bit word

Figure 16: Scheme of the met thresholds in the threshold vector and in the register. Also the size of the numbers are indicated.

2.1.8 Functionality and setup/status checks

Using the database, most (ideally all) of the functional behaviour of the CMX are set before the run starts.

Various functions to setup the configuration in the TTCDec card, DS1 and DS2 fine delays, coarse delay, and topo polarity and the backplane input delays:

```
// set correct clock configuration in TTCDec card, set correct data polarity for topo links , enable DS2
void setClocks () ;
void setTopoPolarity () ;
int enableDSkew2();

// set fine and coarse delay on the TTCDec card
void setDeskew1( int ) ;
void setDeskew2( int ) ;
void checkcoarsedelay ( int cdelay ) ;

// set backplane delays
bool setIDELAY( int channel, int databit , int value ) ;
bool setIDELAYclock( int channel, int value ) ;
bool setIDELAY( int channel, int databit ) ;
```

Setting all registers to a default value (not used, yet):

```
void defaultAllRegisters () ;
```

Various checks on status flags, clock settings and firmware versions:

```
// check various lock monitors
int checkLockMonitorDskew1();
int checkLockMonitorDskew2();
int checkLockMonitorPLL();
// check that TTCDec card is ready
int checkTTCready();
// check firmware versions ( all or BSPT only)
void checkBSPTFWversion(bool checkBSPTonly=false);
void checkFWversion( std :: string whichFW, std:: vector < uint32_t > fw_ok, uint32_t fw_version ) ;
// check TTCDec settings
void checkClockSettings () ;
// general functions to BF configuration , TTCDec clock settings , lock monitors
void checkGeneralStatus ( int mode=0 );
// check various PLL lock monitors
void checkPLLsready();
```

Various resets, `moduleResets()` summarizes various module resets:

```
// reset clock mamanger
void resetMCM();
void resetClockMananger();
// reset counters
void resetAllCounters () ;
// reset error counters
void resetErrorCounters () ;
// reset DAQ/ROI links
void moduleResets();
```

2.1.9 Spy memory related objects and functions

The access to the spy memories are handled through the `GenericCMXSpyMemory` class that not only encapsulates the reading/writing of the spy memories, but also handles setting the mode of the spy

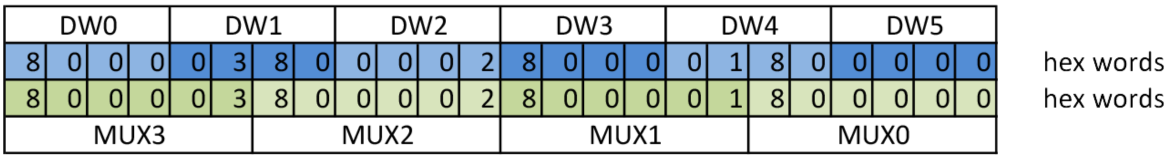


Figure 17: Mapping of the hex words for the backplane data (bottom) and spy memory words (top).

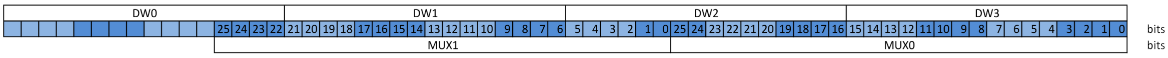


Figure 18: Mapping of the data bits for the RTM data (bottom) and spy memory words (top).

memories, error checking registers, parity error registers, reset of error counters (see Table 4). The class can also return a `SpyMemoryEvents` object with the correct settings of the number of channels, number of events and number of spy memory words per event (see also Table 5). Also the number of bits that form an actual data word are stored in the returned object. Likewise an already existing `SpyMemoryEvents` object can be modified, so that it can store the content of a particular spy memory. Some resets are shared among the spy memories, like all input spy memories use the same pointer to the register that resets the error counts.

For certain CMX flavours some spy memories do not exist in firmware and when `cmxServices` instantiates the objects, the `isActive()` function returns false.

The control and status registers for the CP and non-CP spy memories are in fact the same registers, but the objects internally have the information about the number of channels, so that when the spy memory is read/written the correct number of events are read/written.

The spy memory words are 16-bit words according to the size of the register words and they are read out one by one from the CMX data register (the reading of the register automatically advances to the next spy memory word). The order of the spy memory words (as read back from the CMX data register) is ordered in time (0-th word is part of MUX0), but they are stored in reverse order in the `SpyMemoryEvents` objects. This means that the LSW is read back first from the spy memory, which is also is transmitted/recevied first. As the data words in the `SpyMemoryEvents` object are printed from left to right, it was deceided to reverse the order in the objects, so that the LSW appears on the right side.

A mapping from the data format words and channels is needed: For the backplane words, each MUX has 24bit and the total of 4 MUXes are mapped onto 6 spy memory words (see Figure 17). Each channel corresponds to one input channel. For the JET/ENERGY RTM output spy memories each channel corresponds to each cable, so that those memories have two channels. Both MUXes (with 26bits of payload) are mapped into 4 spy memory words (see Figure 18). Likewise the RTM input spy memories are mapped. For the CP RTM output spy memories there is only one cable, hence only one channel. The CP input RTM spy memory has three channels corresponding to the three CP crate-type CMXs. Care has to be taken when the RTM output spy memories from each CP crate-type CMX is fed into the CP system-type CMX input RTM spy memory due to the different number of channels. The CTP spy memories have only one channel (despite the fact, that there are two cables), both 31bit data words are mapped onto 4 spy memories words (see Figure 19).

For all spy memories an inhibit flag (`p_spyMemoryInhibit`) can be set, so that the CMX stops writing to the spy memories and a consistent state of all spy memories can be obtained (via `setSpyMemoryInhibit()`).

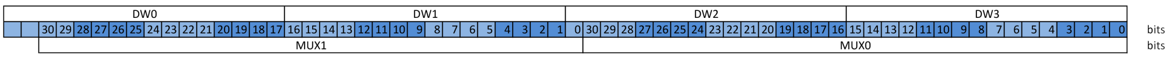


Figure 19: Mapping of the data bits for the CTP data (bottom) and spy memory words (top).

The GenericCMXSpyMemory class encapsulates all status and control register pointers as well as information about the size and data formats that underlie a certain spy memory. A spy memory requires a name, a status register, a control register, a register that contains the memory word and a register that holds the start address (which will only be effective at a TTCBroadcast command. There can be registers that reset the error counters (counterReset()), registers that hold the error bits when the memory is in verify mode (getErrorFlags()). The size of the error register is also defined. Register that store the error counters can also be defined (getParityError(), getNoErrorCounter()). A function (setDPRmode()) and constants (DPR_CONTROL_*) are defined to set the mode of the spy memory. To facilitate the readability of the spy memory words, a rearrangement of the words can be defined (the default is to reverse the order of the spy memory words). The size of the spy memory in terms of channels, spy memory words per event and the number of events can also be stored. A particular GenericCMXSpyMemory can also return a SpyMemoryEvent object (createSpyMemoryEvents()) that inherits the dimensions of the spy memory. Reading and writing of spy memories (readSpyMemory(), writeSpyMemory()) is done through SpyMemoryEvent objects.

Here is an excerpt of the class definition of the GenericCMXSpyMemory class:

```
class GenericCMXSpyMemory {
 public:
  // headers to create with or without pointers to the registers
  GenericCMXSpyMemory( std::string name, CMX_spyMemStatus * MemStatus, CMX_spyMemControl * MemControl,
      ModuleMemory16 * MemWord, ModuleMemory16 * StartAddress, ModuleRegister16 * CounterReset,
      ModuleMemory16 * ErrorCheck, unsigned int errorsize, ModuleMemory16 * NoErrorCounter, ModuleMemory16 *
      ParityErrorCounter, std::vector<int> memorder, size_t channels, size_t words, size_t events=256, int format=-1, bool
      quick=true, bool verbose=false );
  GenericCMXSpyMemory( std::string name, CMX_spyMemStatus * MemStatus, CMX_spyMemControl * MemControl,
      ModuleMemory16 * MemWord, ModuleRegister16 * StartAddress, ModuleRegister16 * CounterReset,
      ModuleMemory16 * ErrorCheck, unsigned int errorsize, ModuleMemory16 * NoErrorCounter, ModuleMemory16 *
      ParityErrorCounter, std::vector<int> memorder, size_t channels, size_t words, size_t events=256, int format=-1, bool
       quick=true, bool verbose=false );
  GenericCMXSpyMemory( std::string name, size_t channels, size_t words, size_t events, int format=-1, bool quick
      =true, bool verbose=false );
  // naming of the spy memory, resets, reading, writing from the spy memory
  std :: string name();
  void counterReset () ;
  int readSpyMemory( Cmxdataformats::SpyMemoryEvents &memory );
  int writeSpyMemory( Cmxdataformats::SpyMemoryEvents &memory );
  int SpyMemoryReset();
  // setting / getting start addresses
  void setStartAddress ( int start , int channel=0 );
  uint32_t getStartAddress ( int channel=0 );
  // reading error related flags
  std :: vector < uint32_t > getErrorFlags () ;
  // [...]
  uint32_t getParityError ( unsigned int channel=0 );
  uint32_t getNoErrorCounter( unsigned int channel=0 );
  // functions for getting / setting and constants for states of the spy memory or the spy memory data buffer
  int setDPRmode( unsigned int mode );
  uint32_t getDPRmode();
  const static int DPR_CONTROL_SPY=1;
  const static int DPR_CONTROL_PLAYBACK=2;
  const static int DPR_CONTROL_VERIFY=3;
  const static int DPR_STATUS_NORMAL=1;
  const static int DPR_STATUS_WAIT_INHIBIT=2;
  const static int DPR_STATUS_WAIT_READ=3;
  const static int DPR_STATUS_WAIT_WRITE=4;
  const static int DPR_STATUS_WRITE=5;
  const static int DPR_STATUS_READ=6;
```

```
  // active state
  void setActive( bool active=true );
  bool isActive();
  // handeling SpyMemoryEvent objects
  void expandSpyMemoryEvents( Cmxdataformats::SpyMemoryEvents &spymemevent );
  Cmxdataformats::SpyMemoryEvents createSpyMemoryEvents();
}
```

2.2 CMX registers

The tables below list the HDMC names of the registers and which pointer and which constant in the VHDL code they correspond to. The HDMC parts file is located in `cmxServices/hdmc/parts /L1CaloCmx.parts`. No special dependencies or substrcture has been defined. The pointers and registers defined in the VHDL code can be related by their byte addresses (VHDL) and word addresses (HDMC). Some pointers are of an user defined class with subfunctions that are defined by the `cmxServices/hdmc/conf/cmx.bits` file and which access certain bit fields in the register via automatically generated functions (this has not much used for "later" registers). These are listed in a separate list below:

```
CMX_moduleIDSN      * p_moduleIDSN;
CMX_moduleRew       * p_moduleRew;
CMX_moduleControl   * p_moduleControl;
CMX_moduleResets    * p_moduleResets;
CMX_moduleStatus1   * p_moduleStatus1;
CMX_moduleStatus2   * p_moduleStatus2;
CMX_linkStatus1     * p_linkStatus1 ;
CMX_linkStatus2     * p_linkStatus2 ;
CMX_sfpControlStatus * p_sfp1ControlStatus ;
CMX_sfpData         * p_sfp1Data;
CMX_sfpControlStatus * p_sfp2ControlStatus ;
CMX_sfpData         * p_sfp2Data;
CMX_sfpControlStatus * p_sfp3ControlStatus ;
CMX_sfpData         * p_sfp3Data;
CMX_sfpControlStatus * p_sfp4ControlStatus ;
CMX_sfpData         * p_sfp4Data;
CMX_mpControlStatus * p_mp12ControlStatus;
CMX_mpData          * p_mp12Data;
CMX_mpControlStatus * p_mp345ControlStatus;
CMX_mpData          * p_mp345Data;
CMX_LEDsControl1    * p_LEDsControl1;
CMX_LEDsRequests    * p_LEDsRequests;
CMX_BFTPDebug       * p_BFTPDebug;
CMX_BSPTDebug       * p_BSPTDebug;
CMX_TTCrxControl    * p_TTCrxControl;
CMX_TTCrxStatus     * p_TTCrxStatus;
CMX_TTCrxBrcst      * p_TTCrxBrcst;
CMX_TTCrxDq         * p_TTCrxDq;
CMX_TTCrxDump       * p_TTCrxDump;
CMX_spyMemControl   * p_inputSpyMemSourceControl;
CMX_spyMemStatus    * p_inputSpyMemSourceStatus;
CMX_spyMemControl   * p_inputSpyMemSystemControl;
CMX_spyMemStatus    * p_inputSpyMemSystemStatus;
CMX_spyMemControl   * p_inputSpyMemCTPControl;
CMX_spyMemStatus    * p_inputSpyMemCTPStatus;

CMX_spyMemControl   * p_inputSpyMemRTMSourceControl;
CMX_spyMemStatus    * p_inputSpyMemRTMSourceStatus;

```

```
CMX_spyMemControl    * p_inputSpyMemRTMSystemDS2Control;
CMX_spyMemStatus     * p_inputSpyMemRTMSystemDS2Status;

CMX_spyMemControl    * p_inputSpyMemRTMSystemControl;
CMX_spyMemStatus     * p_inputSpyMemRTMSystemStatus;

CMX_spyMemControl    * p_inputSpyMemRTMOutputControl;
CMX_spyMemStatus     * p_inputSpyMemRTMOutputStatus;
CMX_quietForce       * p_quietForce ;
CMX_daqROIstatus     * p_daqROIstatus;
```

database name	function	value
V1/CmxCalib/ttcrxPhase1	getTtcrxPhase1	DS1 phase
../ttcrxPhase2	getTtcrxPhase2	DS2 phase
../pipeDelay	getPipeDelay	pipe delay for system-type CMX, this value is actually the value stored in the register (1 in bit 0 for activation of delay, bits 4-7 delay minus 1)
../CmxInputDelayCalib	std::vector getBackplaneInputDelays(line)	fine delays for a particular channel, line==24 is for the delay of the forwarded clock line.
V2/ReadoutConfig/ baselinePointer	not directly used	baseline pointer for all L1Calo, added automatically to all offset values
.../numProcSlice	getNumDaqSlices	number of DAQ slices
.../l1aProcSlice	not used	central L1A DAQ slice, not applicable for CMX, since the L1A slice is always the central one
.../latency*X*Cmx*Y*	getDaqOffsetForPin, getDaqOffsetBpData	absolute latency for the glink pins for the DAQ readout, *X* can be Cp, Jet or Energy, *Y* can be Backplane (offset for DAQ readout), Info (for last glink pin), System (Readout: total result), Local (Local result), Cable (Remote result). For the CMX configuration the relative latency with respect to the backplane latency is used.
.../bcOffsetCMX	getBcOffset	BC offset value to which the BC counter should be set upon a BC reset signal
.../formatTypeCpJep	for ROD configuration	readout format, not applicable for CMX configuration
V2/CmxDeskew1Results	in preparation	result folder for DS1 scans
../CmxDeskew2Results	in preparation	result folder for DS2 scans
../InputTimingResults	in preparation	result folder for fine delay scans

Table 1: Overview of the database entries related to the CMX.

register pointer	rate metering
p_rateCounterInhibit	inhibit rate counters
p_rateCounterReset	reset all counters
p_rateCounterNormalisation	normalisation counter, increased at each clock tick
p_localRates	local threshold rates (local)
p_remoteRates	remote threshold rates (from RTM)
p_totalRates	total threshold rates (to CTP)
p_tobCounter	TOB counters for each presence bit
p_localBackplaneOverflow	local overflow
p_globalBackplaneOverflow	global overflow
p_totalBackplaneOverflow	total overflow
p_sumETCounter	TE, counters
p_METCounter	XE, counters
p_METSignCounter	XS, counters
p_sumETWeightedCounter	rTE counters
p_METWeightedCounter	rXE counters

Table 2: Overview of the rate counter register pointers.

register pointer	MET thresholds
p_sumMETMask	TE restricted mask 8-bit mask
p_etMISSMask	XE restricted mask 8-bit mask
p_etMissThreshold	XE thresholds, squared, 32-bit words
p_etMissResThreshold	rXE thresholds, squared, 32-bit words
p_sumEtThreshold	TE thresholds, 16-bit words
p_sumEtResThreshold	rTE thresholds, 16-bit words
p_xsThreshold	XS thresholds times scale, squared, 32-bits
p_etMissMinParam	xeMin for each threshold, squared, 32-bits
p_etMissMaxParam	xeMax for each threshold, squared, 32-bits
p_sumEtMinParam	teMin for each threshold, squared teSqrMin, 16-bits
p_sumEtMaxParam	teMax for each threshold, squared teSqrMax, 16-bits
p_xsParam	offset squared for each threshold, 16-bits

Table 3: Overview of the MET threshold register pointers.

register pointer	spy memory function
p_parityErrorCounter	parity error counter at the backplane input
p_counterReset	resets all error counters at the backplane input (source and system)
p_inputSpyMemSourceWord	spy memory words
p_inputSpyMemSourceControl	control register
p_inputSpyMemSourceStatus	status register
p_inputSpyMemSourceStartAddress	start address
p_inputSpyMemSourceCheckError	error latches when spy memory is in verify mode
p_inputSpyMemSourceNoerrorCounter	counters for number of clock ticks without any errors
p_inputSpyMemSystemWord	spy memory words
p_inputSpyMemSystemControl	control register
p_inputSpyMemSystemStatus	status register
p_inputSpyMemSystemStartAddress	start address
p_inputSpyMemSystemCheckError	error latches when spy memory is in verify
p_inputSpyMemSystemNoerrorCounter	number of CLK ticks without any errors
p_ctpCounterReset	reset all error counters at the CTP output
p_inputSpyMemCTPWord	spy memory words
p_inputSpyMemCTPControl	control register
p_inputSpyMemCTPStatus	status register
p_inputSpyMemCTPStartAddress	start address
p_inputSpyMemCTPCheckError	error latches when spy memory is in verify
p_inputSpyMemCTPNoerrorCounter	number of CLK ticks without any errors
p_inputSpyMemRTMParityErrorCounter	parity error counter at RTM input
p_rtmCounterReset	resets all error counters at the RTM input (source, system DS2 and system DS1)
p_inputSpyMemRTMSourceWord	spy memory words
p_inputSpyMemRTMSourceControl	control register
p_inputSpyMemRTMSourceStatus	status register
p_inputSpyMemRTMSourceStartAddress	start address
p_inputSpyMemRTMSourceCheckError	error latches when spy memory is in verify
p_inputSpyMemRTMSystemDS2Word	spy memory words
p_inputSpyMemRTMSystemDS2Control	control register
p_inputSpyMemRTMSystemDS2Status	status register
p_inputSpyMemRTMSystemDS2StartAddress	start address
p_inputSpyMemRTMSystemDS2CheckError	error latches when spy memory is in verify
p_inputSpyMemRTMSystemWord	spy memory words
p_inputSpyMemRTMSystemControl	control register
p_inputSpyMemRTMSystemStatus	status register
p_inputSpyMemRTMSystemStartAddress	start address
p_inputSpyMemRTMSystemCheckError	error latches when spy memory is in verify

Table 4: Overview of the register pointers related to the spy memories.

Spy memory	CMX flavour	words	channels	bits per data word
`sourceSpyMemory`	all	6	16	24
`systemSpyMemory`	all	6	16	24
`rtmOutputSpyMemory`	crate-type, JET/en	4	2	26
`rtmSourceSpyMemory`	system-type, JET/en	4	2	26
`rtmSystemSpyMemory`	system-type, JET/en	4	2	26
`rtmSystemDS2SpyMemory`	system-type, JET/en	4	2	26
`rtmOutputCPSpyMemory`	crate-type, CP	4	1	26
`rtmSourceCPSpyMemory`	system-type, CP	4	3	26
`rtmSystemCPSpyMemory`	system-type, CP	4	3	26
`rtmSystemCPDS2SpyMemory`	system-type, CP	4	3	26
`ctpSpyMemory`	system-type	4	1	31

Table 5: Overview of all CMX spy memories. Some spy memories do not exist for certain flavours. The number of words and channels are the number of spy memory words and channels. The mapping to the data format words is explained in the text. The number of bits per data word are the number of bits used per data word (per channel) in the data format, which are then mapped onto 16-bit words.

HDMC name and class	pointer name	VHD constant	byte ADDR	word ADDR
MR::moduleIDSN	p_moduleIDSN	ModuleIDSN	0000	0000
MR::moduleRew	p_moduleRew	ModuleRew	0001	0002
MR::moduleControl	p_moduleControl	ModuleControl	0002	0004
MR::moduleResets	p_moduleResets	ModuleResets	0003	0006
MR::moduleStatus1	p_moduleStatus1	ModuleStatus1	0004	0008
MR::moduleStatus2	p_moduleStatus2	ModuleStatu2	0005	000a
MR::linkStatus1	p_linkStatus1	LinkStatus1	0006	000c
MR::linkStatus2	p_linkStatus2	LinkStatus2	0007	000e
MR::sfp1ControlStatus	p_sfp1ControlStatus	SFP1_CSR	0008	0010
MR::sfp1Data	p_sfp1Data	SFP1_Data	0009	0012
MR::sfp2ControlStatus	p_sfp2ControlStatus	SFP2_CSR	000A	0014
MR::sfp2Data	p_sfp2Data	SFP2_Data	000B	0016
MR::sfp3ControlStatus	p_sfp3ControlStatus	SFP3_CSR	000C	0018
MR::sfp3Data	p_sfp3Data	SFP3_Data	000D	001a
MR::sfp4ControlStatus	p_sfp4ControlStatus	SFP4_CSR	000E	001c
MR::sfp4Data	p_sfp4Data	SFP4_Data	000F	001e
MR::mp12ControlStatus	p_mp12ControlStatus	MP12_CSR	0010	0020
MR::mp12Data	p_mp12Data	MP12_Data	0011	0022
MR::mp345ControlStatus	p_mp345ControlStatus	MP345_CSR	0012	0024
MR::mp345Data	p_mp345Data	MP345_Data	0013	0026
MR::LEDsControl1	p_LEDsControl1	LEDsControl1	0014	0028
MR::LEDsRequests	p_LEDsRequests	LEDsRequests	0015	002a
MR::BFTPDebug	p_BFTPDebug	BFTPDebug	0016	002c
MR::BSPTDebug	p_BSPTDebug	BSPTDebug	0017	002e
MR::TTCrxControl	p_TTCrxControl	TTCrxControl	0018	0030
MR::TTCrxStatus	p_TTCrxStatus	TTCrxStatus	0019	0032
MR::TTCrxBrcst	p_TTCrxBrcst	TTCrxBrcst	0020	0040
MR::TTCrxDq	p_TTCrxDq	TTCrxDq	0021	0042
MR::TTCrxDump	p_TTCrxDump	TTCrxDump	0022	0044
MM::SystemACE	p_SystemACE	SystemACE[1]	0040 - 00DF	0080 - 01be
MR::testRO	p_testRO	[1]	0080	0100
MR::testRW	p_testRW	[1]	0081	0102

HDMC name and class	pointer name	VHD constant	byte ADDR	word ADDR
MR::backplaneForward	p_backplaneForward	*_RO_backplane_forward	0082	0104
MM::idelayBackplane	p_idelayBackplane	*_RW_IDELAY_BACKPLANE	00A2 - 0231	0144 - 0462
MM::evCounter	p_evCounter	*_RO_EV_COUNTER	0232 - 0233	0464 - 0466
MM::parityErrorCounter	p_parityErrorCounter	*_RO_PARITY_ERROR_COUNTER	0234 - 0253	0468 - 04a6
MR::counterReset	p_counterReset	*_RW_COUNTER_RESET	0254	04a8
MM::inputSpyMemSourceWord	p_inputSpyMemSourceWord	*_RW_INPUT_SPY_MEM_SOURCE_WORD	0255 - 025A	04aa - 04b4
MR::inputSpyMemSourceControl	p_inputSpyMemSourceControl	*_RW_INPUT_SPY_MEM_SOURCE_CONTROL	025b	04b6
MR::inputSpyMemSourceStatus	p_inputSpyMemSourceStatus	*_RO_INPUT_SPY_MEM_SOURCE_STATUS	025c	04b8
MM::inputSpyMemSourceStartAddress	p_inputSpyMemSourceStartAddress	*_RW_INPUT_SPY_MEM_SOURCE_START_ADDRESS	025D - 026C	04ba - 04d8
MM::inputSpyMemSourceCheckError	p_inputSpyMemSourceCheckError	*_RO_INPUT_SPY_MEM_SOURCE_CHECK_ERROR	026D - 028C	04da - 0518
MM::inputSpyMemSourceNoerrorCounter	p_inputSpyMemSourceNoerrorCounter	*_RO_INPUT_SPY_MEM_SOURCE_NOERROR_COUNTER	028D - 02AC	051a - 0558
MR::ClockManagerReset	p_ClockManagerReset	*_RW_CLOCK_MANAGER_RESET	02AD	055a
MR::iDelayCtrlRDY	p_iDelayCtrlRDY	*_RO_IDELAYCTRL_RDY	02AE	055c
MR::iDelayCtrlRST	p_iDelayCtrlRST	*_RO_IDELAYCTRL_RST	02AF	055e
MR::iDelayCtrlwasRST	p_iDelayCtrlwasRST	*_RO_IDELAYCTRL_WAS_RST	02B0	0560
MR::inputModReset	p_inputModReset	*_RW_INPUT_MOD_RESET	02B1	0562
MR::inputModCounterEnable	p_inputModCounterEnable	*_RO_INPUT_MOD_COUNTER_ENABLE	02B2	0564
MR::ctpTesterDataSelect	p_ctpTesterDataSelect	*_RW_CTP_TESTER_DATA_SELECT	02B3	0566
MR::topoTRGTXreset	p_topoTRGTXreset	*_RW_TOPOTR_GTX_RESET	02B4	0568
MR::rxPolarityA	p_rxPolarityA	*_RW_RX_POLARITY	02B5	056a
MR::rxPolarityB	p_rxPolarityB	*_RW_RX_POLARITY	02B6	056c
MR::rxPolarityC	p_rxPolarityC	*_RW_RX_POLARITY	02B7	056e
MR::txPolarityA	p_txPolarityA	*_RW_TX_POLARITY	02B8	0570
MR::txPolarityB	p_txPolarityB	*_RW_TX_POLARITY	02B9	0572
MR::txPolarityC	p_txPolarityC	*_RW_TX_POLARITY	02BA	0574
MM::JetThreshold	p_JetThreshold	*_RW_JET_THRESHOLD_BLOCK	02BB - 08FA	0576 - 11f4
MR::DAQSlice	p_DAQSlice	*_RW_DAQ_SLICE	08FB	11f6
MR::DAQRAMOffset	p_DAQRAMOffset	*_RW_DAQ_RAM_OFFSET	08FC	11f8
MR::BCIDResetVal	p_BCIDResetVal	*_RW_BCID_RESET_VAL	08FD	11fa
MR::DAQROIreset	p_DAQROIreset	*_RW_DAQ_ROI_RESET	08FE	11fc
MM::inputSpyMemSystemWord	p_inputSpyMemSystemWord	*_RW_INPUT_SPY_MEM_SYSTEM_WORD	08FF - 0904	11fe - 1208
MR::inputSpyMemSystemControl	p_inputSpyMemSystemControl	*_RW_INPUT_SPY_MEM_SYSTEM_CONTROL	0905	120a
MR::inputSpyMemSystemStatus	p_inputSpyMemSystemStatus	*_RO_INPUT_SPY_MEM_SYSTEM_STATUS	0906	120c
MR::inputSpyMemSystemStartAddress	p_inputSpyMemSystemStartAddress	*_RW_INPUT_SPY_MEM_SYSTEM_START_ADDRESS	0907	120e
MM::inputSpyMemSystemCheckError	p_inputSpyMemSystemCheckError	*_RO_INPUT_SPY_MEM_SYSTEM_CHECK_ERROR	0908 - 0927	1210 - 124e
MM::inputSpyMemSystemNoerrorCounter	p_inputSpyMemSystemNoerrorCounter	*_RO_INPUT_SPY_MEM_SYSTEM_NOERROR_COUNTER	0928 - 0947	1250 - 128e
MR::ctpCounterReset	p_ctpCounterReset	*_RW_CTP_OUTPUT_COUNTER_RESET	0949	1292
MM::inputSpyMemCTPWord	p_inputSpyMemCTPWord	*_RW_CTP_SPY_MEM_WORD	094A - 094D	1294 - 129a
MR::inputSpyMemCTPControl	p_inputSpyMemCTPControl	*_RW_CTP_SPY_MEM_CONTROL	094E	129c
MR::inputSpyMemCTPStatus	p_inputSpyMemCTPStatus	*_RO_CTP_SPY_MEM_STATUS	094F	129e
MR::inputSpyMemCTPStartAddress	p_inputSpyMemCTPStartAddress	*_RW_CTP_SPY_MEM_START_ADDRESS	0950	12a0
MM::inputSpyMemCTPCheckError	p_inputSpyMemCTPCheckError	*_RO_CTP_SPY_MEM_CHECK_ERROR	0951 - 0954	12a2 - 12a8
MM::inputSpyMemCTPNoerrorCounter	p_inputSpyMemCTPNoerrorCounter	*_RO_CTP_SPY_MEM_NOERROR_COUNTER	0955 - 0956	12aa - 12ac
MR::rtmCounterReset	p_rtmCounterReset	*_RW_RTM_INPUT_COUNTER_RESET	0957	12ae
MM::inputSpyMemRTMSourceWord	p_inputSpyMemRTMSourceWord	*_RW_RTM_SPY_SOURCE_MEM_WORD	0958 - 095B	12b0 - 12b6

HDMC name and class	pointer name	VHD constant	byte ADDR	word ADDR
MR::inputSpyMemRTMSourceControl	p_inputSpyMemRTMSourceControl	*_RW_RTM_SPY_SOURCE_MEM_CONTROL	095C	12b8
MR::inputSpyMemRTMSourceStatus	p_inputSpyMemRTMSourceStatus	*_RO_RTM_SPY_SOURCE_MEM_STATUS	095D	12ba
MM::inputSpyMemRTMSystemDS2Word	p_inputSpyMemRTMSystemDS2Word	*_RW_RTM_SPY_SYSTEMDS2_MEM_WORD	095E - 0961	12bc - 12c2
MR::inputSpyMemRTMSystemDS2Control	p_inputSpyMemRTMSystemDS2Control	*_RW_RTM_SPY_SYSTEMDS2_MEM_CONTROL	0962	12c4
MR::inputSpyMemRTMSystemDS2Status	p_inputSpyMemRTMSystemDS2Status	*_RO_RTM_SPY_SYSTEMDS2_MEM_STATUS	0963	12c6
MM::inputSpyMemRTMSystemWord	p_inputSpyMemRTMSystemWord	*_RW_RTM_SPY_SYSTEM_MEM_WORD	0964 - 0967	12c8 - 12ce
MR::inputSpyMemRTMSystemControl	p_inputSpyMemRTMSystemControl	*_RW_RTM_SPY_SYSTEM_MEM_CONTROL	0968	12d0
MR::inputSpyMemRTMSystemStatus	p_inputSpyMemRTMSystemStatus	*_RO_RTM_SPY_SYSTEM_MEM_STATUS	0969	12d2
MM::inputSpyMemRTMSourceStartAddress	p_inputSpyMemRTMSourceStartAddress	*_RW_RTM_SPY_SOURCE_MEM_START_ADDRESS	096A - 096C	12d4 - 12d8
MM::inputSpyMemRTMSystemStartAddress	p_inputSpyMemRTMSystemStartAddress	*_RW_RTM_SPY_SYSTEM_MEM_START_ADDRESS	096D - 096F	12da - 12de
MM::inputSpyMemRTMSystemDS2StartAddress	p_inputSpyMemRTMSystemDS2StartAddress	*_RW_RTM_SPY_SYSTEMDS2_MEM_START_ADDRESS	0970 - 0972	12e0 - 12e4
MM::inputSpyMemRTMParityErrorCounter	p_inputSpyMemRTMParityErrorCounter	*_RO_RTM_PARITY_ERROR_COUNTER	0973 - 0978	12e6 - 12f0
MM::inputSpyMemRTMSourceCheckError	p_inputSpyMemRTMSourceCheckError	*_RO_RTM_SPY_SOURCE_MEM_CHECK_ERROR	0979 - 097E	12f2 - 12fc
MM::inputSpyMemRTMSystemDS2CheckError	p_inputSpyMemRTMSystemDS2CheckError	*_RO_RTM_SPY_SYSTEMDS2_MEM_CHECK_ERROR	097F - 0984	12fe - 1308
MM::inputSpyMemRTMSystemCheckError	p_inputSpyMemRTMSystemCheckError	*_RO_RTM_SPY_SYSTEM_MEM_CHECK_ERROR	0985 - 098A	130a - 1314
MR::inputSpyMemClockManagerStatus	p_ClockManagerStatus	*_RO_CLOCK_MANAGER_STATUS	098B	1316
MR::delayInputDataAdder	p_delayInputDataAdder	*_RW_DELAY_INPUT_DATA_ADDER	098C	1318
MR::backplaneInputmask	p_backplaneInputmask	*_RW_BACKPLANE_INPUT_CHANNEL_MASK	098D	131a
MR::rtmoutputCounterReset	p_rtmoutputCounterReset	*_RW_RTM_OUTPUT_COUNTER_RESET	098E	131c
MM::inputSpyMemRTMOutputWord	p_inputSpyMemRTMOutputWord	*_RW_RTM_OUTPUT_SPY_SYSTEM_MEM_WORD	098F - 0992	131e - 1324
MR::inputSpyMemRTMOutputControl	p_inputSpyMemRTMOutputControl	*_RW_RTM_OUTPUT_SPY_SYSTEM_MEM_CONTROL	0993	1326
MR::inputSpyMemRTMOutputStatus	p_inputSpyMemRTMOutputStatus	*_RO_RTM_OUTPUT_SPY_SYSTEM_MEM_STATUS	0994	1328
MM::inputSpyMemRTMOutputCheckError	p_inputSpyMemRTMOutputCheckError	*_RO_RTM_OUTPUT_SPY_SYSTEM_MEM_CHECK_ERROR	0995 - 099A	132a - 1334
MR::inputSpyMemRTMOutputStartAddress	p_inputSpyMemRTMOutputStartAddress	*_RW_RTM_OUTPUT_SPY_SYSTEM_MEM_START_ADDRESS	099B	1336
MR::spyMemoryInhibit	p_spyMemoryInhibit	*_RW_SPY_MEM_WRITE_INHIBIT	0948	1290
MM::DAQRAMrelativeOffset	p_daqRAMrelativeOffset	*_RW_DAQ_RAM_RELATIVE_OFFSET	099C - 09AE	1338 - 135c
MR::sumMETMask	p_sumMETMask	*_RW_SUMET_MASK	09AF	135e
MR::etMISSMask	p_etMISSMask	*_RW_MISSET_MASK	09B0	1360
MM::clockDetectCounter	p_clockDetectCounter	*_RO_CLOCK_DETECT_COUNTER	09B1 - 09C0	1362 - 1380
MR::quietForce	p_quietForce	*_RW_QUIET_FORCE	09C1	1382
MR::rtmMask	p_rtmMask	*_RW_RTM_INPUT_CHANNEL_MASK	09C2	1384
MR::daqROIstatus	p_daqROIstatus	*_RO_DAQ_ROI_STATUS	09C3	1386
MR::daqROIgtxReset	p_daqROIgtxReset	*_RW_DAQ_ROI_GTX_RESET	09C4	1388
MR::daqTOPOTRgtxStatus	p_daqTOPOTRgtxStatus	*_RO_TOPOTR_GTX_STATUS	09C5	138a
MR::rateCounterInhibit	p_rateCounterInhibit	*_RW_RATE_COUNTER_INHIBIT	09C6	138c
MR::rateCounterReset	p_rateCounterReset	*_RW_RATE_COUNTER_RESET	09C7	138e
MM::rateCounterNormalisation	p_rateCounterNormalisation	*_RO_RATE_NORMALISATION_COUNTER	09C8 - 09C9	1390 - 1392
MM::localRates	p_localRates	*_RO_MULT_LOCAL_COUNTER	09CA - 09FB	1394 - 13f6
MM::remoteRates	p_remoteRates	*_RO_MULT_REMOTE_COUNTER	09FC - 0A5C	13f8 - 14b8
MM::totalRates	p_totalRates	*_RO_MULT_TOTAL_COUNTER	0A5D - 0A8E	14ba - 151c
MM::tobCounter	p_tobCounter	*_RO_TOB_COUNTER	0A8F - 0AAE	151e - 155c
MM::localBackplaneOverflow	p_localBackplaneOverflow	*_RO_LOCAL_BACKPLANE_OVERFLOW_COUNTER	0AAF - 0ACE	155e - 159c
MM::globalBackplaneOverflow	p_globalBackplaneOverflow	*_RO_GLOBAL_BACKPLANE_OVERFLOW_COUNTER	0ACF - 0AD0	159e - 15a0
MM::totalBackplaneOverflow	p_totalBackplaneOverflow	*_RO_GLOBAL_BACKPLANE_OVERFLOW_COUNTER	0AD1 - 0AD2	15a2 - 15a4
MR::bcResetErrorCounterCounter	p_bcResetErrorCounterCounter	*_RO_BC_RESET_ERROR_COUNTER	0AD3	15a6

HDMC name and class	pointer name	VHD constant	byte ADDR	word ADDR
MR::bcResetErrorCounterReset	p_bcResetErrorCounterReset	*_RW_BC_RESET_ERROR_COUNTER_RESET	0AD4	15a8
MM::sumETCounter	p_sumETCounter	*_RO_SUM_ET_COUNTER	0AD5 - 0AE4	15aa - 15c8
MM::METCounter	p_METCounter	*_RO_MISSING_ET_COUNTER	0AE5 - 0AF4	15ca - 15e8
MM::METSignCounter	p_METSignCounter	*_RO_MISSING_ET_SIGN_COUNTER	0B05 - 0B14	160a - 1628
MM::sumETWeightedCounter	p_sumETWeightedCounter	*_RO_SUM_ET_WEIGHTED_COUNTER	0B15 - 0B24	162a - 1648
MM::METWeightedCounter	p_METWeightedCounter	*_RO_MISSING_ET_RES_COUNTER	0B25 - 0B34	164a - 1668
MM::presenceCounter	p_presenceCounter	*_RO_PRESENCE_COUNTER	0B35 - 0CF5	166a - 19ea
MR::disableOverflowMask	p_disableOverflowMask	*_RW_DISABLE_OVERFLOW_MASK	0CF6	19ec
MM::systemMonitor	p_systemMonitor	*_RO_SYSMON_DATA_BLOCK	0CF7 - 0D06	19ee - 1a0c
MM::etMissThreshold	p_etMissThreshold	*_RW_MISS_E_THR_BLOCK	0D07 - 0D16	1a0e - 1a2c
MM::etMissResThreshold	p_etMissResThreshold	*_RW_MISS_E_RES_THR_BLOCK	0D17 - 0D26	1a2e - 1a4c
MM::sumEtThreshold	p_sumEtThreshold	*_RW_SUM_ET_THR_BLOCK	0D27 - 0D2E	1a4e - 1a5c
MM::sumEtResThreshold	p_sumEtResThreshold	*_RW_SUM_ET_RES_THR_BLOCK	0D2F - 0D36	1a5e - 1a6c
MM::xsThreshold	p_xsThreshold	*_RW_XS_T2_A2_THR_BLOCK	0D37 - 0D46	1a6e - 1a8c
MM::etMissMinParam	p_etMissMinParam	*_RW_T_MISS_E_MIN_PARAM_BLOCK	0D47 - 0D56	1a8e - 1aac
MM::etMissMaxParam	p_etMissMaxParam	*_RW_T_MISS_E_MAX_PARAM_BLOCK	0D57 - 0D66	1aae - 1acc
MM::sumEtMinParam	p_sumEtMinParam	*_RW_T_SUM_E_MIN_PARAM_BLOCK	0D67 - 0D6E	1ace - 1adc
MM::sumEtMaxParam	p_sumEtMaxParam	*_RW_T_SUM_E_MAX_PARAM_BLOCK	0D6F - 0D76	1ade - 1aec
MM::xsParam	p_xsParam	*_RW_XS_B2_PARAM_BLOCK	0D77 - 0D7E	1aee - 1afc
MM::clockDiffCounter	p_clockDiffCounter	*_RO_CLOCK_DIFF_DETECT_COUNTER	0D7F - 0D9E	1afe - 1b3c
MM::clockDiffDuration	p_clockDiffDuration	*_RO_CLOCK_DIFF_DURATION_COUNTER	0D9F - 0DBE	1b3e - 1b7c
MM::clockDiffRachetUp	p_clockDiffRachetUp	*_RO_CLOCK_DIFF_RATCHET_UP_COUNTER	0DBF - 0DDE	1b7e - 1bbc
MM::clockDiffRachetDown	p_clockDiffRachetDown	*_RO_CLOCK_DIFF_RATCHET_DOWN_COUNTER	0DDF - 0DFE	1bbe - 1bfc
MR::cmxFlavour	p_cmxFlavour	*_RO_CMX_FLAVOR	24CC	4998
MR::versionCommon	p_versionCommon	*_RO_VERSION_COMMON	2800	5000
MR::versionFlavourCommon	p_versionFlavourCommon	*_RO_VERSION_FLAVOR_COMMON	2802	5004
MR::versionFlavourLocal	p_versionFlavourLocal	*_RO_VERSION_FLAVOR_LOCAL	2804	5008

[1] not used
* replaces ADDR_REG
MR is ModuleRegister16
MM is ModuleMemory16
Addresses are given as first–last inclusive and in hex

2.3 Online simulation

The online simulation utilises many classes that provide the simulation for the CMX, either within the online simulation or in the test tools used for the CMX logic test.

2.3.1 CMX dataformats

The simulation is based on classes defined in the CMX data formats. All the CMX data objects are defined in `Cmxdataformats.h` with very similar structure for the different CMX flavours. These classes hold single TOBs, which are `Cmxdataformats::TOBJET` for jet TOBs, `Cmxdataformats::TOBCP` for CP TOBs. Each TOB is stored with its η and ϕ position, which are abstract positions indicating the lower edge of the TOB in a regular grid of maximally $-3.2 \leq \eta < 3.2$ and $0 \leq \phi < 6.4$ in 0.1 steps and multiplied by 10, as only integer values are used in the objects. For the jet TOBs a reasonable granularity is 0.2, while CP TOBs use the full granularity of 0.1. The tools will treat jet TOBs with odd $\eta \times 10$ (e.g. $\eta = 31$) coordinates as TOBs overlapping with the TOBs with adjacent (and smaller) even $\eta \times 10$ coordinate (e.g. $\eta = 30$). This coordinate system has been chosen to be able to easily derive the η-threshold bin and do not need to reflect the true detector position. Hence, jet TOBs are allowed to have $-32 \leq \eta < 32$ with 32 η threshold bins and CP TOBs have $-25 \leq \eta < 25$ with 50 threshold bins. The ϕ coordinate simply indicates the quadrant (and hence crate) position. Functions exist to set and read the coordinates and also convert them into presence bit and local ROI position (`position()`, `setposition()`, `presence()`, `ROIvalue()`). Coordinates for CP TOBs can be optionally (`CPMcoordinate==true`) given as coordinates that are used for the CPM ROIs (the lowest of the chip number is used as the highest bit of the local coordinate, so that e.g. chip=13, LC=2 is chip=6, LC=6). The L1Topo word per TOB is also generated by the TOB classes (`TopoWord()`). A method to print the content is available and also a check, if the η/ϕ position is valid, can be made. A flag in the object indicates, if it could not be added to an input module (`Cmxdataformats::JETCMXdata` or `Cmxdataformats::CPCMXdata`) for a particular problem or it was added as an overflow TOB (`overflowTOB()`). Two functions have been added for convenience (`duplicate_tob*()`) to detect duplicate TOBs (same crate, input module and presence nit position) within a list of TOBs.

An excerpt of the header definitions of `TOBJET` and `TOBCP` can be found below:

```
namespace Cmxdataformats {
 class TOBJET {
  public:
   // constructors for empty TOBs, in eta/phi or hardware coordinates
   TOBJET( bool verbose=false );
   TOBJET( int eta, unsigned int phi, unsigned int energy_large , unsigned energy_small, bool verbose=false );
   TOBJET( unsigned int presencebit , unsigned int ROI, unsigned int jemno, unsigned int cratenr , unsigned int
      energy_large , unsigned energy_small , unsigned int overflowTOB=0, bool verbose=false );
   // getting / setting position , flags and energy values
   int position ( unsigned int &cratenumber, unsigned int &jemnumber, unsigned int &presence, unsigned int &
     ROIvalue ) const;
   void setposition ( unsigned int presencebit , unsigned int ROI, unsigned int jemno, unsigned int cratenr );
   TOBJET &energy_large( unsigned int new_energy_large );
   TOBJET &energy_small( unsigned int new_energy_small );
   TOBJET &eta( int new_eta );
   TOBJET &overflowTOB( unsigned int new_overflowTOB );
   TOBJET &phi( unsigned int new_phi );
   int eta () const;
   int flag () const;
   unsigned int ROIvalue() const;
   unsigned int cratenumber() const;
   unsigned int energy_large () const;
   unsigned int energy_small () const;
```

```
    unsigned int  etabin () ;
    unsigned int  jemnumber() const;
    unsigned int  overflowTOB() const;
    unsigned int  phi ()  const;
    unsigned int  presence ()  const;
    TOBJET &flag( int new_flag  ) ;
    // create topo word for a single TOB
    uint32_t  TopoWord();
    // print and checking for duplicate TOBs
    int  print (  int  mode=0 ) const;
    int   duplicate_tobjets  (  std :: vector  < TOBJET > jettobs ) ;
  };
```

```
class TOBCP {
 public:
    // constructors for empty TOBs, in eta/phi or hardware coordinates , CMX or CPM coordinates
    TOBCP( bool verbose=false ) ;
    TOBCP( int eta,  unsigned int  phi ,  unsigned int  energy,  unsigned  isolation ,  bool  verbose=false  ) ;
    TOBCP( unsigned int presencebit ,  unsigned int  ROI, unsigned int  cpno,  unsigned int  cratenr ,  unsigned int
      energy,  unsigned  isolation ,  unsigned int  overflowTOB=0, bool CPMcoordinate=false, bool verbose=false  ) ;
    // getting / setting  position ,  flags ,  energy and isolation  values
    int  position (  unsigned int  &cratenumber, unsigned int  &cpnumber, unsigned int &presence,  unsigned int  &
      ROIvalue, bool CPMcoordinate=false )  const;
    void  setposition (  unsigned int  presencebit ,  unsigned int  ROI, unsigned int  cpmno, unsigned int  cratenr ,  bool
      CPMcoordinate=false );
    TOBCP &energy( unsigned int new_energy );
    TOBCP &eta( int new_eta ) ;
    TOBCP &isolation( unsigned int  new_isolation  ) ;
    TOBCP &overflowTOB( unsigned int new_overflowTOB );
    TOBCP &phi( unsigned int new_phi );
    int  eta ()  const;
    int  flag ()  const;
    unsigned int  ROIvalue()  const;
    unsigned int  cpnumber() const;
    unsigned int  cratenumber()  const;
    unsigned int  energy ()  const;
    unsigned int  etabin () ;
    unsigned int   isolation  ()  const;
    unsigned int  overflowTOB() const;
    unsigned int  phi ()  const;
    unsigned int  presence ()  const;
    TOBCP &flag( int new_flag  ) ;
    // create topo word for a single TOB
    uint32_t  TopoWord();
    // print and check for duplicate TOBs
    int  print (  int  mode=0 ) const;
    int  duplicate_tobcps (  std :: vector  < TOBCP > cptobs );
};
}
```

The JEM and CP input channels are represented by `Cmxdataformats::JETCMXdata` or `Cmxdataformats :: CPCMXdata` objects. These objects have a fixed channel and crate number, so that when a list of TOBs is added to the objects (`inserttobs()`), only the TOBs matching to the location are added. TOBs that are overflowing the backplane are flagged as such and an internal overflow flag is set as well (`overflowTOB()`). A copy of the TOBs that have the correct position is kept in the object (`jettobs()`, `cptobs()`). The `*CMXdata` classes can read and create spy memory and normal data words (`readspymemwords()`, `readdatawords()`, `createspymemwords()`, `createdatawords()`). When reading data words, parity errors are detected and flagged (`parityerrordetected()`). In case of a parity error and depending on the configuration, the data words can be zeroed before the TOBs are extracted (hence the object does not contain any TOBs) When reading data words the TOBs are created with consistent η and ϕ position, i.e. always the smallest value is used. For overflow TOBs the η position is correct, but the local position is zero as well as the energy and isolation values.

An excerpt of the header definitions of `JETCMXdata` and `CPCMXdata` can be found below:

```
namespace Cmxdataformats {
 class JETCMXdata {
  public:
   // constructor with crate, crate position
   JETCMXdata( unsigned int createno, unsigned int jemno, bool cleardataonerror =true, bool setparityerrorbit =
      true, bool verbose=false );
   // reading/writing datawords, clearing of all TOBs
   unsigned int readdatawords( std::vector < uint32_t > &datawords, bool returndatawords =true );
   unsigned int readspymemwords( Bitcoder::spymemevent spymemdatawords );
   unsigned int createdatawords ( std::vector < uint32_t > &datawords );
   unsigned int createspymemwords( Bitcoder::spymemevent &spymemwords );
   void clearall ();
   // inserting /adding TOBs
   int inserttobs ( std::vector < TOBJET > &new_jettobs, bool flagtobs =false, bool testonly =false );
   int addtob( TOBJET &new_tob, bool flagtob=false, bool testonly =false );
   // setting / getting flags / list of TOBs
   bool overflow () const;
   unsigned int crateno () const;
   unsigned int jemno() const;
   bool parityerrordetected () const;
   std::vector < TOBJET > jettobs() const;
   JETCMXdata &crateno( unsigned int newcrateno );
   JETCMXdata &jemno( unsigned int newjemno );
   JETCMXdata &jettobs( std::vector < TOBJET > new_jettobs, bool flagtobs =false, bool testonly =false );
   bool cleardataonerror () const;
   bool setparityerrorbit () const;
   // printing of object
   int print ( int mode=0 ) const;
 };
```

```
 class CPCMXdata {
  public:
   // constructor with crate, crate position
   CPCMXdata( unsigned int createno, unsigned int cpno, bool cleardataonerror =true, bool setparityerrorbit =true,
      unsigned int internalparityerror =0, bool verbose=false );
   // reading/writing datawords, clearing of all TOBs
   unsigned int readdatawords( std::vector < uint32_t > &datawords, bool returndatawords =true );
   unsigned int readspymemwords( Bitcoder::spymemevent spymemdatawords );
   unsigned int createdatawords ( std::vector < uint32_t > &datawords );
   unsigned int createspymemwords( Bitcoder::spymemevent &spymemwords );
   void clearall ();
   // inserting /adding TOBs
   int inserttobs ( std::vector < TOBCP > &new_cptobs, bool flagtobs=false, bool testonly =false );
   int addtob( TOBCP &new_tob, bool flagtob=false, bool testonly =false );
   // setting / getting flags / list of TOBs
   bool overflow () const;
   unsigned int crateno () const;
   unsigned int cpno() const;
   unsigned int internalparityerror () const;
   std::vector < TOBCP > cptobs() const;
   CPCMXdata &crateno( unsigned int newcrateno );
   CPCMXdata &cpno( unsigned int newcpno );
   CPCMXdata &cptobs( std::vector < TOBCP > new_cptobs, bool flagtobs=false, bool testonly =false );
```

```
    CPCMXdata &internalparityerror( unsigned int newinternalparityerror );
    bool parityerrordetected () const;
    bool compare( CPCMXdata other );
    bool cleardataonerror () const;
    bool setparityerrorbit () const;
    // printing of object
    int print ( int mode=0 ) const;
};
}
```

For the energy CMX simulation there are no single energy objects, but there are already objects of the class `Cmxdataformats::ENERGYCMXdata` that hold the energy value from a certain JEM. E_x, E_y and E_T can be set individually (`et()`, etc.) without a check if the values are physical. Each `ENERGYCMXdata` object has a definite crate and JEM number for documentation purposes (a quadrant when adding the energies is assigned later in the `ENERGYCMX_RTMdata` class). Also flags for parity errors, detection thereof and clearing the data on a parity error can be set. As the other `*CMXdata` classes, it can read and create spy memory and normal data words.

An excerpt of the header definition of `ENERGYCMXdata` can be found below:

```
namespace Cmxdataformats {
 class ENERGYCMXdata {
  public:
    ENERGYCMXdata( unsigned int createno, unsigned int jemno, bool cleardataonerror =true, bool setparityerrorbit
      =true, bool verbose=false );
    // reading/writing datawords, clearing of energy values
    unsigned int readdatawords( std :: vector < uint32_t > &datawords, bool returndatawords =true );
    unsigned int readspymemwords( Bitcoder::spymemevent spymemdatawords );
    unsigned int createdatawords ( std :: vector < uint32_t > &datawords );
    unsigned int createspymemwords( Bitcoder :: spymemevent &spymemwords );
    void clearall ();
    // setting / getting flags / list of TOBs
    bool parityerrordetected () const;
    int setexeyet ( unsigned int ex, unsigned int ey, unsigned int et );
    int getexeyet ( unsigned int &ex, unsigned int &ey, unsigned int &et );
    unsigned int crateno () const;
    unsigned int jemno() const;
    ENERGYCMXdata &crateno( unsigned int newcrateno );
    ENERGYCMXdata &et( unsigned int newet );
    ENERGYCMXdata &ex( unsigned int newex );
    ENERGYCMXdata &ey( unsigned int newey );
    ENERGYCMXdata &jemno( unsigned int newjemno );
    unsigned int et () const;
    unsigned int ex () const;
    unsigned int ey () const;
    bool cleardataonerror () const;
    bool setparityerrorbit () const;
    bool verbose () const;
    // printing of object
    int print ( int mode=0 ) const;
 };
}
```

For the simulation of the crate-type functionality (i.e. for the local thresholding and energy summation) the `Cmxdataformats::*CMX_RTMdata` objects are used. These objects contain the output of the CMX logic, either threshold multiplicity counts or energy sums (full and restricted). Overflow flags (defined by the data format) are also stored (`overflowTOB()`). Reading and generating spy memory and data words is possible and the detection of parity errors and behaviour upon parity error detection is stored (`readspymemwords()`,`readdatawords()`, `createspymemwords()`, `createdatawords()`, `parityerrordetected()`). For testing purposes the unlimited (real) threshold multiplicity is stored as well, while the value that is read by the next step of the simulation is maxed out at the maximum value (`thresholdmultiplicity*`, `thresholdrealmultiplicity*`). For the energy CMX the energy sums, unrestricted and restricted are stored in the `ENERGYCMX_RTMdata` objects (`et()`, `et_res()` etc.) as well as the real value without the limitation on the bit length (`realet()`). Overflows for the each energy sum are also stored (`overflow*()`).

The data words for L1Topo are generated for the JET and CP CMX in the CMX simulation classes (`simulate*CMX_crate::createtopowords()`), while for the energy CMX the data class (`ENERGYCMX_RTMdata :: createtopowords()`) can directly create the data words. A defintion if the L1Topo dataformats can be found here [20].

An excerpt of the header definition of `JETCMX_RTMdata`, `CPCMX_RTMdata` and `ENERGYCMX_RTMdata` can be found below:

```
namespace Cmxdataformats {
 class JETCMX_RTMdata{
  public:
   // constructor
   JETCMX_RTMdata( bool cleardataonerror=true, bool setparityerrorbit =true, bool verbose=false );
   // reading/writing datawords, clearing of thresholds
   unsigned int readdatawords( std::vector<uint32_t> &datawords0, std::vector<uint32_t> &datawords1, bool
      returndatawords =true );
   unsigned int readspymemwords( Bitcoder::spymemevent spymemwords0, Bitcoder::spymemevent spymemwords1 );
   unsigned int createdatawords ( std::vector <uint32_t> &datawords0, std::vector<uint32_t> &datawords1 );
   unsigned int createspymemwords( Bitcoder::spymemevent &spymemwords0, Bitcoder::spymemevent &spymemwords1
      );
   void clearall ();
   // getting / setting multiplicities
   JETCMX_RTMdata &thresholdmultiplicity0( const std::vector < unsigned int > & new_thresholdmultiplicity0 );
   JETCMX_RTMdata &thresholdmultiplicity1( const std::vector < unsigned int > & new_thresholdmultiplicity1 );
   JETCMX_RTMdata &thresholdrealmultiplicity0( const std::vector < unsigned int > &
      new_thresholdrealmultiplicity0 );
   JETCMX_RTMdata &thresholdrealmultiplicity1( const std::vector < unsigned int > &
      new_thresholdrealmultiplicity1 );
   std::vector < unsigned int > thresholdmultiplicity0 () const;
   std::vector < unsigned int > thresholdmultiplicity1 () const;
   std::vector < unsigned int > thresholdrealmultiplicity0 () const;
   std::vector < unsigned int > thresholdrealmultiplicity1 () const;
   // getting / setting flags
   unsigned int overflowTOB() const;
   bool parityerrordetected () const;
   JETCMX_RTMdata &overflowTOB( unsigned int new_overflowTOB );
   bool cleardataonerror () const;
   bool setparityerrorbit () const;
   bool verbose () const;
   // printing of object
   int print ( int mode=0 ) const;
 };
```

```
 class CPCMX_RTMdata{
  public:
   // constructor
   CPCMX_RTMdata( bool cleardataonerror=true, bool setparityerrorbit =true, bool verbose=false );
   // reading/writing datawords, clearing of thresholds
   unsigned int readdatawords( std::vector <uint32_t> &datawords, bool returndatawords =true );
   unsigned int readspymemwords( Bitcoder::spymemevent spymemwords );
   unsigned int createdatawords ( std::vector <uint32_t> &datawords );
```

```
    unsigned int createspymemwords( Bitcoder::spymemevent &spymemwords );
    void clearall ();
    // getting / setting multiplicities
    CPCMX_RTMdata &thresholdmultiplicity( std::vector < unsigned int > new_thresholdmultiplicity );
    CPCMX_RTMdata &thresholdrealmultiplicity( std::vector < unsigned int > new_thresholdrealmultiplicity );
    std::vector < unsigned int > thresholdmultiplicity () const;
    std::vector < unsigned int > thresholdrealmultiplicity () const;
    // getting / setting flags
    unsigned int overflowTOB() const;
    CPCMX_RTMdata &overflowTOB( unsigned int new_overflowTOB );
    bool parityerrordetected () const;
    bool cleardataonerror () const;
    bool setparityerrorbit () const;
    bool verbose () const;
    // printing of object
    int print ( int mode=0 ) const;
  };
```

```
class ENERGYCMX_RTMdata{
  public:
    // constructor
    ENERGYCMX_RTMdata( bool cleardataonerror=true, bool setparityerrorbit=true, bool verbose=false );
    // reading/writing datawords, clearing of thresholds
    unsigned int readdatawords( std::vector <uint32_t> &datawords0, std::vector <uint32_t> &datawords1, bool
      returndatawords=true );
    unsigned int readspymemwords( Bitcoder::spymemevent spymemwords0, Bitcoder::spymemevent spymemwords1 );
    unsigned int createdatawords ( std::vector <uint32_t> &datawords0, std::vector <uint32_t> &datawords1 );
    unsigned int createspymemwords( Bitcoder::spymemevent &spymemwords0, Bitcoder::spymemevent &spymemwords1
      );
    void clearall ();
    // getting / setting energy values
    ENERGYCMX_RTMdata &et( unsigned int new_et );
    ENERGYCMX_RTMdata &et_res( unsigned int new_et_res );
    ENERGYCMX_RTMdata &overflow_et( unsigned int new_overflow_et );
    ENERGYCMX_RTMdata &overflow_et_res( unsigned int new_overflow_et_res );
    ENERGYCMX_RTMdata &realet( unsigned int new_et );
    ENERGYCMX_RTMdata &realet_res( unsigned int new_et_res );
    unsigned int et () const;
    unsigned int et_res () const;
    unsigned int realet () const;
    unsigned int realet_res () const;
    // [...] same for ex, ey, overflow flags are unsigned int, values are int
    // getting / setting flags
    bool parityerrordetected () const;
    bool cleardataonerror () const;
    bool setparityerrorbit () const;
    bool verbose () const;
    // create topo words for this CMX
    void createtopowords ( std::vector < uint32_t > &topowords, uint32_t bcid=0 );
    // print of object
    int print ( int mode=0 ) const;
  };
}
```

Finally, the system-type logic simulation of the CMX uses the `Cmxdataformats::*CMX_CTPdata` classes to store the output for the CTP. The functionality is very similar to the `*CMX_RTMdata` objects. All necessary flags are stored and in addition the real threshold multiplicity (`thresholdmultiplicity*()`). In case of the energy CMX object also the conditions $c_0 - c_3$ are stored.

An excerpt of the header definition of `JETCMX_CTPdata`, `CPCMX_CTPdata` and `ENERGYCMX_CTPdata` can be found below:

```
namespace Cmxdataformats {
 class JETCMX_CTPdata{
  public:
   // constructor
   JETCMX_CTPdata( bool cleardataonerror=true, bool setparityerrorbit =true, bool verbose=false );
   // reading/writing datawords, clearing of thresholds
   unsigned int readdatawords( uint32_t &dataword0, uint32_t &dataword1, bool returndatawords =true );
   unsigned int readspymemwords( Bitcoder::spymemevent datawords );
   unsigned int createdatawords ( uint32_t &dataword0, uint32_t &dataword1 );
   unsigned int createspymemwords( Bitcoder::spymemevent &spymemwords );
   void clearall ();
   // getting / setting multiplicities
   JETCMX_CTPdata &thresholdmultiplicity0( const std::vector < uint32_t > & new_thresholdmultiplicity0 );
   JETCMX_CTPdata &thresholdmultiplicity1( const std::vector < uint32_t > & new_thresholdmultiplicity1 );
   JETCMX_CTPdata &thresholdrealmultiplicity0( const std::vector < uint32_t > & new_thresholdrealmultiplicity0 );
   JETCMX_CTPdata &thresholdrealmultiplicity1( const std::vector < uint32_t > & new_thresholdrealmultiplicity1 );
   std::vector < uint32_t > thresholdmultiplicity0 () const;
   std::vector < uint32_t > thresholdmultiplicity1 () const;
   std::vector < uint32_t > thresholdrealmultiplicity0 () const;
   std::vector < uint32_t > thresholdrealmultiplicity1 () const;
   // getting / setting flags
   bool parityerrordetected () const;
   bool cleardataonerror () const;
   bool setparityerrorbit () const;
   bool verbose () const;
   int print ( int mode=0 ) const;
};
```

```
 class CPCMX_CTPdata{
  public:
   CPCMX_CTPdata( bool cleardataonerror=true, bool setparityerrorbit =true, bool verbose=false );
   ~CPCMX_CTPdata();
   unsigned int readdatawords( uint32_t &dataword0, uint32_t &dataword1, bool returndatawords =true );
   unsigned int readspymemwords( Bitcoder::spymemevent spymemwords );
   unsigned int createdatawords ( uint32_t &dataword0, uint32_t &dataword1 );
   unsigned int createspymemwords( Bitcoder::spymemevent &spymemwords );
   // getting / setting multiplicities
   CPCMX_CTPdata &thresholdmultiplicity( std::vector < unsigned int > new_thresholdmultiplicity );
   CPCMX_CTPdata &thresholdrealmultiplicity( std::vector < unsigned int > new_thresholdrealmultiplicity );
   std::vector < unsigned int > thresholdmultiplicity () const;
   std::vector < unsigned int > thresholdrealmultiplicity () const;
   void clearall ();
   // getting / setting flags
   bool parityerrordetected () const;
   bool compare( CPCMX_CTPdata other );
   bool cleardataonerror () const;
   bool setparityerrorbit () const;
   bool verbose () const;
   // print of object
   int print ( int mode=0 ) const;
 };
```

```
class ENERGYCMX_CTPdata{
  public:
   ENERGYCMX_CTPdata( bool cleardataonerror=true, bool setparityerrorbit=true, bool verbose=false );
   ~ENERGYCMX_CTPdata();
```

```
    unsigned int readdatawords( uint32_t &dataword0, uint32_t &dataword1, bool returndatawords=true );
    unsigned int readspymemwords( Bitcoder::spymemevent spymemwords );
    unsigned int createdatawords( uint32_t &dataword0, uint32_t &dataword1 );
    unsigned int createspymemwords( Bitcoder::spymemevent &spymemwords );
            unsigned int hitmap_met() const;
    // getting / setting energies and decision flags
    ENERGYCMX_CTPdata &c0( unsigned int new_c0 );
    unsigned int c0() const;
    // [...] same for c1 .. c3
    ENERGYCMX_CTPdata &hitmap_met( unsigned int new_hitmap_met );
    unsigned int hitmap_met() const;
    // [...] same for met_res, metsig, sumet, sumet_res
    ENERGYCMX_CTPdata &realet( unsigned int new_realet );
    ENERGYCMX_CTPdata &realet_res( unsigned int new_realet_res );
    unsigned int realet () const;
    unsigned int realet_res () const;
    // [...] same for ex, ey, but values are int
    // getting / setting flags
    bool parityerrordetected ();
    bool compare( ENERGYCMX_CTPdata other );
    bool cleardataonerror () const;
    bool setparityerrorbit () const;
    bool verbose() const;
    void clearall ();
    // print of object
    int print ( int mode=0 ) const;
};
}
```

The events from the spy memories are stored in a class that can be setup to be aware of the size of the spy memory and the format of the data words. Each event is represented by a `Bitcoder :: spymemevent` which is a `std::vector < uint32 >`. The spy memories in `L1CaloCMX` do setup those parameters correctly when requesting a `Cmxdataformats::SpyMemoryEvents` object.

The dimensions can be changed (`setsize()`, `setnevents()`, `setnchannels()`, `setconvertparameters()`). Events can be set for a certain channel and event number (`setevents()`), but also a list of events can be set for a certain channel (`setevents_channel()`) or at a certain event number for all channels (`setevents_event()`). Similarily events can be retrieved. Events be be rotated (`rotate()`), i.e. the event number is shifted by a certain number. Events can be matched with another `SpyMemoryEvents` object (`match()`), so that each event is compared and the best match on an event-by-event, data word-by-data word or event bit-by-bit basis is found. The parity can be calculated for all the events (`checkparity()`). Other functions are for printing (`printEvents()` and comparing (`compare()`) events, as well as saving (`write()`) events to disk or reading (`read()`) them from disk.

An excerpt of the header for the `SpyMemoryEvents` can be found below:

```
namespace Cmxdataformats {
 class SpyMemoryEvents {
  public:
    // constructor with dimensions for the spy memory events
    SpyMemoryEvents( unsigned int words, unsigned int channels, unsigned int events, unsigned int bitlength=32,
      unsigned int convertedbitlength=32, bool reverseinput=false, bool reverseoutput=false, std::string name=""
      );
    // getting and setting events, either single events for events of a channel or particular events from all
      channels
    Bitcoder::spymemevent getevent( unsigned int channel, unsigned int event, bool convert=false ) const;
    std::vector < Bitcoder::spymemevent > getevents_channel( unsigned int channel=0, bool convert=false ) const;
    std::vector < Bitcoder::spymemevent > getevents_event( unsigned int event, bool convert=false ) const;
    std::vector < std::vector < Bitcoder::spymemevent > > getevents( bool convert=false ) const;
```

```
std::vector<Bitcoder::spymemevent> getevents_channel(unsigned int channel, int startevent, int endevent=-1, bool convert=false) const;
std::vector<Bitcoder::spymemevent> getevents_event(unsigned int event, int startchannel, int endchannel=-1, bool convert=false) const;
std::vector<std::vector<Bitcoder::spymemevent> > getevents(int startevent=0, int endevent=-1, int startchannel=0, int endchannel=0, bool convert=false) const;
int setevent(unsigned int channel, unsigned int event, Bitcoder::spymemevent newevent, bool convert=false, bool force=false, bool test=false);
int setevents_channel(unsigned int channel, std::vector<Bitcoder::spymemevent> newevents, bool convert=false, bool force=false, bool test=false);
int setevents_event(unsigned int event, std::vector<Bitcoder::spymemevent> newevents, bool convert=false, bool force=false, bool test=false);
int setevents(std::vector<std::vector<Bitcoder::spymemevent> > newevents, bool convert=false, bool force=false, bool test=false);
int setevents_channel(unsigned int channel, std::vector<Bitcoder::spymemevent> newevents, int startevent, int endevent=-1, bool convert=false, bool force=false, bool test=false);
int setevents_event(unsigned int event, std::vector<Bitcoder::spymemevent> newevents, int startchannel, int endchannel=-1, bool convert=false, bool force=false, bool test=false);
int setevents(std::vector<std::vector<Bitcoder::spymemevent> > newevents, int startevent=0, int endevent=0, int startchannel=0, int endchannel=0, bool convert=false, bool force=false, bool test=false);
// rotate events
void rotateEvents(int steps);
// matching events either coded as a single float (best match event as integer, fraction of matches as float) or more verbose
float matchEvents(const SpyMemoryEvents &other, int mode, bool verbose=false);
void matchEvents(const SpyMemoryEvents &other, int mode, int &match, int &total, int &bestmatch, bool verbose=false);
// parity related functions
unsigned int calculateparity(unsigned int channel, unsigned int event, unsigned int starbit, unsigned int length, bool odd=true, bool verbose=false);
bool checkparityevent(unsigned int channel, unsigned int event, unsigned int starbit, unsigned int length, unsigned int paritbit, bool odd=true);
bool checkparitychannel(unsigned int channel, unsigned int starbit, unsigned int length, unsigned int paritbit, bool odd=true);
bool checkparitychannel(unsigned int channel, unsigned int starbit, unsigned int length, unsigned int paritybit, std::vector<bool> &paritycheck, bool odd=true);
bool checkparity(unsigned int starbit, unsigned int length, unsigned int paritbit, bool odd=true);
bool checkparity(unsigned int starbit, unsigned int length, unsigned int paritybit, std::vector<bool> &paritycheck, bool odd=true);
// printing/formating events for printout
void reformat(const Bitcoder::spymemevent &event, bool convert=false, bool linebreak=true, bool hexprefix=false);
void printEvents(bool convert=false, bool full=false, int channel=-1, int event=-1, bool headless=false, bool linebreak=true, bool hexprefix=false);
// comparing, reading/writing
int compare(const SpyMemoryEvents &patternB, bool printdiff=true, bool printsame=false, bool convert=false, int mode=0, int channel=-1);
int compare(const SpyMemoryEvents &patternB, std::vector<uint32_t> &diffindex, bool printdiff=true, bool printsame=false, bool convert=false, int mode=0, int channel=-1);
int write(const std::string filename, bool append=false, bool convert=false, bool oldformat=false);
int read(std::string filename, int startat=0, int length=0, bool convert=false, bool oldformat=false);
// checking, clearing, expanding internal vectors, changing dimensions of object
bool check(bool verbose=false);
bool clear();
void expand();
unsigned int words() const;
unsigned int channels() const;
unsigned int events() const;
unsigned int bitlength() const;
unsigned int convertedbitlength() const;
```

```
    bool reverseinput () const;
    bool reverseoutput () const;

    void setsize ( unsigned int newwords, unsigned int newchannels, unsigned int newevents );
    void setnevents ( unsigned int newevents );
    void setnchannels ( unsigned int newchannels );
    void setconvertparameters ( unsigned int bitlength , unsigned int convertedbitlength , bool reverseinput , bool
        reverseoutput );
    Cmxdataformats::SpyMemoryEvents clone( unsigned int startevent =0, unsigned int length=0 );
  };
}
```

2.3.2 CMX simulation objects

The classes concerning the CMX simulation can be found in `Cmxsimulation.h`.

The simulation is demonstrated here with the L1Calo simulation. A standalone simulation as it is realised in the `cmxSimLab` program is also compared.

It should be noted, that the crate(-type) simulation refers to the crate-type functionality, i.e. the calculation of the local result (local summing). This means that also the system-type CMX (physical) has to have crate-type CMX functionality in addition to the global result (global summing).

The simulation matches the firmware functionality as it is described in the CMX firmware documentation. This has also been tested thoroughly as described in the next section.

The JET CMX simulation starts from `JETCMXdata` objects (2×16 objects). Either these are filled with `TOBJET` objects or directly from data words. The examples either simulate the whole L1Calo system (like in `cmxSimLab`) or just one CMX (like in the CMX online simulation).

```
// cmxSimLab
std :: vector < Cmxdataformats::TOBJET > tobjets;
std :: vector < std :: vector < Cmxdataformats::JETCMXdata > > jetcmxdata;
jetcmxdata. resize ( 2 );
for ( int crate =0;crate < 2; crate ++ ) {
    for( int jemnr=0;jemnr< 16;jemnr++ ){
        Cmxdataformats::JETCMXdata jetcmx( crate, jemnr );
        jetcmx. inserttobs ( tobjets );
        jetcmxdata[ crate ]. push_back( jetcmx );
    }
}
```

```
// online simulation
std :: vector < std :: vector < uint32_t > > m_input;
m_input. resize ( 16 );
std :: vector < Cmxdataformats::JETCMXdata > jetcmxdata;
for ( int n=0;n<16;n++ ){
    m_input[n]. resize ( 4 );
    Cmxdataformats::JETCMXdata thisjetcmx( 0,n, quiet );
    jetcmxdata .push_back( thisjetcmx );
    unsigned int result = jetcmxdata[n]. readdatawords( m_input[n] );
}
```

A threshold map in the proper format has to be provided which is the same as used in the `cmxServices` class. The map is either generated by hand, read from the database or created from the threshold registers. With the threshold map and some additional flags that steer the behaviour (e.g. `force` flag), a simulation object (`Cmxsimulation::simulateJETCMX_crate`) is created. The `simulateJETCMX_crate::simulate()` function creates from the `JETCMXdata` objects (a vector of 16 per crate) a `JETCMX_RTMdata` object.

```
namespace Cmxsimulation{
 class simulateJETCMX_crate {
  public:
   // constructor with location, threshold maps and other flags for the simulation
   simulateJETCMX_crate( unsigned int crateno, std::vector < std::vector < uint32_t > > thresholdmap, unsigned
       int overflowmask, bool force, bool quiet, bool bugfix , bool verbose=false );
   simulateJETCMX_crate( unsigned int crateno, unsigned int overflowmask, bool force, bool quiet, bool bugfix ,
       bool verbose=false );
   // simulation using JETCMXdata as inputs and JETCMX_RTMdata as output
   Cmxdataformats::JETCMX_RTMdata simulate( const std::vector < Cmxdataformats::JETCMXdata > &jetcmxdata );
   // creation of topowords from all TOBs in all JETCMXdata objects for this crate
   void createtopowords( const std::vector < Cmxdataformats::JETCMXdata > &jetcmxdata, std::vector < std::vector
       < uint32_t > > &topowords, uint32_t bcid=0 );
   // from datawords create all JETCMXdata objects for the whole crate, used if only datawords, but no
       JETCMXdata objects are not available
   std::vector < Cmxdataformats::JETCMXdata > generateJETCMXdata( std::vector < std::vector < uint32_t > > &
       datawords, bool autodetectword=true, bool spymemwords=false );
   // getting flags
   bool force() const;
   bool quiet() const;
   bool bugfix() const;
 };
```

```
namespace Cmxsimulation{
 class simulateCPCMX_crate {
  public:
   // constructor with location, threshold maps and other flags for the simulation
   simulateCPCMX_crate( unsigned int crateno, std::vector < std::vector < uint32_t > > thresholdmap, unsigned int
        overflowmask, bool force, bool quiet, bool bugfix, bool verbose=false );
   simulateCPCMX_crate( unsigned int crateno, unsigned int overflowmask, bool force, bool quiet, bool bugfix,
       bool verbose=false );
   // simulation using CPCMXdata as inputs and CPCMX_RTMdata as output
   Cmxdataformats::CPCMX_RTMdata simulate( const std::vector < Cmxdataformats::CPCMXdata > &cpcmxdata );
   // creation of topowords from all TOBs in all CPCMXdata objects for this crate
   void createtopowords( const std::vector < Cmxdataformats::CPCMXdata > &cpcmxdata, std::vector < std::vector <
        uint32_t > > &topowords, uint32_t bcid=0 );
   // from datawords create all CPCMXdata objects for the whole crate, used if only datawords, but no
       CPCMXdata objects are not available
   std::vector < Cmxdataformats::CPCMXdata > generateCPCMXdata( std::vector < std::vector < uint32_t > > &
       datawords, bool autodetectword=true, bool spymemwords=false );
   // getting flags
   bool force() const;
   bool quiet() const;
   bool bugfix() const;
 };
```

```
// cmxSimLab
for( int crate=0; crate < 2; crate++ ) {
    Cmxdataformats::JETCMX_RTMdata jetcmx_rtmdata= jetcmx_crate_simulations[crate].simulate( jetcmxdata[crate] )
        ;
    jetcmx_rtmdatas.push_back( jetcmx_rtmdata );
}
```

```
// online simulation
Cmxsimulation::simulateJETCMX_crate jetcmx_crate_simulation( 0,thresholdmap,0, force, quiet, false );
Cmxdataformats::JETCMX_RTMdata jetcmx_rtmdata= jetcmx_crate_simulation.simulate( jetcmxdata );
```

For the online simulation the data words for the RTM output data has to be created (in `cmxSimLab` the `JETCMX_RTMdata` objects can be used directly without the back-and-forth conversion into data words).

```
// online simulation
std :: vector < uint32_t > datawords0;
std :: vector < uint32_t > datawords1;
jetcmx_rtmdata . createdatawords ( datawords0,datawords1 ) ;
Cmxbitcoder:: writetoIntPort ( getOutputPort (0) , datawords0 ,2 ) ;
Cmxbitcoder:: writetoIntPort ( getOutputPort (1) , datawords1 ,2 ) ;
```

The online simulation creates two more data structures, the topo data words and the data words for the readout (glink). For the glink data there are no data classes, but the data words are created on the spot.

```
// online simulation
std :: vector < std :: vector < uint32_t > > topowords;
jetcmx_crate_simulation . createtopowords ( jetcmxdata , topowords ) ;
for ( size_t nfibre =0; nfibre <( size_t ) std :: min(12,( int )topowords. size () ) ; nfibre ++ ) {
    Cmxbitcoder:: writetoIntPort ( getTopoOutputPort(0, nfibre ) , topowords[ nfibre ],4 ) ;
}

Cmxbitcoder::Glinkdata glinkJetData ;
glinkJetData . clearframes () ;
for( int n=0;n < 16;n++ ) {
    for( int m=0;m<4;m++ ){
        glinkJetData . writedata ( n,m*24,24,m_input[n][m] ) ;
    }
}
glinkJetData . writedata ( 17,0,15, datawords0[0] ) ;
[...]
glinkJetData . writebit ( 17,64, Bitcoder :: getbit (datawords0 [1],15) ) ;
glinkJetData . writeWords( getDaqOutputPort() ,96 ) ;
```

The system-type simulation in `cmxSimlab` directly uses the `JETCMX_RTMdata` objects, while the online simulation recreates these objects from the data words. The simulation object `simulateJETCMX_system` creates two new objects, another `JETCMX_RTMdata` and a `JETCMX_CTPdata` object. Both hold the global sums, but in two different data objects. The order of the `JETCMX_RTMdata` that is in the `std::vector` which is given to the `simulate()` function determines which is the local result from the crate-type CMX and which is the local result of the system-type CMX result. Insofar the crate numbering in the `JETCMX_RTMdata` is not important.

```
namespace Cmxsimulation{
 class simulateJETCMX_system {
  public:
   // constructor
   simulateJETCMX_system( bool force, bool quiet , bool bugfix , bool verbose=false ) ;
   // simulation from JETCMX_RTMdata objects
   Cmxdataformats::JETCMX_CTPdata simulate( const std:: vector < Cmxdataformats::JETCMX_RTMdata > &jetcmxdata,
       Cmxdataformats::JETCMX_RTMdata &jetcmx_rtm_total );
   // create JETCMX_RTMdata from datawords, used before simulating from JETCMX_RTMdata objects
   std :: vector < Cmxdataformats::JETCMX_RTMdata > generateJETCMX_RTMdata( std::vector < std::vector < uint32_t >
       > &datawords0, std :: vector < std :: vector < uint32_t > > &datawords1, bool autodetectword=true, bool
       spymemwords=false );
   // getting flags
   bool force () const;
   bool quiet () const;
   bool bugfix () const;
};
```

```
namespace Cmxsimulation{
class simulateCPCMX_system {
 public:
   // constructor
   simulateCPCMX_system( bool force, bool quiet , bool bugfix , bool verbose=false );
   // simulation from CPCMX_RTMdata objects
   Cmxdataformats::CPCMX_CTPdata simulate( const std::vector < Cmxdataformats::CPCMX_RTMdata > &cpcmxrtmdata
        , Cmxdataformats::CPCMX_RTMdata &cpcmx_rtm_total );
   // create COCMX_RTMdata from datawords, used before simulating from CPCMX_RTMdata objects
   std :: vector < Cmxdataformats::CPCMX_RTMdata > generateCPCMX_RTMdata( std::vector < std::vector < uint32_t > >
        &datawords, bool autodetectword=true, bool spymemwords=false );
   // getting flags
   bool force () const;
   bool quiet () const;
   bool bugfix () const;
};
```

```
// cmxSimLab
Cmxdataformats::JETCMX_RTMdata totaljetdata;
Cmxdataformats::JETCMX_CTPdata jetcmx_ctpdata= jetcmx_system_simulation.simulate( jetcmx_rtmdatas , totaljetdata
        );
```

```
// online simulation
std :: vector < uint32_t > localdatawords0 , localdatawords1 ;
std :: vector < uint32_t > remotedatawords0, remotedatawords1;
remotedatawords0. resize ( 2 );
remotedatawords1. resize ( 2 );
std :: vector < Cmxdataformats::JETCMX_RTMdata > jetcmxrtmdata;
Cmxdataformats::JETCMX_RTMdata localjetcmxrtmdata( quiet );
Cmxdataformats::JETCMX_RTMdata remotejetcmxrtmdata( quiet );
Cmxdataformats::JETCMX_RTMdata total_jetcmxrtmdata( quiet );
unsigned int result =localjetcmxrtmdata . readdatawords( localdatawords0 , localdatawords1 );
 result =remotejetcmxrtmdata.readdatawords( remotedatawords0,remotedatawords1 );
Cmxsimulation::simulateJETCMX_system jetcmx_system_simulation( force , quiet , false );
Cmxdataformats::JETCMX_CTPdata jetcmxctpdata=jetcmx_system_simulation.simulate( jetcmxrtmdata,
        total_jetcmxrtmdata );
```

Again additional data words are produced for the output.

```
std :: vector < uint32_t > totaldatawords0 , totaldatawords1 ;
 total_jetcmxrtmdata . createdatawords ( totaldatawords0 , totaldatawords1 );
uint32_t dataword0, dataword1;
jetcmxctpdata . createdatawords ( dataword0,dataword1 );
Cmxbitcoder:: writetoIntPort ( getOutputPort (0) , dataword0 );
Cmxbitcoder:: writetoIntPort ( getOutputPort (1) , dataword1 );

Cmxbitcoder::Glinkdata glinkJetData ;
 glinkJetData . clearframes () ;

 glinkJetData . writedata ( 16,0,15, remotedatawords0[0] );
  [...]
 glinkJetData . writebit ( 18,64, Bitcoder :: getbit ( totaldatawords0 [1],15) );
 glinkJetData . writeWords( getDaqOutputPort() ,96 );
```

The simulation of the CP CMX is very similar via the `Cmxsimulation::simulateCPCMX_*` classes. The only differences are the number of CPMs and the number of RTM input channels. For the simulation 16 are still needed (to properly account for the numbering), but channels 0 and 15 are ignored (for the simulation object the order is important, not the channel number of the `CPCMXdata` object). There are four crates of which three are (physical) inputs to the system-type CMX, so that the `simulateCPCMX_system::simulate()` function needs four `CPCMXdata` objects. From these four `CPCMXdata` objects, four `SpyMemoryEvents` objects with one channel each can be made. When the (physical) RTM input spy memory on the system-type CMX is read, it is actually represented by one `SpyMemoryEvents` object composed of three channels instead of three with one channel each (note that the two "channels" of the JET/ENERGY RTM input and outputs refer to the two cables). The merging is simply made with the following lines, using `rtmevents` as the RTM output from each channel and `rtminevents` as the input to the system-type CMX.

```
// cmxSimLab
rtminevents= rtmevents [0];
rtminevents . setnchannels ( 3 ) ;
for( unsigned int n=1;n<3;n++ ) {
    rtminevents . setevents_channel ( n, rtmevents [n ]. getevents_channel (0) ) ;
}
```

The energy simulation (`Cmxsimulation::simulateENERGYCMX_*`) is similar with the exception that it does not start from single energy elements. The number of data words is similar to the JET simulation. The topo words are generated by the `ENERGYCMX_RTMdata` object.

```
namespace Cmxsimulation{
class simulateENERGYCMX_crate {
 public:
  // constructor
  simulateENERGYCMX_crate( unsigned int crateno, unsigned int restrictedrangeTE , unsigned int restrictedrangeXE ,
      bool force , bool quiet , bool bugfix , bool verbose=false ) ;
  // simulation from ENERGYCMXdata objects
  Cmxdataformats::ENERGYCMX_RTMdata simulate( const std::vector < Cmxdataformats::ENERGYCMXdata > &
      energycmxdata );
  // create ENERGYCMXdata objects from datawords
  std :: vector < Cmxdataformats::ENERGYCMXdata > generateENERGYCMXdata( std::vector < std::vector < uint32_t >
      > &datawords, bool autodetectword=true, bool spymemwords=false );
  // getting flags
  bool force () const;
  bool quiet () const;
  bool bugfix () const;
};
```

```
namespace Cmxsimulation{
class simulateENERGYCMX_system {
 public:
  // constructor
  simulateENERGYCMX_system( std::vector < std::vector < uint32_t > > thresholdmap, bool force , bool quiet , bool
      bugfix , bool verbose=false ) ;
  // simulation from ENERGYCMX_RTMdata objects
  Cmxdataformats::ENERGYCMX_CTPdata simulate( const std::vector < Cmxdataformats::ENERGYCMX_RTMdata > &
      energycmxdata, Cmxdataformats::ENERGYCMX_RTMdata &energycmx_rtm_total );
  // create ENERGYCMX_RTMdata objects from datawords
  std :: vector < Cmxdataformats::ENERGYCMX_RTMdata > generateENERGYCMX_RTMdata( std::vector < std::vector
      < uint32_t > > &datawords0, std::vector < std::vector < uint32_t > > &datawords1, bool autodetectword=true, bool
      spymemwords=false );
  // getting flags
  bool force () const;
  bool quiet () const;
  bool bugfix () const;
};
}
```

2.3.3 CMX simulation algorithms in software

The JET CMX algorithms are the same as in the firmware. Notable facts are the window size definition. If bit 11 of the threshold is set, then the small energy is used for comparison. The comparison of the energy values the energy is required to be larger than the threshold (not larger or equal).

```
Cmxdataformats::JETCMX_RTMdata Cmxsimulation::simulateJETCMX_crate::simulate( const std::vector <
      Cmxdataformats::JETCMXdata > &jetcmxdata ){
   // [...]
   // looping over all 25 thresholds , 10 7-bit, 15 3-bit counts all available JEMCMXdata objects
   for (unsigned int thresn=0; thresn<25; thresn++){
      for (unsigned int jem=0; jem<(unsigned int)std::min(16,(int)jetcmxdata.size()); jem++) {
         parityerrordetected |=jetcmxdata[jem].parityerrordetected();
         jettobs = jetcmxdata[jem].jettobs();
         if (jetcmxdata[jem].overflow() && Bitcoder::getbit(m_overflowmask,jem)==0) overflowTOB=1;
         // looping over all TOBs, max 4
         for (size_t j=0; j < (size_t)std::min(4,(int)jettobs.size()); j++) {
            unsigned int etabin =jettobs[j].etabin();
            uint32_t threshold=-1;
            if (m_thresholdmap.size()>etabin && m_thresholdmap[etabin].size()>thresn){
               threshold = m_thresholdmap[etabin][thresn];
            }
            unsigned int windowsize = Bitcoder::getbit(threshold,10);
            threshold = Bitcoder::getbits(threshold,0,10);
            uint32_t jetenergy;
            // windowsize == 0 -> large energy
            if (windowsize==1) {
               jetenergy=jettobs[j].energy_small();
            } else {
               jetenergy=jettobs[j].energy_large();
            }
            // threshold is defined as ">", not ">="
            if (jetenergy > threshold){
               if (thresn<10){
                  thresholdmultiplicity0[thresn]+=1;
                  thresholdrealmultiplicity0[thresn]+=1;
               } else {
                  thresholdmultiplicity1[thresn-10]+=1;
                  thresholdrealmultiplicity1[thresn-10]+=1;
               }
            }
         }
      }
   }
   // [...]
   // overflow and parity error handling
   if ((m_force && parityerrordetected) || (overflowTOB==1)) {
      for (int thresn=0; thresn<25; thresn++){
         if (thresn<10){
            thresholdmultiplicity0[thresn]=7;
         } else {
            thresholdmultiplicity1[thresn-10]=3;
         }
      }
   }
   // maxing out of thresholds
   for (unsigned int thresn=0; thresn<25; thresn++){
      if (thresn<10){
         if ( thresholdmultiplicity0[thresn] > 7) {
            thresholdmultiplicity0[thresn]=7;
```

```
            }
        } else {
            if ( thresholdmultiplicity1 [thresn−10] > 3) {
                thresholdmultiplicity1 [thresn−10]=3;
            }
        }
    }
    // [...]
}


Cmxdataformats::JETCMX_CTPdata Cmxsimulation::simulateJETCMX_system::simulate( const std::vector <
    Cmxdataformats::JETCMX_RTMdata > &jetcmxrtmdata, Cmxdataformats::JETCMX_RTMdata &jetcmx_rtm_total ){
    // [...]
    // looping over both threshold sets and adding the threshold counts, maxing out at max bit size
    for (int i=0; i< 10; i++) {
        unsigned int sum=0;
        for ( size_t n=0; n<jetcmxrtmdata.size () ; n++) {
            sum+=jetcmxrtmdata[n]. thresholdmultiplicity0 () [i ];
        }
        thresholdrealmultiplicity0 [i]=sum;
        if (sum>7) {
            sum=7;
        }
        thresholdmultiplicity0 [i]=sum;
    }
    for (int i=0; i< 15; i++) {
        unsigned int sum=0;
        for ( size_t n=0; n<jetcmxrtmdata.size () ; n++) {
            sum+=jetcmxrtmdata[n]. thresholdmultiplicity1 () [i ];
        }
        thresholdrealmultiplicity1 [i]=sum;
        if (sum>3) {
            sum=3;
        }
        thresholdmultiplicity1 [i]=sum;
    }
    // [...]
    // parity error handeling
    if ((m_force && parityerrordetected ) ) {
        for (int thresn=0; thresn<25; thresn++){ // loop over thresholds
            if (thresn<10){ // check for the first 10 thresholds with range A
                thresholdmultiplicity0 [thresn]=7;
            } else { // check for the remaining 15 thresholds with range B
                thresholdmultiplicity1 [thresn−10]=3;
            }
        }
    }
    // [...]
}
```

The CP CMX algorithms identical to the algorithms in the firmware. Also here the comparison of the energy values require the energy to be larger than the threshold. The isolation bit mask must have at least the same bits set as the required isolation bit mask.

```
Cmxdataformats::CPCMX_RTMdata Cmxsimulation::simulateCPCMX_crate::simulate( const std::vector < Cmxdataformats
    ::CPCMXdata > &cpcmxdata ){
    // [...]
    // loop over all thresholds and CPMs, ignore CPM 0 and CPM 15
    for ( unsigned int thresn = 0; thresn < 16; thresn++) {
```

```
            for (unsigned int cpm=1; cpm<(size_t)std :: min(15,( int )cpcmxdata.size () ); cpm++) {
                parityerrordetected |=cpcmxdata[cpm]. parityerrordetected () ;
                cptobs=cpcmxdata[cpm].cptobs();
                if (cpcmxdata[cpm].overflow()==1 && Bitcoder:: getbit (m_overflowmask,cpm) ==0) overflowTOB=1;
                // looping over all TOBs, max 5
                for ( size_t em=0; em < ( size_t ) std :: min(5,( int )cptobs . size () ) ; em++) {
                    unsigned int emenergy= cptobs[em].energy() ;
                    unsigned int emisolation = cptobs [em]. isolation () ;
                    unsigned int etabin = cptobs [em]. etabin () ;
                    unsigned int threshold =−1;
                    unsigned int isolationCriteria ;
                    if (m_thresholdmap. size () >etabin && m_thresholdmap[etabin]. size () >thresn) {
                        threshold =m_thresholdmap[etabin][ thresn ];
                    }
                    isolationCriteria =Bitcoder :: getbits ( threshold ,8,5) ;
                    threshold =Bitcoder :: getbits ( threshold ,0,8) ;
                    // energy threshold '>'
                    if ( (emenergy > threshold ) && (emisolation & isolationCriteria ) == isolationCriteria ) {
                        thresholdmultiplicity [ thresn ] +=1;
                        thresholdrealmultiplicity [ thresn ] +=1;
                    }
                }
            }
        }
        // [...]
        // maxing out thresholds
        for ( unsigned int thresn = 0; thresn < 16; thresn ++) {
            if ( thresholdmultiplicity [ thresn ] > 7) {
                thresholdmultiplicity [ thresn ] =7;
            }
        }
        // overflow and parity error handling
        if ((m_force && parityerrordetected ) || (overflowTOB==1)) {
            for (int thresn =0; thresn <16; thresn ++){
                thresholdmultiplicity [ thresn ]=7;
            }
        }
        // [...]
}


Cmxdataformats::CPCMX_CTPdata Cmxsimulation::simulateCPCMX_system::simulate( const std::vector <
    Cmxdataformats::CPCMX_RTMdata > &cpcmxrtmdata, Cmxdataformats::CPCMX_RTMdata &cpcmx_rtm_total ){
        // [...]
        // looping over all threshold multiplicities and adding then
        for ( size_t rtmin=0; rtmin<cpcmxrtmdata.size() ; rtmin++) {
            for (int thresn =0; thresn <16; thresn ++) {
                thresholdmultiplicity [ thresn ]+=cpcmxrtmdata[rtmin]. thresholdmultiplicity () [ thresn ];
                thresholdrealmultiplicity [ thresn ]+=cpcmxrtmdata[rtmin]. thresholdmultiplicity () [ thresn ];
            }
        }
        // [...]
        // maxing out of multiplicities
        for (int thresn =0; thresn <16; thresn ++) {
            if ( thresholdmultiplicity [ thresn ] >7) {
                thresholdmultiplicity [ thresn ] =7;
            }
        }
        // parity error handling
        if ((m_force && parityerrordetected )) {
```

```
        for (int thresn=0; thresn<16; thresn++){
            thresholdmultiplicity [thresn]=7;
        } // end loop over thresholds
    }
    // [...]
}
```

The ENERGY CMX algorithms are almost identical to the algorithms in the firmware. Some caution has to be taken with the threshold calculation from float values and calculation with large integers. For the threshold registers, the threshold values are calculated as floats and truncated before the comparison, so that this is aquivalent to the way thresholds are handled during the configuration and in the firmware. However, the internal calculations in the firmware are done with 96-bit precision, while the simulation is using floats, although all calculations do not yield fractional numbers, since square-roots and divisions are avoided. This was done merely to extend the numerical range beyond the 32/64-bit range of the integers. So far no differences have been seen between the firmware and software calculation.

```
Cmxdataformats::ENERGYCMX_RTMdata Cmxsimulation::simulateENERGYCMX_crate::simulate( const std::vector <
     Cmxdataformats::ENERGYCMXdata > &energycmxdata ){
    // copy of old simulation, using arrays with 0=ex, 1=ey, 2=et
    for (int i = 0; i < 3; i++){
        sumJem[i]            = 0;
        crateSum[i]          = 0;
        crateSum_res[i]      = 0;
        overflowCrate[i]     = 0;
        overflowCrate_res[i] = 0;
    }
    // loop over all jems
    for ( size_t jemno = 0 ; jemno < 16 ; ++jemno ) {
        parityerrordetected |=energycmxdata[jemno]. parityerrordetected ();
        sumJem[0] = energycmxdata[jemno].ex();
        sumJem[1] = energycmxdata[jemno].ey();
        sumJem[2] = energycmxdata[jemno].et();
        // et is simple sum
        crateSum[2] += sumJem[2];
        // checking restriction
        if ( Bitcoder :: getbit (m_restrictedrangeTE ,jemno) == 1 ) {
            crateSum_res[2] += sumJem[2];
        }
        // for ex, ey, first quadrant is +ex, +ey
        if ( jemno < 8 ){
            crateSum[0] += sumJem[0];
            crateSum[1] += sumJem[1];
            // checking restricted range
            if ( Bitcoder :: getbit (m_restrictedrangeXE ,jemno) == 1 ) {
                crateSum_res[0] += sumJem[0];
                crateSum_res[1] += sumJem[1];
            }
        }
        else{
        // for ex, ey, second quadrant is -ex, -ey
            crateSum[0] -= sumJem[0];
            crateSum[1] -= sumJem[1];
            // checking restricted range
            if ( Bitcoder :: getbit (m_restrictedrangeXE ,jemno) == 1 ) {
                crateSum_res[0] -= sumJem[0];
                crateSum_res[1] -= sumJem[1];
            }
        }
```

```
    } // end of JEM module No. loop
    // [...]
    // checking for overflow
    for ( unsigned int i=0; i<3; ++i){
        if ( crateSum[i] >= 0x4000 || crateSum[i]< −0x4000) {
            overflowCrate[i] = 1;
        }
    }
    // [...]
    for ( unsigned int i=0; i<3; ++i){
        if ( crateSum_res[i] >= 0x4000 || crateSum_res[i]< −0x4000 ) {
            overflowCrate_res [i] = 1; // set crate overflow
        }
    }
    // [...]
}

Cmxdataformats::ENERGYCMX_CTPdata Cmxsimulation::simulateENERGYCMX_system::simulate( const std::vector <
        Cmxdataformats::ENERGYCMX_RTMdata > &energycmxrtmdata, Cmxdataformats::ENERGYCMX_RTMdata &
        energycmx_rtm_total ){
    // [...]
    // summing over all crates
    for ( size_t encmx=0; encmx<(size_t)std::min(2,(int)energycmxrtmdata.size()); encmx++) {
        crateSum[0][encmx]       = energycmxrtmdata[encmx].ex();
        crateSum[1][encmx]       = energycmxrtmdata[encmx].ey();
        crateSum[2][encmx]       = energycmxrtmdata[encmx].et();
        crateSum_res [0][ encmx]  = energycmxrtmdata[encmx].ex_res();
        crateSum_res [1][ encmx]  = energycmxrtmdata[encmx].ey_res();
        crateSum_res [2][ encmx]  = energycmxrtmdata[encmx].et_res();
        overFlowCable[0][encmx] = energycmxrtmdata[encmx].overflow_ex();
        overFlowCable[1][encmx] = energycmxrtmdata[encmx].overflow_ey();
        overFlowCable[2][encmx] = energycmxrtmdata[encmx].overflow_et();
        overFlowCableRes[0][encmx] = energycmxrtmdata[encmx].overflow_ex_res();
        overFlowCableRes[1][encmx] = energycmxrtmdata[encmx].overflow_ey_res();
        overFlowCableRes[2][encmx] = energycmxrtmdata[encmx].overflow_et_res();
         parityerrordetected |=energycmxrtmdata[encmx]. parityerrordetected ();
    }
    // merging overflows
    for (unsigned int i = 0; i < 3; i++ ) {
        if (( overFlowCable[i][0] == 1 ) || ( overFlowCable[i][1] == 1 )){
            overFlowSystem[i] = 1;
        }
        if (( overFlowCableRes[i][0] == 1 ) || ( overFlowCableRes[i][1] == 1 )){
            overFlowSystemRes[i] = 1;
        }
    }
    // ex, ey are ex= − crate 0 + crate 1, ey= crate 0 + crate 1
    int sysMergeEx = 0, sysMergeEy = 0;
    unsigned int sysMergeSumET = crateSum[2][0] + crateSum[2][1];
    sysMergeEx = −crateSum[0][0] + crateSum [0][1];
    sysMergeEy = crateSum[1][0]  + crateSum [1][1];
    // checking overflow
    if (sysMergeEx >= 0x4000 || sysMergeEx < −0x4000) {
        overFlowSystem[0]=1;
    }
    if (sysMergeEy >= 0x4000 || sysMergeEy < −0x4000) {
        overFlowSystem[1]=1;
    }
    if (sysMergeSumET >= 0x4000) {
```

```
            overFlowSystem[2]=1;
        }
        // [...] same for the restricted
        // checking thresholds
        std::vector < uint32_t > metthresholds         = m_thresholdmap[0];
        std::vector < uint32_t > metresthresholds      = m_thresholdmap[1];
        std::vector < uint32_t > summetthresholds      = m_thresholdmap[2];
        std::vector < uint32_t > summetresthresholds   = m_thresholdmap[3];
        std::vector < uint32_t > metsigthresholds      = m_thresholdmap[4];
        uint32_t scale      = m_thresholdmap[5][0];
        uint32_t offset     = m_thresholdmap[5][1];
        uint32_t xeMin      = m_thresholdmap[5][2];
        uint32_t xeMax      = m_thresholdmap[5][3];
        uint32_t teSqrtMin  = m_thresholdmap[5][4];
        uint32_t teSqrtMax  = m_thresholdmap[5][5];
        // automatically generate hit maps
        unsigned int methitmap=createMEThitmap(sysMergeEx, sysMergeEy, metthresholds);
        unsigned int metreshitmap=createMEThitmap(sysMergeExRes, sysMergeEyRes, metresthresholds);
        unsigned int summethitmap=createSUMEThitmap( sysMergeSumET, summetthresholds);
        unsigned int summetreshitmap=createSUMEThitmap( sysMergeSumETRes, summetresthresholds);
        // overflow overrides hitmaps
        if (overFlowSystem[0]==1 || overFlowSystem[1] ==1) {
            methitmap=0xff;
        }
        if (overFlowSystemRes[0]==1 || overFlowSystemRes[1] ==1) {
            metreshitmap=0xff;
        }
        if (overFlowSystem[2]==1) {
            summethitmap=0xff;
        }
        if (overFlowSystemRes[2]==1) {
            summetreshitmap=0xff;
        }
        // calculations for XS thresholds
        uint32_t metSQR = sysMergeEx * sysMergeEx + sysMergeEy * sysMergeEy;
        // checking special conditions
        bool c0= (metSQR < xeMin * xeMin);
        bool c1= (metSQR > xeMax * xeMax) || (overFlowSystem[0]==1 || overFlowSystem[1]==1);
        bool c2= (sysMergeSumET < teSqrtMin*teSqrtMin);
        bool c3= (sysMergeSumET >= teSqrtMax*teSqrtMax) || (overFlowSystem[2]==1);
        // checking thresholds, large numbers are calculated as floats first, then stripped to integers (no rounding )
        // the float->int precision loss does not have an influence on thresholding (done purely with integers, but using floats as data type)
        // only on the 'threshold value' (convolution of several quantities) is subject to rounding, but if the register values are the same, the simulation will be the same
        uint32_t metsigresult =0;
        for ( size_t n=0; n < 8; n++) {
            float Tisqr         = (float) metsigthresholds [n]*(float) metsigthresholds [n]/10/10;
            float scalesqr      = (float) scale*scale /1000/1000;
            float offsetqr      = (int)(float) offset * offset /1000/1000;
            float tiscales      = (int)(Tisqr * scalesqr);
            float tiscalessqr   = tiscales * tiscales ;
            float rightside     = tiscales * sysMergeSumET + tiscales * offsetqr - metSQR;
            if (( 4 * tiscalessqr * offsetqr * sysMergeSumET > rightside*rightside) || (metSQR > tiscales * sysMergeSumET + tiscales * offsetqr )) {
                metsigresult =Bitcoder:: setbit ( metsigresult , n, 1 );
            }
        }
        // checking against special conditions
```

```
    if (c0) {
        metsigresult =0;
    } else {
        if (c1) {
            metsigresult =0xff;
        } else if (c2 || c3) {
            metsigresult =0;
        }
    }
    // [...]
}
```

2.4 Other general tools and classes

2.4.1 Tools for manipulating bits

General tools for manipulating bits have been collected in `Bitcoder.h`. These include CRC calculation, merging and splitting of data words, conversion of words in to bit or strings. Functions to calculate parity bits for a list of words, functions to read/write/print multi-dimensional vectors are provided.

```
namespace Bitcoder {
 // shortcut for vector of uint32_t that form one event
 typedef std::vector < uint32_t > spymemevent;
 // shortcut for functions that split bits
 struct splitint {
  uint32_t lsb;
  uint32_t hsb;
 };
 // CRC calculation, used for topo
 uint32_t calculateCRC( const std::vector < uint32_t > &data, unsigned int length, uint32_t crcmask, unsigned int
     crclength );
 uint32_t calculateCRC( uint32_t data, unsigned int length, uint32_t crcmask, unsigned int crclength );
 // functions for sorting according to the Batcher sort algorithm
 void mergesort( unsigned int lo, unsigned int hi, unsigned int r, std::vector < std::pair < unsigned int,
     unsigned int > > &pairs );
 void splitsort ( unsigned int lo, unsigned int hi, std::vector < std::pair<unsigned int, unsigned int> > &pairs
     );
 std::vector < std::pair<unsigned int, unsigned int> > generatesortpairs ( unsigned int n );
 // code/decode twos-complement numbers
 int decode2complement( uint32_t word, unsigned int length );
 uint32_t encode2complement( int word, unsigned int legnth );
 // triming word to a certain number of bits
 uint32_t trimdata ( uint32_t word, unsigned int length );
 // convert numbers
 std::string word2bit( uint32_t word, unsigned int length );
 std::string word2hex( uint32_t word, unsigned int length );
 std::string word2word( uint32_t word, unsigned int length );
 uint32_t bit2word( std::string strn );
 uint32_t hex2word( std::string strn );
 uint32_t word2word( std::string strn );
 uint32_t string2number( std::string strn );
 std::string double2string ( double num, std::string format, std::string unit="", bool space=true, bool scale=
     false );
 std::string float2string ( float num, std::string format, std::string unit="", bool space=true, bool scale=false
     );
 std::string number2string( uint32_t word );
 std::string intnumber2string ( int word );
 std::string wordasbits( uint32_t word, unsigned int lenght=32 );
```

```
void printwordasbits( uint32_t word, unsigned int lenght=32, std::string prefix="0x" );
void printwordsasbits( unsigned int nwords, uint32_t words[], unsigned int lenght=32, std::string prefix="0x" );
// getting / setting single / multiple bits
uint32_t getbit( uint32_t word, unsigned int bit );
uint32_t getbits( uint32_t word, unsigned int startbit, unsigned int length );
uint32_t setbit( uint32_t word, unsigned int bit, uint32_t value );
uint32_t setbits( uint32_t word, unsigned int startbit, unsigned int length, uint32_t value );
// parity related functions
bool checkparity( std::vector< std::vector< uint32_t > > &words, unsigned int nwords, bool cleardataonerror, bool setparityerrorbit, unsigned int startword, unsigned int startbit, unsigned int length, unsigned int position, bool verbose );
bool checkparity( std::vector< uint32_t > &words, unsigned int nwords, bool cleardataonerror=true, bool setparityerrorbit=true, unsigned int startword=0, unsigned int startbit=0, unsigned int length=PARITYBIT, unsigned int position=PARITYBIT, bool verbose=false );
bool checkparitybit( uint32_t &word, unsigned int startbit, unsigned int length, unsigned int position, bool odd=true, bool cleardataonerror=false, bool setparityerrorbit=false, bool verbose=false );
uint32_t calculateparity( uint32_t word, unsigned int startbit, unsigned int length, bool odd=true );
uint32_t setparitybit( uint32_t word, unsigned int startbit, unsigned int length, unsigned int position, bool odd=true );
void setparity( std::vector< std::vector< uint32_t > > &words, unsigned int nwords, unsigned int startword, unsigned int startbit, unsigned int length, unsigned int position );
void setparity( std::vector<uint32_t> &words, unsigned int nwords, unsigned int startword=0, unsigned int startbit=0, unsigned int length=PARITYBIT, unsigned int position=PARITYBIT );
void setparity( uint32_t words[], unsigned int nwords, unsigned int startword=0, unsigned int startbit=0, unsigned int length=PARITYBIT, unsigned int position=PARITYBIT );
void setparity( uint32_t words[][DATADEPTH], unsigned int nwords, unsigned int startword=0, unsigned int startbit=0, unsigned int length=PARITYBIT, unsigned int position=PARITYBIT );
// merging/ splitting bits in multiple words
uint32_t mergebits( uint32_t lsbword, unsigned int lsblength, uint32_t hsbword, unsigned int hsblength );
splitint splitbits( uint32_t word, unsigned int splitbit );
uint32_t getmergebits( uint32_t lsbword, unsigned int lsbstartbit, unsigned int lsblength, uint32_t hsbword, unsigned int hsbstartbit, unsigned int hsblength );
splitint setsplitbits( uint32_t word, uint32_t lsbword, unsigned int lsbstartbit, unsigned int lsblength, uint32_t hsbword, unsigned int hsbstartbit, unsigned int hsblength );
// copy part of a vector into a new vector
std::vector< uint32_t > slicewords( std::vector< uint32_t > words, unsigned int nwords, unsigned int startword );
// clearing parts of a vector using control bits
void clearchannels( std::vector< uint32_t > &words, unsigned int nwords, unsigned int controlbits, unsigned int slicesize );
void clearchannels( std::vector< std::vector< uint32_t > > &words, unsigned int nwords, uint32_t controlbits );
// masking out bits
uint32_t applybitmask( uint32_t word, uint32_t mask );
void applybitmask( std::vector<uint32_t> &words, uint32_t mask );
// creating bit patterns
uint32_t makemask_ones( unsigned int length );
uint32_t makemask_alternate( unsigned int length, unsigned int startwith=0, unsigned int periods_zero=1, unsigned int periods_one=1 );
// copy of parts of a vector
std::vector<uint32_t> copydata( const std::vector< std::vector<uint32_t> > &sourcewords, unsigned int nword );
std::vector<uint32_t> copydata( const std::vector<uint32_t> &sourcewords, unsigned int nwords );
void copydata( const std::vector<std::vector< uint32_t > > &sourcewords, std::vector< std::vector< uint32_t > > &targetwords, unsigned int nword );
void copydata( const std::vector<std::vector< uint32_t > > &sourcewords, std::vector< std::vector< uint32_t > > &targetwords, unsigned int nwords, unsigned int startword );
// re-arrange words and split by a certain word length
std::vector<uint32_t> slicemanywords( std::vector<uint32_t> words, int wordsize, int resultwordsize, int
```

```
    startbit , int length, bool reverseinput =false, bool reverseresult =false, bool debug=false );
 int32_t slicemanywords( const std::vector < uint32_t >&words, int wordsize, int startbit , int length, bool
    reverse =false );
void slicemanywords( std::vector < uint32_t > words, std::vector < unsigned int > wordsize, std::vector <
    uint32_t > &result, std::vector < unsigned int > resultwordsize , unsigned int startbit , unsigned int length,
    unsigned int starttargetbit , bool expandtarget, bool reverseinput , bool reverseresult , bool verbose=false
    );
void slicemanywords( const std::vector < uint32_t > &words, unsigned int wordsize, std::vector <uint32_t> &
    result, const std::vector < unsigned int > &resultwordsize, unsigned int startbit , unsigned int length,
    unsigned int starttargetbit , bool expandtarget, bool reverseinput , bool reverseresult , bool verbose=false )
    ;
void slicemanywords( const std::vector < uint32_t > &words, std::vector < unsigned int > wordsize, std::vector
    <uint32_t> &result, unsigned int resultwordsize , unsigned int startbit , unsigned int length, unsigned int
    starttargetbit , bool expandtarget, bool reverseinput , bool reverseresult , bool verbose=false );
void slicemanywords( const std::vector < uint32_t > &words, unsigned int wordsize, std::vector <uint32_t> &
    result, unsigned int resultwordsize , unsigned int startbit , unsigned int length, unsigned int starttargetbit
    , bool expandtarget, bool reverseinput , bool reverseresult , bool verbose=false );
// reading/writing/printing 1D, 2D, 3D vectors
int writeVector( const std::string name, const std::vector < uint32_t > &data, const std::string contentname="
    none", unsigned int size=-1, unsigned int wordlength=8, bool append=false );
int printVector( const std::vector < uint32_t > &data, const std::string contentname="none", unsigned int size
    =-1, unsigned int wordlength=8, bool append=false );
int readVector( const std::string name, std::vector < uint32_t > & data, std::string contentname="none", bool
    checkcontent=true, int startat =-1, int length=-1 );
int writeVector( const std::string name, const std::vector < std::vector < uint32_t > > &data, const std::
    string contentname="none", unsigned int size=-1, unsigned int wordlength=8, bool append=false );
int printVector( const std::vector < std::vector < uint32_t > > &data, const std::string contentname="none",
    unsigned int size=-1, unsigned int wordlength=8, bool append=false );
int readVector( const std::string name, std::vector < std::vector < uint32_t > > & data, std::string
    contentname="none", bool checkcontent=true, int startat =-1, int length=-1 );
int writeVector( const std::string name, const std::vector < std::vector < std::vector < uint32_t > > > &data,
    const std::string contentname="none", unsigned int size=-1, unsigned int wordlength=8, bool append=false );
int printVector( const std::vector < std::vector < std::vector < uint32_t > > > &data, const std::string
    contentname="none", unsigned int size=-1, unsigned int wordlength=8, bool append=false );
int readVector( const std::string name, std::vector < std::vector < std::vector < uint32_t > > > & data, std::
    string contentname="none", bool checkcontent=true, int startat =-1, int length=-1 );
// printing events in hex
template <typename T>
 void printevent_hex( std::vector < T > event, unsigned int minsize=0, std::string delimiter =" ", size_t start
    =0, size_t end=0 )
// check size of vectors and correct, if needed
template <typename T>
 bool checkvectorsize( std::vector < T > &vec, size_t size, bool correct, bool exact, bool warning, std::string
    warningstring )
template <typename T>
 bool checkvectorsize( std:: vector < std::vector < T > > &vec, size_t size, size_t subsize, bool checksub,
    bool correct, bool correctsub, bool exact , bool warning, std::string warningstring )
template <typename T>
 bool checkvectorsize( std::vector < std:: vector < std::vector < T > > > &vec, size_t size, size_t subsize,
    size_t subsubsize, bool checksub, bool checksubsub, bool correct, bool correctsub, bool correctsubsub, bool
    exact, bool warning, std::string warningstring )
template <typename T>
 bool checkvectorsize( const std::vector < T > &vec, size_t size, bool exact , bool warning, std::string
    warningstring )
template <typename T>
 bool checkvectorsize( const std:: vector < std::vector < T > > &vec, size_t size, size_t subsize, bool
    checksub, bool exact, bool warning, std::string warningstring )
template <typename T>
 bool checkvectorsize( const std::vector < std:: vector < std::vector < T > > > &vec, size_t size, size_t
    subsize, size_t subsubsize, bool checksub, bool checksubsub, bool exact, bool warning, std::string
```

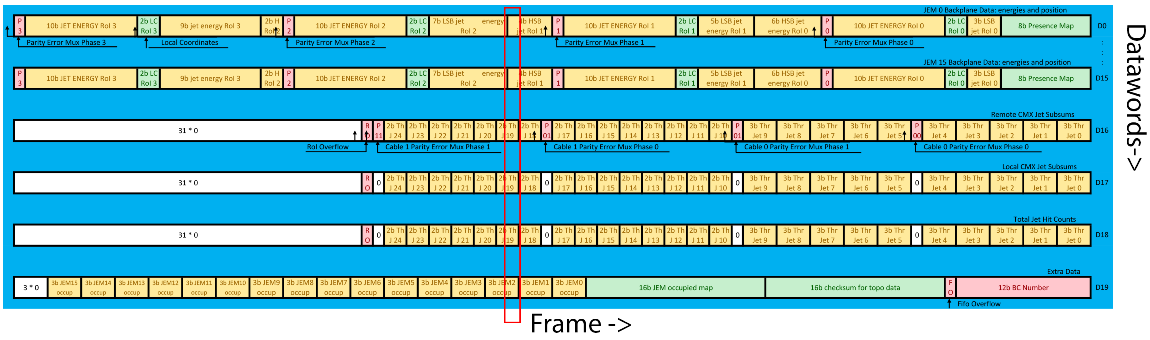


Figure 20: Scheme of the glink data format. A frame is indicated by the red box.

```
      warningstring  )
}
```

2.4.2 Class the glink format, tools handling reading/writing to data ports in the online simulation

Functions and classes directly related to the CMX itself are found in `Cmxbitcoder.h`. Functions to read and write to `IntPorts` are used in the online simulation.

An important is `Glinkdata` which encapsulates the glink data format, where the bits are "rotated" and instead of using an *n*-bit wide bus, the bus size is fixed (to a certain number of glink pins) and the event is transmitted in *n* time ticks. Each time slice is called `frame`. The data are written in to different data words (up to 20) each can be up to *n* bits (currently 96) long (`writedata()`). Finally the frames can be generated with `getframe()` and the data word can be pushed into an `IntPort`. A sketch of the glink data format is shown in Figure 20.

```
namespace Cmxbitcoder {
 class  Glinkdata  {
   public:
      //  constructor  with  default  frame  length
      Glinkdata(  int  size=128 );
      //  reading/writing  words/bits
      uint32_t  readdata(  unsigned int  dataword,  unsigned int  startbit ,  unsigned int  lenght  );
      uint32_t  readbit (  unsigned int  dataword,  unsigned int  bit  );
      void  writedata (  unsigned int  dataword,  unsigned int  startbit ,  unsigned int  lenght ,  uint32_t  value  );
      void  writebit (  unsigned int  dataword,  unsigned int  bit ,  uint32_t  value  );
      //  manipulating  single  bits
      void  anddata(  unsigned int  dataword,  unsigned int  startbit ,  unsigned int  lenght ,  uint32_t  value  );
      void  andbit (  unsigned int  dataword,  unsigned int  bit ,  uint32_t  value  );
      void  ordata (  unsigned int  dataword,  unsigned int  startbit ,  unsigned int  lenght ,  uint32_t  value  );
      void  orbit (  unsigned int  dataword,  unsigned int  bit ,  uint32_t  value  );
      void  notdata (  unsigned int  dataword,  unsigned int  startbit ,  unsigned int  lenght  );
      void  notbit (  unsigned int  dataword,  unsigned int  bit  );
      //  get  all/single  frames
      std :: vector  <uint32_t> getframes () ;
      uint32_t  getframe(  unsigned int  framenumber );
      //  directly  set  frames
      void  setframes (  const  std :: vector  <uint32_t> &frames );
      void  setframe (  unsigned int  framenumber,  uint32_t  word );
      void  clearframe (  unsigned int  framenumber );
      void  clearframes () ;
      //  parity  related  functions
      uint32_t  calculateparity  (  unsigned int  dataword,  unsigned int  startbit ,  unsigned int  length ,  bool  odd=true )
        ;
```

```
    bool checkparity ( unsigned int dataword, unsigned int startbit , unsigned int length , unsigned int position ,
     bool odd=true, bool cleardataonerror =false , bool setparityerrorbit =false ) ;
    void setparity ( unsigned int dataword, unsigned int startbit , unsigned int length , unsigned int position ,
     bool odd=true ) ;
    bool checkparities ( unsigned int framelength , unsigned int startbit , unsigned int length , unsigned int
     position , bool odd=true, bool cleardataonerror =false , bool setparityerrorbit =false ) ;
    void setparities ( unsigned int framelength , unsigned int startbit , unsigned int length , unsigned int
     position , bool odd=true ) ;
    // directly write into IntPort
    void writeWords( L1CaloSim::IntPort * DaqOutputPort, int ticks ) ;
};
// IntPort related functions
unsigned int readfromIntPort ( L1CaloSim::IntPort* intport , bool applymask=false, uint32_t mask=0 );
void readfromIntPort ( L1CaloSim::IntPort* intport , uint32_t words [], unsigned int nwords, bool applymask=false,
     uint32_t mask=0 );
void readfromIntPort ( L1CaloSim::IntPort* intport , std :: vector < uint32_t > &words, unsigned int nwords, bool
     applymask=false, uint32_t mask=0 );
void writetoIntPort ( L1CaloSim::IntPort* intport , uint32_t word ) ;
void writetoIntPort ( L1CaloSim::IntPort* intport , uint32_t words [], unsigned int nwords ) ;
void writetoIntPort ( L1CaloSim::IntPort* intport , const std :: vector < uint32_t > &words, unsigned int nwords ) ;
}
```

2.5 Diagnostics software

The `cmxLab` executable is a collection of tools that allow access and manipulation of the spy memories. There are general options that can be set as one of the first parameters:

- `--CMX0 --CMX1` actions are only taken on CMX0 or CMX1 instead on both.
- `--hexprefix` selects the prefix that will be shown before hex numbers.
- `--ignoreactive` ignore the checks if a certain spy memory is active or not.
- `--nativeformat` converts the spy memory words into data words
- `--printall` prints all events, do not replace repeated lines by `[...]`
- `--savenativeformat` when saving the spy memory content also convert the spy memory words into data words.
- `--sidebyside` shows all events of all channels and all spy memories side-by-side
- `--skipCrate` skips the initialisation of crates.
- `--skipDB` skips the database and ignore the disabled state of the CMX in the database (also ignore, if a CMX is not installed, this can lead to unpredictable results).

With the following options, the following spy memories can be accessed. An arbitrary number of spy memories can be accessed in a single command (also a single spy memory can be accessed multiple times) and they are accessed in the same order as given in the parameters. As default, the spy memories are first accessed on CMX0, then on CMX1, unless otherwise selected with the general options.

- `SOURCE` input spy memory in the source clock domain
- `SYSTEM` input spy memory in the system clock domain
- `RTMOUTPUT` output RTM spy memory of JET/ENERGY crate-type CMX

- `RTMSOURCE` input RTM spy memory of JET/ENERGY system-type CMX in the source clock domain
- `RTMSYSTEM` input RTM spy memory of JET/ENERGY system-type CMX in the system (DS1) clock domain
- `RTMSYSTEMDS2` input RTM spy memory of JET/ENERGY system-type CMX in the system (DS2) clock domain
- `RTMCPOUTPUT` output RTM spy memory of CP crate-type CMX
- `RTMCPSOURCE` input RTM spy memory of CP system-type CMX in the source clock domain
- `RTMCPSYSTEM` input RTM spy memory of CP system-type CMX in the system (DS1) clock domain
- `RTMCPSYSTEMDS2` input RTM spy memory of CP system-type CMX in the system (DS2) clock domain
- `CTP` output CTP spy memory of system-type CMX

There are options that must follow the selected spy memory. These define certain actions on the spy memory. Each action is executed in the order given (some only make sense in a certain order) and actions can be executed multiple times. The order is especially important when using the internal buffer, which e.g. needs to be filled before printing it out:

- `--clear` clears the internal buffer.
- `--copy` the current spy memory buffer is copied into a second internal buffer.
- `--load:FILENAME --load0:FILENAME --load1:FILENAME` load the content of the file `FILENAME` into the internal buffer. The variants only load, if CMX0 or CMX1 is used.
- `--match:mode` match the internal buffer with the second internal buffer. Please note that the internal buffer and the second internal buffer must be filled first. The match is indicated by a positive integer number, if the match was 100% on an `mode=1`) event-by-event, `mode=2`) spy memory word `mode=3`) bit-wise basis. The number indicates a shift by a certain number of events. If the number is negative, fractional and larger than -256 the events were only matched by the fractional part of the number and the integer part indicates the best matched (positive) shift of all events.
- `--playback` sets the mode of the spy memory into `PLAYBACK` and the current content of the spy memory is played back into the real time path. Note that the spy memory has to be put in `PLAYBACK` first (if it was in `SPY` mode) before writing the playback pattern into the spy memory.
- `--print` prints out the internal buffer. All events from a particular channel are printed. If events are identical to previous events, they are omitted and indicated by `[...]`, unless otherwise chosen by the general options.
- `--read` read the content of the spy memory into the internal buffer
- `--rotate:NUM` shift events in the internal buffer by `NUM`.
- `--save:FILENAME --save0:FILENAME --save1:FILENAME` save the internal buffer into a file with `FILENAME` filename. The variants `save0 save1` save only the content of CMX0 or CMX1

- `--spy` the spy memory is set into `SPY` mode and the content of the spy memory is filled with events from the real time path.
- `--testalive` tests if the spy memory is alive (i.e. the driving input channel is active) by setting the memory to `VERIFY` mode, writing 0xabad to all spy memory words, setting the memory to `SPY` mode and comparing the content of the spy memory with 0xabad. The driving input channel is alive, if the content is different. This test can be done in addition to checking the event counters of each input channel (see `p_clockDetectCounter`).
- `--testparity` tests the parities of the events in the internal buffer.
- `--testreliable:NUM` tests if the spy memory is reliable by repeatedly reading the memory `NUM` times and comparing the content with each other.
- `--verify` the spy memory is set into `VERIFY` mode and the content of the spy memory is compared with the real time path.
- `--write` write the internal buffer into the selected spy memory.

2.6 Reconfiguring the CMX firmware with new firmware from CF card

The CMX BF firmware is loaded at power-up from the CF card. The BSPT firmware checks the geographical address that is coded in the crate and translates this into a CF slot number or CMX flavour (see Table 7). The `geoAddr` value is used in the database, bit 0 encodes left=0, right=1 position of the CMX in the crate, bits 1-3 encode the crate number, the HW bits have the crate bits inverted and the flavour denotes the CMX flavour. In addition there are two special slots that can contain test versions of the CMX firmware. The CMX geographical address can be overwritten in the `p_moduleControl`register (relative address `0x0004`) by setting the first 4 bits to the HW bits, then the flavour/CF slot number will appear in bits 8-11. If a module reset is issued (`p_moduleResets`, relative address `0x0006` via bit 3) the SystemAce reloads the firmware using the slot indicated in the control register.

To load the firmware via the SystemAce the program `cmxFlashWriter` can be used. This will take a disk image formatted as FAT12 file system with block size of 512b. The disk image has to be created with the following directory structure: In the main directory there is `xilinx.sys` file that defines the directory and the subdirectories where the different CMX firmware configurations are located. In each subdirectory the different CMX flavour configurations can be found.

The `cmxFlashWriter` program is used as follows:

```
cmxFlashWriter CMXNO STRING FILENAME START END 1
```

`CMXNO` is the CMX number, `STRING` can be `0` for reading and `1` for writing a file image to the CF, `FILENAME` is filename of the disk image and `START` and `END` indicate the start and end block number. The `1` at the end starts the reading/writing immediately.

The SystemAce is at the moment very unstable, so that the CMX CF should only be written via the SystemAce when no other activities are ongoing. Also interrupting the read/writing process is not advisable, it can leave the CF and the SystemAce in a bad state.

2.6.1 Example commands

```
cmxFlashWriter 0 1 CF.img 0 327680 1
echo 3F | vme edit 700004
echo 8 | vme edit 700006
```

This will copy the file `CF.img` (with 160MB size) to CMX 0, then force the CMX to load the CP crate-type flavour (2) firmware and trigger reloading.

`geoAddr`	HW bits	CMX type	crate	flavour/CF slot
1	`1111` F	EM	0	2
3	`1101` D	EM	1	2
5	`1011` B	EM	2	2
7	`1001` 9	EM	3	3
0	`1110` E	TAU	0	2
2	`1100` C	TAU	1	2
4	`1010` A	TAU	2	2
6	`1000` 8	TAU	3	3
8	`0110` 6	ENERGY	4	4
A	`0100` 4	ENERGY	5	5
9	`0111` 7	JET	4	6
B	`0101` 5	JET	5	7
SP	`0000(1)` 0/1	SP	SP	0
SP	`0010(1)` 2/3	SP	SP	1

Table 7: Overview of different CMX location and flavour coding. "SP" denotes special test firmware slots.

3 CMX logic test

The following section describes the logic tests that have been performed with the CMX using the online simulation.

3.1 Test software

For the test the standalone program `cmxSimLab` has been used. The source code resides in the `cmxTest` package.

The program is able to test all CMX flavours with a set of thresholds and test vectors. For one test a multiple of test sequences are tested with 256 events each. It can also read the current configuration from the CMX and can also simulate and predict the output of the current content of the input spy memories.

`cmxSimLab` can be started with general options:

- `--CMX0` and `--CMX1` selects CMX0 or CMX1 for tests.
- `--bugfix` is used, if a current bug in the firmware needs to be bypassed. Currently there are no known firmware bugs.
- `--nativeformat`
- `--printall`
- `--sidebyside`
- `--skipCrate`
- `--skipDB` ignores the database state of CMX modules, so that the tests can be performed on disabled CMX.

- `--slice`

The order of the following parameters needs to be preserved. The sequences (or single steps) can be repeated, if it makes sense. Each step has several options, that are also described below:

- `CONFIG` configures the behaviour of the simulation with the following flags.
  - `--force` uses the force mode in the simulation, forcing to set all thresholds to maximum, when a parity error is found.
  - `--quiet` uses the quiet mode in the simulation, zeroing all the datawords when a parity error is found.
  - `--direct` uses directly the simulation objects instead of converting the objects into data words and converting them back.
  - `--events:NUM` creates and simulate `NUM` events. If the number of events are not a multiple of 256, the last event is repeated until a multiple of 256 has been reached.
  - `--latency:A:B` compensates the latency for the local (`A`) and remote events (`B`), so that the correct events are added in the CMX when playing back. The events after the simulation are rotated by the number of events given and then written into the playback memory.
  - `--configure` configures the hardware to match the simulation settings.
- `MODE` one of the following modes must be chosen for the simulation. The mode must match the type of firmware that is loaded in the CMX. A warning will be given, if there is a mismatch between the simulation mode and the CMX firmware type.
  - `--jet_system`
  - `--energy_system`
  - `--cp_system`
  - `--jet_crate`
  - `--energy_crate`
  - `--cp_crate:CRATENR` for the CP crate-type simulation the crate number must be given.
- `THRESHOLD`
  - `--generate:MODE:SUBMODE` generates a threshold map which is given by `MODE` and `SUBMODE`. A list of threshold patterns is given in Section 3.1.1.
  - `--load:FILENAME` a threshold map is loaded from a file with the name `FILENAME`.
  - `--save:FILENAME` a threshold map is written to a file with the name `FILENAME`. This only makes sense after a threshold map has been loaded either from a file, from the CMX itself or from the database.
  - `--read:SUBMODE` read a threshold map from the CMX. For energy thresholds `SUBMODE` is used to determine the scale ($\times 1000$) for the calculation of the parameters. It is the same value, that is used in the menu. It has to be given, otherwise the thresholds and parameters cannot be calculated unambiguously.
  - `--get` get the threshold map from the database in the format that is used by the simulation or by the `configure_*` functions that configure the CMX thresholds.
  - `--write` write the threshold map into the CMX.

  - `--print` print the threshold map to screen.

- `GENERATE` this generates and simulates events according to the mode. The default behaviour is to generate the events at the input to the backplane and then simulate the whole chain down to the CTP output.

  - `--mode:MODE:SUBMODE` generates events given by `MODE` and `SUBMODE`. A list of event patterns is given in Section 3.1.2.
  - `--verbose` sets the verbosity of the event generation and simulation.
  - `--load:FILENAME` load events at the input from file `FILENAME`.
  - `--read` read the input data from the backplane of the CMX directly. Please note that the spy memory is not set to `SPY` mode by this function.

- `SAVE` this saves all generated and simulated events.

  - `--prefix:PREFIX` this uses `PREFIX` as prefix to all files, the name is completed with the type of events (`CP`,`ENERGY`,`JET` and `SYSTEM`,`CTP`,`RTM.OUT`,`RTM.IN`,`RTM`, please note that only for the CP type CMX there are different files for `RTM.IN` and `RTM.OUT`).

- `TEST` after the simulation the simulated events can be tested against the hardware. This usually consists of reading back the output spy memories and comparing the events from the CMX with the simulated events. The order of the events are always matched to the order of the events from the hardware by shifting the events to their best match (due to the latency when the events are processed in the system).

  - `--wait:WAIT` this will wait `WAIT` seconds after the playback events have been loaded into the spy memory.
  - `--verify` in addition to the pattern comparison the simulated events are loaded into the spy memory for bit-wise real-time verification.
  - `--save` the events that are read from the CMX are saved into files with `.HW.` in the filename.
  - `--prefix:PREFIX` sets another prefix `PREFIX` for the events from the hardware.
  - `--automatch` the events that are saved to file are matched to the order of the simulated output events (basically reverting the shift of events due to the latency).
  - `--showinputs` prints the inputs.
  - `--showoutputs` prints the outputs.
  - `--showhardware` prints the events from the hardware.
  - `--nodiff` do not print the differences between the events from the simulation and from the hardware.
  - `--showall` is a short cut for `--showinputs`, `--showoutputs` and `--showhardware`

3.1.1 Threshold modes

Mode `0` is available for all thresholds which are all random thresholds. For the energy thresholds the *min* and *max* values are randomized as well, but they follow the order that *max* > *min* value. For the JET and CP thresholds there are modes `1`-`5` which are randomized energy and jetsize/isolation values, but `1` has the same energy for all thresholds, `2` same jetsize/isolation for all thresholds, `3` all *eta*-slices have the same energy and jetsize/isolation value, `4` same jetsize/isolation per threshold and `5` same energy per threshold (it is implied that the other values are random).

3.1.2 Event generation modes

It has to be noted, that some generation modes are based on randomly generated TOBs or energy values and that depending on the settings, the boundary conditions cannot be easily met. In some cases this will result in an endless loop. There are only a few measures in place that try to avoid this by restarting the generation of an event, but it is not guaranteed that the event generation will run into a corner where the event generation is impossible. There are some test modes which are not mentioned here, e.g. mode `0` is in general one random event. The modes described here try to target specific tests. For the purpose of checking the timing (unless otherwise stated) every second event is either just random, empty or tries to test an alternative test condition. Further, the patterns cycle through the input modules (note that for the CP CMX, module 0 and 15 are "cycled" as well, but ignored in the subsequent simulation) and input crates, although some modes (see below) choose to loop over thresholds instead of the module or instead of walking patterns, choose a random input module or are even totally random (basically the event number is a running number from which nest loops of running numbers over channels, crates etc. are generated).

- JET CMX: TOBs are generated either randomly or walking through each input module in one crate, then through all the crates. Every second event fulfils certain conditions, the other event does not.
  - `0`: one random TOB anywhere in every event.
  - `5000`: one random TOB in each crate, only for the first event.
  - `10000:SUBMODE`: generate between 5-8 TOBs to provoke overflows in every second event (the other event is filled with a random TOB to check timing, adding `10` to the mode suppresses this, the event will not contain any TOBs), modes `10001` – `10003` allow to modify the lower/upper/both limits via the `SUBMODE`, resp., `10004` generates up to `SUBMODE` TOBs, `10005` generates exactly `SUBMODE` TOBs. The pattern cycles through the crates and JEMs, local positions and energies are random.
  - `20000`: random TOBs will be generated until a particular threshold is at the maximum count allowed by the bit length (i.e. 7 or 3 for 3bit and 2bit thresholds, respectively) for even-numbered events. For odd-numbered events a random number of TOBs (up to 4) must pass the threshold in addition (the transmitted count is still the maximum). After every second event the next threshold is tested. Events with backplane TOB overflows will be rejected. There can be TOBs in the event that do not pass the threshold.
  - `20001`: same as `20000`, but it will create TOBs randomly in a particular crate, every 32nd event will advance to the next crate. Also only TOBs are generated that pass the threshold under test.
  - `30000`: same as `20000`, but local sums must not be more than the maximum counts, but the total counts must the at the maximum (for even events) or at the maximum plus a random number of TOBs (up to 4).
  - `30001`: same as `20001` and `30000`.
- CP CMX: TOBs are generated either randomly or walking through each input module in one crate, then through all the crates. Every second event fulfils certain conditions, the other event does not.
  - `0`: one random TOB anywhere in every event.
  - `5000`: one random TOB in each crate, only for the first event.

- `10000:SUBMODE`: generate between 6-16 TOBs to provoke overflows in every second event (the other event is filled with a random TOB to check timing adding `10` to the mode suppresses this, the event will not contain any TOBs), modes `10001` – `10003` allow to modify the lower/upper/both limits via the `SUBMODE`, resp., `10004` generates up to `SUBMODE` TOBs, `10005` generates exactly `SUBMODE` TOBs. The pattern cycles through the crates and CPMs (also 0 and 15 are cycled through, but this does not create any actual TOBs), local positions and energies are random.
- `20000`: random TOBs will be generated until a particular threshold is at the maximum count allowed by the bit length (i.e. 7 or 3 for 3-bit and 2-bit thresholds, respectively) for even-numbered events. For odd-numbered events a random number of TOBs (up to 4) must pass the threshold in addition (the transmitted count is still the maximum). After every second event the next threshold is tested. Events with backplane TOB overflows will be rejected. There can be TOBs in the event that do not pass the threshold.
- `20001`: same as `20000`, but it will create TOBs randomly in a particular crate, every 32nd event will advance to the next crate. Also only TOBs are generated that pass the threshold under test.
- `30000`: same as `20000`, but local sums must not be more than the maximum counts, but the total counts must the at the maximum (for even events) or at the maximum plus a random number of TOBs (up to 4).
- `30001`: same as `20001` and `30000`.

• ENERGY CMX: Energy values on the backplane are generated that appear to come from a JEM input module. Generally, in each iteration a random value is generated for each component (E_x, E_y, E_T) that do not have to be physical. The patterns have every second event filled with a payload that fulfils certain conditions, while the other event does not. Then the patterns usually walk through all the JEM input modules in one quadrant, then through all the quadrants. `SUBMODE` is used for limiting the random value to $0x4000/2^{\mathrm{SUBMODE}}$.

- `0`: one random energy entry anywhere in every event.
- `5000`: one random energy entry in each crate, only for the first event.
- `10000:SUBMODE`: generates events with local overflows in E_x by randomly generating energy entries in a random JEM in a fixed quadrant in every second event. The other event does not necessarily have an overflow. The other energy components are set to a random value. Every 16-th event another quadrant is used. Modes `10001` and `10002` test E_y and E_T respectively. If `10` is added to the mode, the second event is empty.
- `20000:SUBMODE`: same as `10000`, but generates events without overflows in E_x (`20001` and `20002` are testing E_y and E_T respectively). There is a random number of JEMs (up to 7) that need to have energy entries (if a JEM is selected twice, the energy values are added).
- `20010:SUBMODE`: generates events without overflows in E_x (`20011` and `20012` are testing E_y and E_T respectively). The energy entries are generated in a particular JEM. Every second event the pattern advances to the next JEM. The E_y and E_T values (if another quantity is tested, the other remaining components are calculated from the randomized value) are calculated according to $E_T = \sqrt{E_x^2 + E_y^2}$.
- `30000:SUBMODE`: random energy values are generated within the detector acceptance, E_T is not allowed to overflow locally, but globally. Modes `30001` and `30002` test E_x and E_y. Every second event does not need to overflow globally. If `10` is added, every second event is empty.

– `40000` test in every second event the conditions c_0 to c_3 for the energy thresholds using the energy parameters. The events cycle through each condition requiring the condition to be fulfilled or not, leading to a periodicity of 16 events (including the fact that every second event is either random or, if `40010` is used, empty.).

3.2 Test program

An extensive test program has been proposed that uses the different generation modes of the `cmxSimLab` program. The goal is to provide a thorough test of the CMX logic and reliability. For the different flavours of the CMX there are slightly different goals. There are always tests with random input data and test that provoke overflows. Some tests were only done on the basis of single events, due to the limitation of the setup. These tests are mentioned specifically at the end of each list.

3.2.1 JET and CP CMX

For the JET flavour CMX beyond random TOBs there are tests that specifically probe the behaviour of the crate-type or system-type CMX. The patterns generally simulate the whole L1Calo system, so that many patterns that test crate-type CMX behaviour can also be used to test a certain system-type CMX behaviour.

- Crate-type CMX
  - are all thresholds are saturated, i.e. at the maximum number on the RTM output if there are more than 4 (5) TOBs on any of the inputs (all 16 (14) inputs are tested) for the JET (CP) CMX. Test modes `10000` – `10015` can be used for this.
  - if there are not more than 4 (5) TOBs anywhere, but the total count of TOBs over given threshold is equal or more than maximum permitted by the bit range, the threshold counts max out at the maximum value. All threshold slots are tested. For this purpose test modes `20000` and `20001` can be used.
- System-type CMX
  - if the RTM input count for a given threshold is saturated, the CTP output is saturated (tested for all thresholds). The saturation can be local backplane overflow, local sums exceed the maximum count given by the bit range or neither of the two previously mentioned conditions are present. The latter is the case if the saturation is already present when the threshold count is at its maximum. When a system-type CMX is tested using test modes `10000` – `10015`, `20000` and `20001`, these conditions are tested as well.
  - if the RTM sum is not saturated and the local sum is also below maximum, but the global sum exceeds maximum count permitted by the bit range, the CTP output for a given threshold maxes out at the maximum value. All thresholds are tested. Test modes `30000` and `30001` can be used for this.
- Test with special tests or limited setup
  - if there is a parity error on the backplane are all counts saturated on the RTM output in case of the `force` setting, in case of `quiet` setting is the input from the module in question disregarded. This was only tested in a special run, where the fine delays of the backplane inputs were set to the maximum value, so that parity errors are highly probable. It was confirmed that in case of `force` setting all threshold counts are at maximum, in case of `quiet` the threshold counts are zero.

- if there is a parity error on the RTM input, all the thresholds are saturated on the CTP output while the `force` flag is on, if the `quiet` flag is on the input from the module in question is ignored (but the error is still counted, flagged and reported in the readout for this event). This has been tested with a dedicated pattern played back at the output of the crate-type CMX where one event has the wrong parity. To check the timing the same pattern was played back with the correct parity and the position of the event in the CTP input spy memory is compared.

3.2.2 Energy CMX

- Crate-type CMX
  - for each of E_x, E_y, ET the overflows are signalled correctly on the RTM output if the local sum from the same quadrant overflows, if the E_{T} in the same quadrant does not overflow, but the added sum overflows. Test modes `10000` – `10012` are used for this test.
  - there are no overflow bits set, when the sum is below the overflow value. Test modes `20000` – `20012` are used for this test.
- System-type CMX
  - if the remote sums overflow, all the TE and XE threshold hits are set and the XS behave according to the conditions set by the parameters. Running test modes `10000` – `10012` implicitly tests this behaviour. Test mode `40000` probes each condition set by the parameters, however explicit overflow cannot be requested for certain conditions.
  - if the local sums overflow, all the TE and XE threshold hits are set and the XS behave according to the conditions set by the parameters. Test mode `40000` probes each condition set by the parameters, however explicit overflow cannot be requested for certain conditions. Test mode `30000` also exclusively probes global overflow.
- Test with special tests or limited setup
  - if there is a parity error on the backplane all sums sent on RTM are saturated and overflows set in the case of 'force' condition, in the case of 'quiet' condition the input from the module in question is suppressed (but the error is still counted, flagged and reported in the readout for this event). This was only tested in a special run, where the fine delays of the backplane inputs were set to the maximum value, so that parity errors are highly probable. It was confirmed that in case of `force` setting all threshold counts are at maximum, in case of `quiet` the threshold counts are zero.
  - if there is a parity error on the RTM input all threshold bits are set on the CTP output in case of `force` condition, in the case of `quiet` condition the input from the connector in question is suppressed (but error still counted and reported on readout for this event). This has been tested with a dedicated pattern played back at the output of the crate-type CMX where one event has the wrong parity. To check the timing the same pattern was played back with the correct parity and the position of the event in the CTP input spy memory is compared.

3.2.3 Example commands

```
cmxLab --CMX1 SYSTEM --spy --read --print RTMOUTPUT --spy --read --print
cmxLab --CMX0 SYSTEM --playback --load:test.event --write --read --print
cmxSimLab --CMX1 --randomseed:0 MODE --energy_system CONFIG --latency:0:1 THRESHOLD --read:1250 --
    save --print GENERATE --mode:10012 SAVE TEST --automatch --save --verify --showall
```

The first command will set the SYSTEM spy memory of CMX1 into `SPY` mode, read from the spy memory and print out the content to the screen, then the same is done for the RTM output spy memory. The second command will set the SYSTEM spy memory of CMX0 into `PLAYBACK` mode, then a spy memory content is loaded from test.event and written into the spy memory, then read back from the spy memory and finally printed to screen. The third command will start the simulation tests for CMX1, an energy system-type CMX with the latencies 0 and 1 for the local and remote events, respectively. The thresholds are taken from the CMX itself with a scale of 1250. The thresholds are saved after the simulation and printed to screen. The generation of mode `10012` is requested, the events are all saved after the simulation. The events are tested against the hardware. The output of the hardware is saved and the expect pattern is automatically matched to the output events, so that latency effects are compensated. All input, output and expected patterns are printed to screen.

3.3 Calibration software

For the timing calibration, modes are included in the `cmxLab` program. The result is saved as root and PDF files for inspection.

3.3.1 Fine delay

The fine delay scan will change the programmable delays of the backplane lines (for each module there are 24 data and one forwarded clock line, in total 400 lines, IODELAY delay circuit) and scan for the optimal delay settings for both data and clock line for stably latching the data into the system domain. For the fine delay scan

```
cmxLab --CMXx FineDelayScan
```

can be used, where `x` is the CMX that should be tested. It is suggested to run the L1Calo partition with playback pattern from the JEMs/CPMs or PPMs. A pattern that probes as many bits (i.e. as many bits of a data word should change its state) is preferred. The following options are allowed:

- `--fromfile:FILE` load scanned data from file `FILE`.
- `--cpm` scan over CP CMX (omits module 0 and 15 in the analysis).
- `--jem` scan over JET CMX.
- `--prefix:PREFIX` change prefix of saved files to `PREFIX`.
- `--same` all bits in each channel is set to the same delay value.
- `--analyse` analyse the scanned data and determine the optimal delay values.
- `--debug` debug output.
- `--createrootfile` create root file with the raw scan data and the result of the scan.
- `--createpdf` create a PDF with the results.
- `--setvalues` set the optimal values direct in the registers.
- `--waittime:WAITTIME` Sets the delay in μs between the scanning steps.
- `--from:FROM` choose an alternative start value for the scan.
- `--to:TO` choose an alternative stop value for the scan.

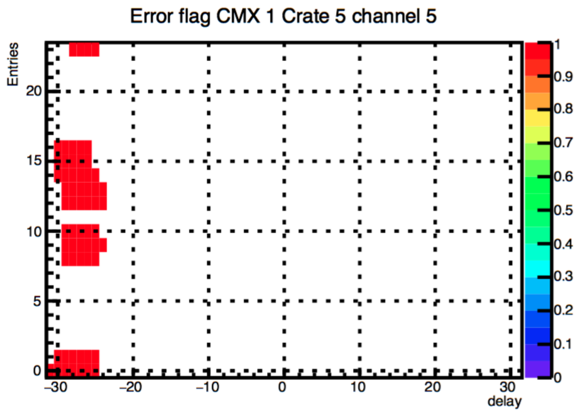


Figure 21: Fine delay scan for one channel. The x-axis indicate the total delay setting (data lines: negative delays, clock line: positive delays), the y axis is the data bit in that channel. A red entry means that there have been errors. The error free region is assumed to be the largest region error free region (for all channel) between the delay settings with errors.

For the scan the delays of the data lines (negative delays) and of the clock line (positive delays) are set such that totals delays (note, that a delay on the clock line and a delay on a data line compensate, i.e. total delay = clock delay plus negative individual clock delay) from from -30 to 30 taps (=steps of 78 ps) is achieved. Figures 21 to 25 show example plots for a single channel. The result of the scan is a set of delays for each backplane data and clock line for each channel which results in a total delay that is in the centre of the error free window. In addition the result PDF shows the region with errors (using the bit-wise error flags of the spy memory verification functionality), the error free counters versus the delay setting, the parity error count per channel versus the delay, the scan time, the number of events tested, the delay settings with the boundaries of the error free region, the relative delay settings with respect to the boundaries and the overlay of the regions with errors (page 1) and the optimal delay settings (page 6). For bits where no information can be obtained, the delay is set to 0.

3.3.2 DS1 scan

The Deskew1 (DS1) scan moves the delay of DS1 clock provided by the TTCDec card. This is the main system clock. The scan moves the DS1 setting from $K = 0$ to $K = 239$ which corresponds roughly to 0.1 ns delay per K value (25 ns in 240 steps). Measurements have shown that there can be an uncertainty on the DS1 versus DS2 setting and the true delay between the two delays up to 2 ns. The scan shows the K values where the playback pattern is received in correct order and without errors. The position where the transition from the error region to the error free region happens is an indication for the position of forwarded clock. The latency between the reception of the data and the system clock can be small, but as a safety value 2.5 ns from the “latest” channel have been chosen for the DS1 settings (25 K values after the transition value). Figure 27 illustrates the result of a DS1 scan.

For the DS1 delay scan

```
cmxLab --CMXx DS1scan
```

can be used, where x is the CMX that should be tested.

It is suggested to run the L1Calo partition with playback pattern from the JEMs/CPMs or PPMs. A playback pattern with one TOB is sufficient for this test. Ideally an energy ramp with one JET/CP TOB or ramping energy is used.

The following options are allowed:

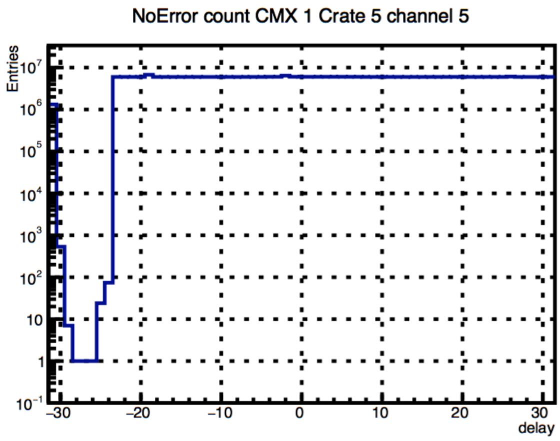


Figure 22: Fine delay scan for one channel. The x-axis indicate the delay setting, the y axis shows the number of ticks where no errors occurred. The number of ticks max out at maximum number of ticks that were counted during a scan step, which indicates that no errors have occurred.

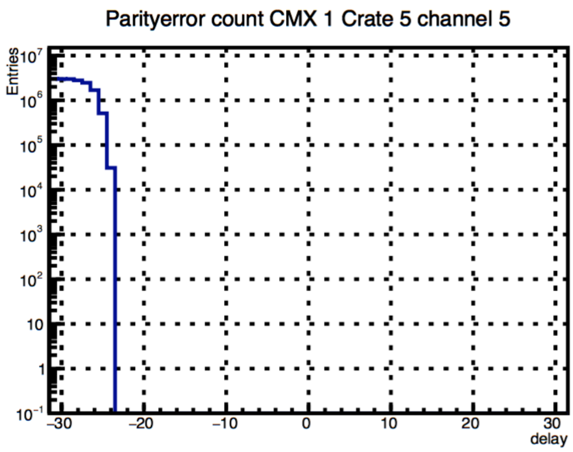


Figure 23: Fine delay scan for one channel. The x-axis indicate the delay setting, the y axis shows the number of parity errors.

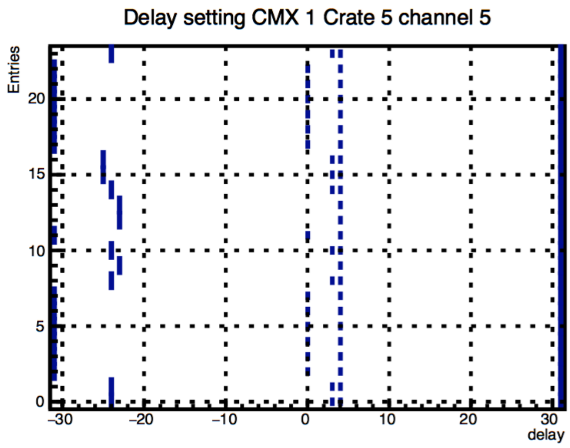


Figure 24: Fine delay scan for one channel. The x-axis indicate the delay setting, the y axis is the data bit in that channel. The blue bars indicate the edges of the transition between good and bad region. The dashed blue bars (more to the centre) indicate the clock (long line over all channels) and the resulting data delay setting (=clock delay plus negative individual clock delay).

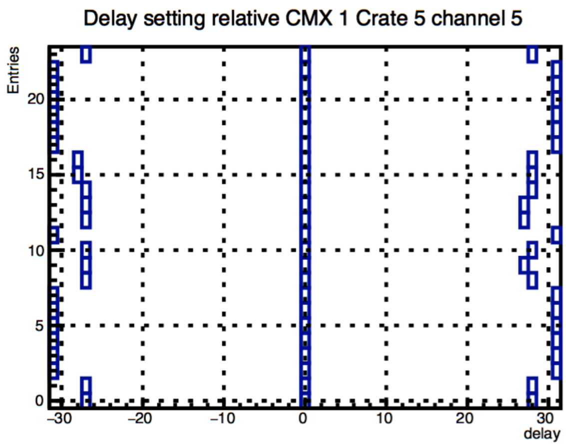


Figure 25: Fine delay scan for one channel. The x-axis indicate the delay setting, the y axis is the data bit in that channel. The blue bars indicate the edges of the transition between good and bad region relative to the centre at zero.

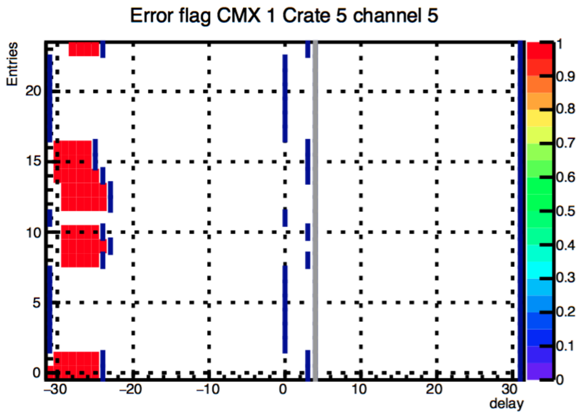


Figure 26: Fine delay scan for one channel. The x-axis indicate the delay setting, the y axis is the data bit in that channel. This is an overlay of Figures 21 and 24.

- `--cpm` scan over CP CMX (omits module 0 and 15 in the analysis).
- `--jem` scan over JET CMX.
- `--prefix:PREFIX` sets the prefix for the output files.
- `--analyse` analyse the data and find the best DS1 setting.
- `--debug` debug output.
- `--step:STEP` step in which the DS1 value is scanned.
- `--margin:MARGIN` sets the margin between the error region and the DS1 setting.
- `--createrootfile` create a root file with the analysis result.
- `--createpdf` create a PDF file with the analysis result.
- `--setvalues` sets the optimal DS1 value in the CMX.
- `--seconds:SEC` set the delay between the scan steps.
- `--start:START` sets the start value for the scan.
- `--stop:STOP` sets the end value for the scan.

3.3.3 DS2 scan

The DS2 scan moves the DS2 setting from $K = 0$ to $K = 239$ which corresponds roughly to 0.1 ns delay per unit of K. This should be done with the DS1 setting found in the DS1 scan. Measurements have shown that there can be an uncertainty on the DS1 versus DS2 setting and the true delay between the two delays up to 2 ns. The scan shows the K values where the data at the RTM DS2 to RTM system memory is received in correct order and without errors. The position where the transition from the error region to the error free region happens is an indication for the position of RTM data. The latency between the reception of the data from the RTM and the system clock should be at least 7 ns plus a safety value of 2.5 ns. Hence, the DS2 value should be set 9.5 ns before the DS1 (95 K values smaller than DS1 value).

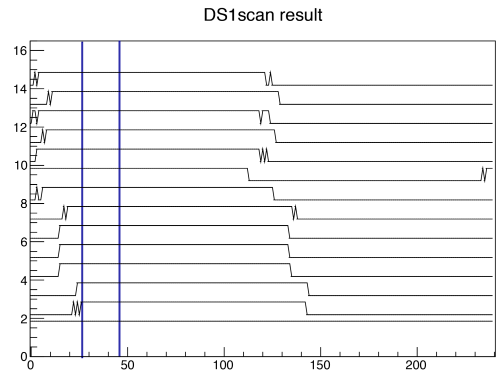


Figure 27: Result of a DS1 scan. The *x*-axis indicate the K delay setting, the *y* axis indicate the channel number. A high means no error. The blue lines indicate the transition value and the best value for DS1.

The DS2 scan is just a way to confirm that the fixed setting of DS2 falls into the good timing region. If this would not be the case a manual adjustment of DS1 and DS2 can be made at the cost of the overall latency of the system. Figure 28 illustrates the result of a DS2 scan. For the DS2 (deskew-2) delay scan

```
cmxLab --CMXx DS1scan
```

can be used, where `x` is the CMX that should be tested.

It is suggested to run the L1Calo partition with playback pattern from the JEMs/CPMs or PPMs. A playback pattern with one TOB is sufficient for this test. Ideally an energy ramp with one JET/CP TOB or ramping energy is used.

The following options are allowed:

- `--debug` prints out debug output.
- `--step:STEP` step in which the DS2 value is scanned.
- `--cpm` scan over CP CMX (omits module 0 and 15 in the analysis).
- `--jem` scan over JET CMX.
- `--start:START` sets the start value for the scan.
- `--stop:STOP` sets the end value for the scan.
- `--seconds:SEC` set the delay between the scan steps.
- `--prefix:PREFIX` sets the prefix for the output files.

3.3.4 Pipeline delay

The pipeline delay can be set using dedicated playback pattern. Ideally a playback pattern with L1Calo from the PPMs, JEMs or CPMs with one single filled event is sufficient. The CTP output should also only contain one filled event, if the pipeline delay is set correctly, otherwise zhe pipeline delay should be adjusted until only one output event is seen at the CTP output.

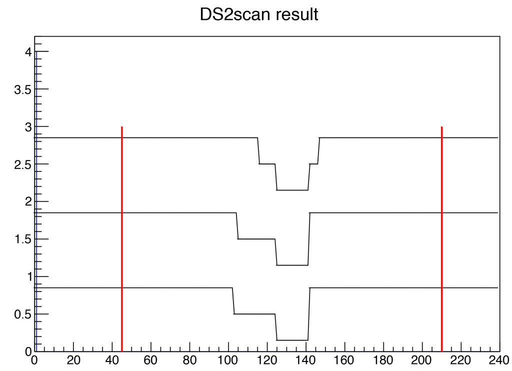


Figure 28: Result of a DS1 scan. The x-axis indicate the K delay setting, the y axis indicate the channel number. A high means no error. The red lines indicate the value for DS1 and the best value for DS2.

3.3.5 Example commands

```
cmxLab --CMX0 DS1scan --cpm --analyse --createrootfile --createpdf --prefix:DS1scan.CMX0
cmxLab --CMX1 DS2scan --analyse --createrootfile --createpdf --start:20 --stop:210 --step:1 --jem --prefix:
      DS2scan.CMX1
```

The first command will perform a DS1 scan on CMX0 (which is a CP type CMX), analyse the data and writes out root and PDF files with the results. The second command will perform a DS2 scan on CMX1 (which is a JET type CMX), analyse the data and writes out root and PDF files with the results. The DS2 scan is limited to a range of 20 and 210.

4 Open issues

- At the time of writing this document the results are not saved to the database, although the folders have been created.
- Playback from the `TDAQGUI` is not possible, yet.
- Full configuration of `cmxSimLab` by the state of the CMX (at the moment missing: force/quiet flags from hardware).

References

[1] http://www.pa.msu.edu/hep/atlas/l1calo/cmx/.

[2] http://www.pa.msu.edu/hep/atlas/l1calo/cmx/specification/4_production_design_review/.

[3] http://www.pa.msu.edu/hep/atlas/l1calo/cmx/firmware/general/.

[4] http://www.pa.msu.edu/hep/atlas/l1calo/cmx/hardware/.

[5] http://www.pa.msu.edu/hep/atlas/l1calo/cmx/specification/4_production_design_review/.

[6] https://atlas-l1calo.web.cern.ch/atlas-l1calo/html/orgweb/Modules/Modules.html.

[7] https://atlas-l1calo.web.cern.ch/atlas-l1calo/html/orgweb/Modules/CMM/CMM_V1_8.pdf.

[8] https://atlas-l1calo.web.cern.ch/atlas-l1calo/html/orgweb/Modules/CMM/MissingEnergySignificance_2012.pdf.

[9] https://atlas-l1calo.web.cern.ch/atlas-l1calo/html/orgweb/Modules/CPM/CPM_Specification_2_03.pdf.

[10] https://atlas-l1calo.web.cern.ch/atlas-l1calo/html/orgweb/Modules/JEM/JEMspec12d.pdf.

[11] https://atlas-l1calo.web.cern.ch/atlas-l1calo/html/orgweb/Modules/PPr/PPMod_Wrup.pdf.

[12] https://atlas-l1calo.web.cern.ch/atlas-l1calo/html/orgweb/Modules/ROD/ROD-spec-version1_2_2.pdf.

[13] http://atlas-l1calo.web.cern.ch/atlas-l1calo/build/nightly/logfiles/.

[14] https://twiki.cern.ch/twiki/bin/view/Atlas/LevelOneCaloDatabases.

[15] https://twiki.cern.ch/twiki/bin/Atlas/LevelOneCaloOnlineNotes/.

[16] https://atlas-l1calo.web.cern.ch/atlas-l1calo/html/orgweb/Tin/Reduced_vme_spec_v1_2.pdf.

[17] http://ttc.web.cern.ch/TTC/TTCrx_manual3.10.pdf.

[18] http://www.pa.msu.edu/hep/atlas/l1calo/cmx/firmware/bspt_fpga_fw/BSPT_FPGA_FW_v5.4_20141212/BSPT_FPGA_FW_v5.4_20141212.txt.

[19] https://atlasops.cern.ch/oncall/l1calo/.

[20] http://www.pa.msu.edu/hep/atlas/l1calo/cmx/firmware/data_formats/CMX_topo_data_format_spec.pdf.