The ATLAS Level-1 Trigger Offline Simulation: Overview, Core Parts, and Use

T. Schörner-Sadenius

Abstract

This note gives an overview of the offline simulation software for the ATLAS LVL1 trigger and describes core parts of it, namely the LVL1 configuration, the simulations of the CTP and the RoIB, the RDO definition, and the byte stream conversion. In addition, the installation and use of the LVL1 simulation are explained.

Document History

Date	Version	Author	Comment
12 May 2003	v1.0	TSS	First version.
22 July 2003	v1.1	TSS	Version submitted for approval.

Contents

List of Figures

1 Introduction

The ATLAS level 1 (LVL1) trigger [1] is a complex system built from various subsystems, namely the calorimeter trigger, the muon trigger and the central trigger processor (CTP). The calorimeter and muon triggers consist of a multitude of different devices themselves; in the context of the simulation the division of the muon trigger in the RPC barrel part, the TGC endcap part and the muon-to-CTP interface (MuCTPI) is especially important since these three components are simulated by different groups. Figure 1 gives a rough overview of the ATLAS LVL1 trigger. The trigger, timing and control (TTC) part will be neglected in this document since it has no counterpart in the offline simulation world. In contrast to this the Region-of-Interest builder (RoIB), which according to the ATLAS product break-down structure forms part of the high-level trigger (HLT), is treated within the environment of the LVL1 simulation since its task basically is to combine all partial results of the various LVL1 components to a global LVL1 result; the simulation of the RoIB therefore requires knowledge of the LVL1 simulation, but not of the HLT. Figure 2 shows a more detailed picture of the LVL1 trigger system and some of its connections to other ATLAS systems.

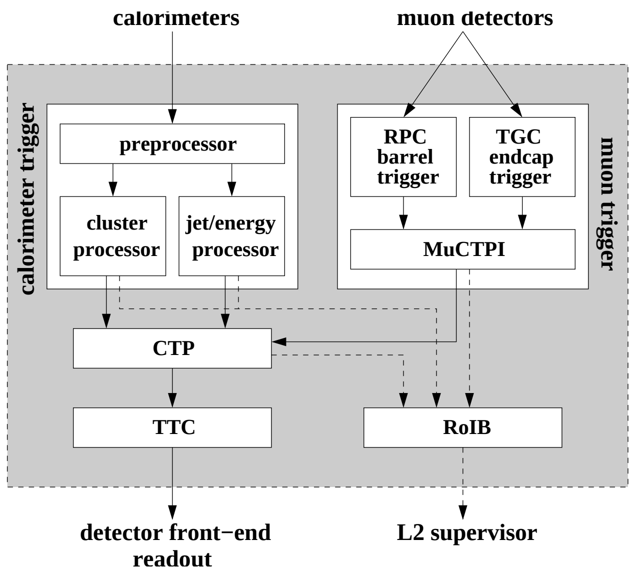


Figure 1: An overview of the ATLAS LVL1 trigger system. Solid lines indicate information on the latency-critical trigger path, dashed lines RoI information.

The purpose of the offline simulation of the LVL1 trigger is multifold: First, it (or at least parts of it) may be used for tests of the LVL1 trigger hardware by feeding simulation and hardware with the same configuration and input data and comparing the outputs. Actually, parts of the simulation, especially in the muon sector, were originally designed for that purpose and were first provided as stand-alone programs which had to be, or are still being, incorporated into the ATLAS offline computing framework Athena and combined with the other parts of the LVL1 simulation. Second, the simulation is already now used for tests and developments of the HLT environment, especially for the generation of the LVL1 result as input data to the development of HLT algorithms and of the HLT test beds. Third, the

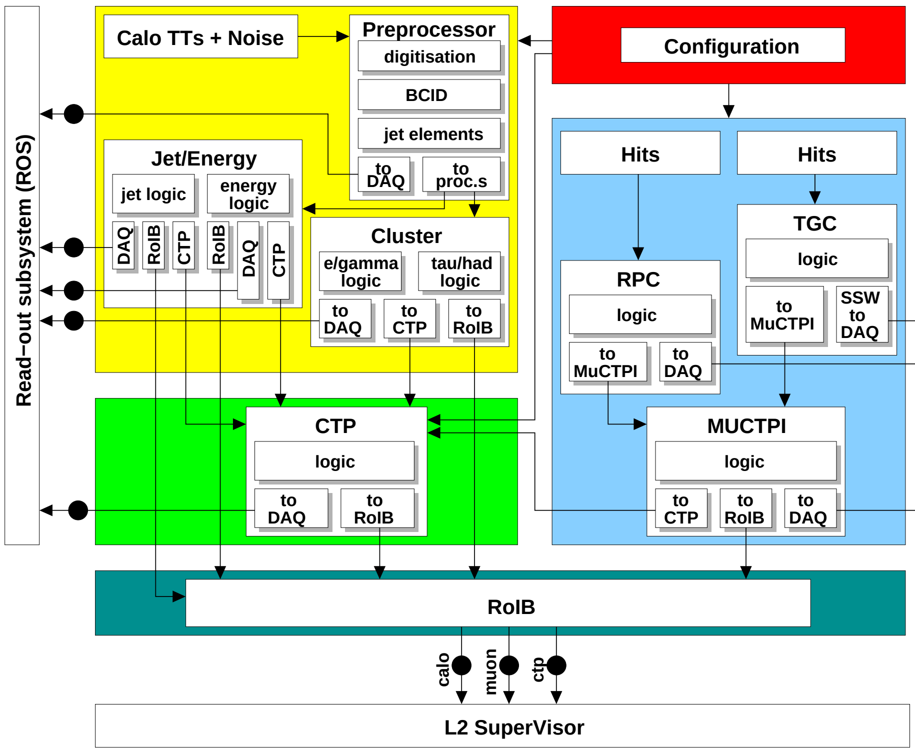


Figure 2: An more detailed overview of the ATLAS LVL1 trigger system and its connections to other parts of ATLAS.

offline simulation will be part of the ATLAS detector simulation which will be used for the production of Monte Carlo events, e.g. for physics studies.

The purpose of this note is:

1. to describe basic parts of the LVL1 simulation: The LVL1 configuration, the CTP simulation, the RoIB simulation, the definition of the LVL1 raw data object (RDO, see later in this note), and the LVL1 result byte stream (BS, see also later) conversion;

2. to illustrate how to install a LVL1 simulation environment;

3. to explain how to run the LVL1 simulation and how to interprete its results.

The parts of the LVL1 simulation that will not be explained here (the calorimeter and muon trigger simulation) are, or will be, documented in other places [2]. The HLT Technical Design Report [3] also contains a section on the LVL1 simulation and additional references.

The note is organised as follows: Section 2 gives a short overview of the LVL1 simulation. Sections 3 through 8 contain information on the various parts of the LVL1 simulation which are within the scope of this note. Section 9 is concerned with the various installation options for the LVL1 simulation, including the stand-alone option and with a demonstration of how to run the code using a very basic job-options file. Section 10, finally, gives a detailed list of things which should be done to improve the LVL1 simulation. This list contains rather general points (like thorough tests of the software for memory leaks) as well as very detailed design and implementation issues for which the author did not find time before he had to leave ATLAS.

2 The LVL1 Trigger

The task of the LVL1 trigger [1] is to select or reject events based on the information provided by the calorimeters and muon detectors. In order to make its decision, the signals found in the detectors are compared to the signal expectation for the passage of certain particles through the detector:

The calorimeter trigger [1] takes care of finding localised energy depositions caused by electrons or photons, by tau leptons or hadrons, or by hadronic jets. It discriminates the transverse energies E_T of the candidates against a set of programmable thresholds which are taken from the configuration, and counts the number of candidates that pass each of the thresholds. There are up to 16 thresholds for electron/photon candidates, up to 8 for tau leptons or hadrons, 8 for (central) jets and 4 for jets in the forward regions of the detector. In addition, the calorimeter trigger calculates total energy sums (total transverse energy, total missing transverse energy) which are also discriminated against thresholds. The numbers of objects passing each of the thresholds, the so-called *trigger threshold multiplicities* are sent to the CTP. See Refs. [1, 4] for an overview on the algorithms used to detect the calorimeter trigger candidate objects.

The muon triggers in the barrel and in the endcap [1, 5] detect muon candidates based on the hits in fast muon trigger chambers. The candidates found are discriminated in transverse momentum p_T against six programmable thresholds, and the multiplicities are again sent to the CTP, via the MuCTPI which combines the information from the RPC and TGC subsystems.

The calorimeter and muon triggers also deliver more detailed information on the candidates, the so-called Regions-of-Interest or RoIs, to the Region-of-Interest builder (RoIB). The information that is sent comprises bits to indicate the energy threshold(s) that was (were) passed and bit patterns that characterise the location of the candidate in the detector in terms of the electronic modules via which it was processed. This information is used to seed the HLT decision process.

The CTP checks the input trigger threshold multiplicities it receives from the calorimeter trigger and from the MuCTPI against so-called *trigger conditions* which it takes from the trigger menu. These conditions, or rather their logical values, are then combined to complex *trigger items*, each of which corresponds to a physics signature LVL1 is supposed to trigger on (see Section 4 for more information about the trigger menu and its use in the CTP simulation). The final LVL1 signal is the logical OR of all items – each fulfilled trigger item is sufficient to trigger the event on LVL1, provided that the item in question is not masked, the prescale mechanism has not avoided the item to trigger, and the dead-time logic has not prevented the CTP from triggering. The result of the CTP, together with some information on its inputs and on the internal data processing, is then sent to the RoIB where it is concatenated with the RoI information from the calorimeter and muon triggers to give the LVL1 result[1].

[1]It should be noted that the calorimeter and muon triggers as well as the CTP also deliver data to the

3 Framework, Environment and Status

The LVL1 trigger simulation is implemented within the ATLAS offline computing framework Athena [6] using C++. It follows the hardware closely as can be seen from the package view of the simulation code presented in Fig. 3. The code is divided into the simulations of the calorimeter trigger (package TrigT1Calo), the muon RPC trigger (indicated by the generic package name TrigT1RPC), the muon TGC trigger (package TrigT1TGC), of the CTP (package TrigT1CTP) and of the RoIB (package TrigT1RoIB). The details of the hardware, however, are not simulated for all hardware components (for example the calorimeter trigger, for which a separate detailed hardware simulation exists). Omitting the hardware details allows for higher speed and facilitates the use of the simulation for Monte Carlo event generation. In addition to these obvious parts of the simulation, further packages contain the LVL1 raw data object (RDO) definition (package TrigT1Result) and the code needed for conversion between RDO and byte stream (BS), and vice versa (package TrigT1ResultByteStream). One package contains class definitions and implementations for classes that are used by more than one package (package TrigT1Interfaces); this was introduced to avoid circular inclusions via CMT 'use' statements.

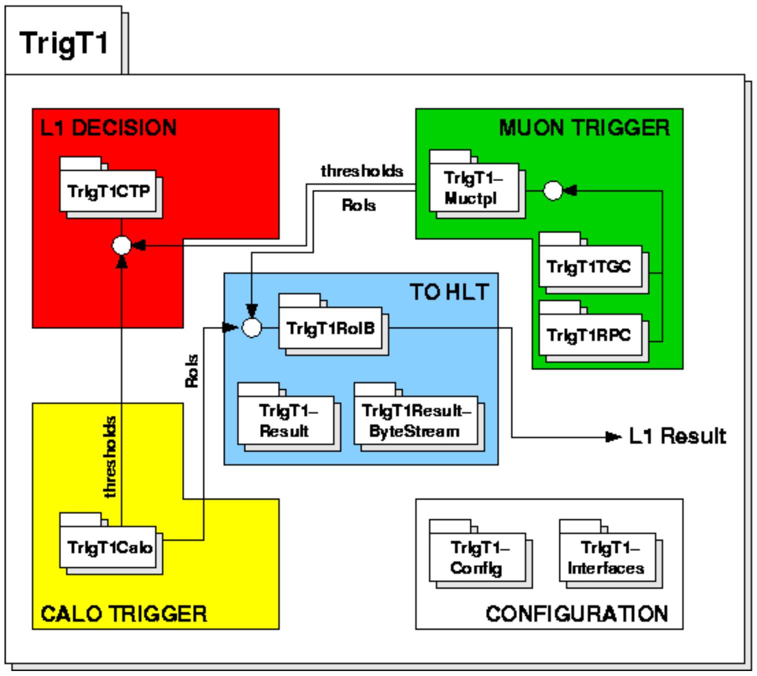


Figure 3: A package view of the LVL1 trigger simulation. Except for the muon RPC simulation the names of the packages given in the figure correspond to their names in the offline software repository offline/Trigger/TrigT1. In case of the muon RPC simulation, the package generically labelled 'TrigT1RPC' comprises several packages.

Some effort has gone into defining the interfaces between the different components of

read-out system. This data path, which is not yet implemented for all of the LVL1 simulation components, is however of minor importance for the LVL1 simulation and will not be treated here.

the simulation. The definitions, which follow very closely the data formats used in the corresponding trigger hardware parts, are documented in a separate document [7] and are based on the data formats and interfaces defined for the hardware in Ref. [8].

Since the LVL1 simulation is still evolving, a version of the code has to be singled out as the reference for this note. For this purpose the code that went into the offline Release 6.0.3 (end of April 2003) is chosen. Users of the LVL1 simulation will need to do the obvious adaptions to newer versions of the simulation. The status of the simulation for Release 6.0.3 was the following:

+ The LVL1 configuration, following the needs of the existing demonstrator hardware implementation of the CTP, was ready; the configuration data provided by it were used for the configurations of the calorimeter trigger and CTP simulations. In addition, the configuration was used for the interpretation of the LVL1 RoIs within the process of data conversion from BS to the reconstructed RDOs.

+ The calorimeter trigger simulation was fully functional, with parts of the jet and energy triggers still to be validated.

+ The simulation of the muon RPC detectors was functional; however it did not yet use the configuration data provided by the LVL1 configuration process.

+ The simulations of the CTP and of the RoIB were fully functional.

+ The conversion to and from BS was fully functional.

- It was not possible to run the calorimeter trigger and muon RPC trigger simulations simultaneously, for unknown reasons.

- The muon TGC trigger simulation was not yet integrated in the overall LVL1 simulation effort.

4 Configuration of the LVL1 Simulation

4.1 Overview

The task of the LVL1 trigger configuration is twofold:

First, the trigger menu, i.e. the collection of event signatures LVL1 is supposed to trigger on, has to be translated into something that the simulation of the CTP can understand and use in making the event decision based on the inputs provided by the calorimeter and muon trigger simulations: The LVL1 signatures, or *trigger items* are combinations of requirements (or *trigger conditions*) on the multiplicities of various kinds of candidate objects delivered by the calorimeter and muon triggers.

A simple example for a trigger item is

one or more electron/photon candidate with $E_T > 10$ GeV
and
one or more muon candidate with $p_T > 15$ GeV.

In a frequently used and obvious notation this reduces to

$$\text{1EM10+1MU15}$$

with the two trigger conditions '1EM10' and '1MU15'. The string 'EM' ('MU') represents an electron/photon (muon) candidate, and the integer numbers before and after the string symbolise the required multiplicity and the required transverse energy/momentum, respectively. The combination of a candidate string and a threshold value like 'EM10' is called a *trigger threshold'*. See Subsection 4.3 for more details on the configuration of the trigger menu. Besides the 'EM' and 'MU' string, other strings may be used to signal tau/hadron candidates ('HA'), jets or forward jets ('JT', 'FR', 'FL'), the total and total missing transverse energy ('ET', 'TM'), and the transverse energy calculated from the sum of all jet E_T ('SM').

Second the calorimeter and muon triggers have to be configured such that they deliver the information required for the event decision by the trigger menu, i.e. that the multiplicities for the required trigger thresholds are sent to the CTP simulation. For the implementation of the above example the calorimeter trigger has to be configured such that it delivers to the CTP the multiplicity count for the threshold 'EM10', i.e. the number of electron/photon candidates with transverse energy above 10 GeV. It is obvious that the trigger menu and the trigger thresholds for the calorimeter and muon triggers have to be defined consistently. In particular, all thresholds used in the definition of any trigger condition in any trigger item must be delivered by the calorimeter and muon trigger simulations and thus need to be configured. See Subsection 4.2 for more details on the threshold configuration.

In the configuration process for the CTP, the restriction imposed by limited abilities and resources of the simulated hardware are taken into account; this leads to a limit on the number of input bits over which the threshold multiplicities are sent to the CTP (currently 2×16 according to the layout of the two LUTs on the CTP demonstrator) and on the number of trigger items (also 2×16). These restrictions will have to be reconsidered when aiming for a simulation of the final CTP as it is currently being designed; this device will have 160 input bits and probably 160 or more trigger items. However, the approach chosen is scalable, and the full functionality should also be available for the final design, as long as the basic structure involving trigger thresholds, trigger conditions and trigger items remains. Subsection 4.4 explains the mechanism that is used to take these hardware limitations into account.

A final piece of configuration software is used to configure details of the MuCTPI simulation. The MuCTPI has the ability to apply a minimum p_T cut on the muon candidate RoIs that it delivers to the RoIB (actually two different cuts in terms of the threshold that must be passed are applied to the highest and second-highest p_T candidate of each sector). In addition, the maximum number of RoIs to be delivered to the RoIB is configurable. Subsection 4.5 shows the details of this configuration step.

The LVL1 trigger configuration software is currently being adapted to also be able to configure the LVL1 trigger hardware by deriving the necessary look-up table and FPGA configuration files from the trigger menu and trigger threshold list. Such a common configuration scheme will allow for cross-checks between hardware and software (see Section 9.3 for more details on this issue).

4.2 XML Definition of Trigger Thresholds

Both the trigger menu and the list of thresholds that have to be configured are defined using XML and are parsed into instances of C++ classes using the Xerces DOM API [9]. The notation exploits the facility of XML to define logical structures by introducing self-defined tags. The tag structure used for the definition of the trigger thresholds is

```
<TriggerThresholdList>
   <TriggerThreshold name="..." type="..." bitnum="...">
      <TriggerThresholdValue thresholdval="..." />
   </TriggerThreshold>
</TriggerThresholdList>
```

The important point here is that a trigger threshold consists of one (or more) *trigger threshold values*. This concept allows one to assign different threshold values (in GeV) to various topological regions of the detector, a concept which is foreseen for the calorimeter and muon trigger hardware but which has not yet been studied in detail.

The <TriggerThreshold> tag has several attributes:

- The ‘name’ attribute serves to give a unique name to the trigger threshold which serves to connect it to the trigger conditions that use it in the course of the simulation process. An example for the attribute is ‘name=“EM01”’ note that the threshold value does not show up in the name since it might vary over the detector.
- The ‘type’ attribute is necessary for the configuration code to know what kind of threshold it is currently working on. This information has to be present, for example, because the total number of thresholds of one type is limited and thus has to be controlled. There exists only a limited amount of types, and the use of other types than these will lead to an error message: ‘EM’ (electron/photon), ‘HA’ (tau/hadron), ‘MU’ (muon), ‘JT’ (jet), ‘FL’ or ‘FR’ (forward jets), ‘ET’ (transverse energy), ‘SM’ (transverse energy from jets), and ‘TM’ (missing transverse energy). In addition strings for technical triggers (calibration, random etc.) may be provided.
- The ‘bitnum’ attribute is required for a correct simulation of the CTP hardware which receives all calorimeter and muon trigger informations, i.e. the trigger threshold signals, on a fixed number of input lines, typically three: ‘bitnum=“3”’ (2 for the forward jets, FL and FR, and 1 for all energy triggers).

The <TriggerThresholdValue> tags also have a number of attributes, with the selection of the attributes used for a given tag depending on the ‘type’ attribute of the embracing <TriggerThreshold> tag. The attributes that may be used are:

- ‘thresholdval’ – which can be used for all kinds of trigger threshold values, with obvious meaning;
- ‘emisolation’, ‘haisolation1’ and ‘haisolation2’ – to characterise the isolation properties of the electron/photon and tau/hadron candidates (actually, the ‘haisolation2’ should more appropriately be called ‘haveto’ - see Section 10);

- ‘phimin’, ‘phimax’, ‘etamin’, ‘etamax’ – in order to give topological constraints to the trigger threshold value in question;
- ‘window’ – to give the size of the cluster that the calorimeter jet/energy processor simulation uses to search for jet candidates;

An example of a trigger threshold file is given in Appendix A.2. The required document type definition (DTD) file *trigger2.dtd*, which defines the data and their types to be used in the trigger threshold file, can be found in Appendix A.5.

4.3 XML Definition of the Trigger Menu

The basic structure of the trigger menu XML files is the following:

```
<TriggerMenu TM_ID="...">
   <TriggerItem TI_ID="..." mask="..." priority="..." prescale="...">
      <TriggerCondition threshold="..." mult="..."/>
   </TriggerItem>
</TriggerMenu>
```

In addition to the <TriggerMenu>, <TriggerItem> and <TriggerCondition> tags also tags <AND>, <OR> and <NOT> are available which may serve to make the trigger item an arbitrary logical function of the trigger conditions involved. For example:

```
<TriggerMenu ...>
   <TriggerItem ...>
      <AND>
         <TriggerCondition ... />
         <OR>
            <TriggerCondition ... />
            <NOT>
               <TriggerCondition ... />
            </NOT>
         </OR>
      </AND>
   </TriggerItem>
</TriggerMenu>
```

The <TriggerMenu> tag has as only attribute ‘TM_ID’ – a string which provides a name for the trigger menu in question. The <TriggerItem> has as attributes ‘TI_ID’ (a name), ‘mask’ (indicating via values “on” or “off” whether or not the item is to be used in the LVL1 decision), ‘priority’ (“low” or “high”, indicating whether the item should have priority, i.e.override the CTP dead-time algorithm – not yet used in the CTP simulation), and ‘prescale’ (an integer indicating the prescale factor to be used in the CTP simulation). The <TriggerCondition> tag has two attributes: ‘threshold’ must correspond to the ‘name’ attribute of a defined trigger threshold tag – this is the threshold the multiplicity of which will be compared with the the integer provided in the ‘mult’ (for ‘multiplicity’) attribute of the <TriggerCondition> tag. The configuration code will generate an error message and force the program to exit if a trigger threshold required by a trigger condition is not configured. For the energy triggers the ‘mult’ attribute is always required to be one.

An example of a trigger menu file is given in Appendix A.1. The corresponding DTD files *entities.dtd* and *trigger.dtd* can be found in Appendices A.4 and A.3, respectively.

4.4 XML Definition of CTP Hardware

An example for a CTP hardware file is given in Appendix A.7 for the case of the CTP demonstrator layout, with the corresponding DTD file hardware.dtd shown in Appendix A.8: First, two LUTs (tag <LUT>) are defined with 16 input bits (tag <PIT> and 8 output bits (tag <MIO>). The attributes 'range_begin' and 'range_end') of the <PIT> tag identify the physical input lines to the LUTs (and hence to the CTP) on which the threshold multiplicities are encoded, and the corresponding attributes of the <MIO> tag specify the lines on which the trigger conditions information will be passed on. Then, two programmable devices (tag <CMB>) are defined with 16 input bits (for the 16 possible trigger conditions – tag <MIO>) and 16 output bits (for 16 items, tags <TBV>) each[2] are given. Again, the 'range_begin' and 'range_end' attributes identify the physical lines from which the trigger conditions for the programmable device in question will be taken (tag <MIO>) or on which the final trigger items will be output (<TBV>). The example given is designed such that all conditions from both LUTs are input to both programmable devices.

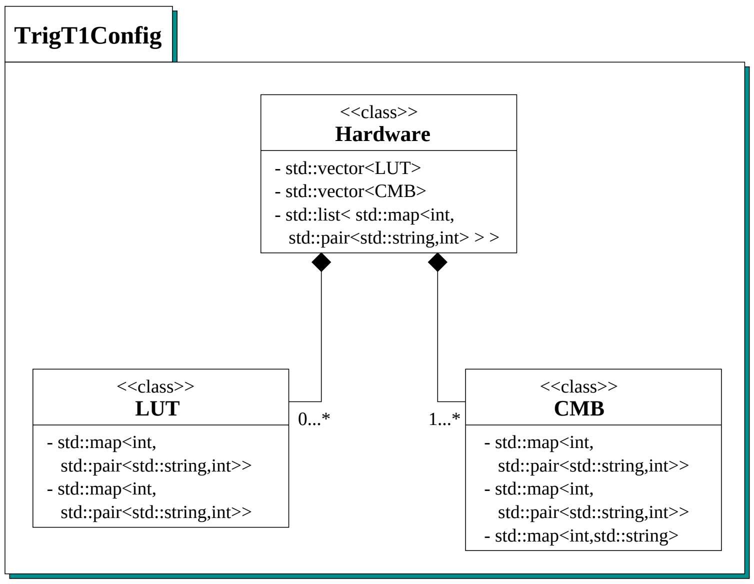

TrigT1Config
<<class>>
Hardware
- std::vector<LUT>
- std::vector<CMB>
- std::list< std::map<int, std::pair<std::string,int> > >
<<class>>
LUT
- std::map<int, std::pair<std::string,int>>
- std::map<int, std::pair<std::string,int>>
0...*
1...*
<<class>>
CMB
- std::map<int, std::pair<std::string,int>>
- std::map<int, std::pair<std::string,int>>
- std::map<int,std::string>

Figure 4: A UML class diagram of the classes involved in the CTP hardware configuration. The data members indicated for the classes are explained in the text.

Figure 4 shows a UML class diagram of the classes involved in the hardware configuration; they all reside in the TrigT1Config package. A short explanation will be given for the data members that are indicated; these are filled during the *mapHardware* method execution of the *Hardware::init* step, see Subsection 4.7.

- *Hardware*: std::vector<LUT> m_HardwareLUTVector: A vector containing the look-up table instances (class *LUT*) configured for the CTP.

[2]The acronym TBV stands for 'trigger before veto' and reflects the fact that, after the logical combination of conditions to items, the trigger items are subject to masks and the dead-time vetos (leading to 'trigger after veto' or TAV) and to prescales (leading to 'trigger after prescale' or TAP). The final decision is built from the logical OR of all TAP bits.

- *Hardware*: std::vector<CMB> m_HardwareCMBVector: A vector containing the instances of the programmable devices (class *CMB*) configured for the CTP.
- *Hardware*: std::list<std::map<int,std::pair<std::string, int>>> m_HardwareMIOMapList: A list containing a map for each configured LUT. The maps build a relation between the number of the MIO (the lines on which the trigger conditions are sent from the LUTs to the programmable devices) and the pair built from the corresponding trigger condition name (std::string) and the multiplicity it requires.
- *LUT*: std::map<int,std::pair<std::string,int>> m_LUTMIOMap: The map mentioned above between the MIO (trigger condition signal line) number and the trigger condition name and the corresponding multiplicity requirement.
- *LUT*: std::map<int,std::pair<std::string,int>> m_LUTPITMap: A map between the PIT (input trigger threshold signal line) number and the trigger threshold name and the corresponding number of bits (PITs) used for this threshold.
- *CMB*: std::map<int,std::pair<std::string,int>> m_CMBMIOMap: A map between the MIO (trigger condition signal line) number and the trigger condition name and the corresponding multiplicity requirement.
- *CMB*: std::map<int,std::pair<std::string,int>> m_CMBPITMap: A map between the PIT (input trigger threshold signal line) number (in case a PIT is directly input to the programmable devices and not first to the LUTs) and the trigger threshold name and the corresponding number of bits (PITs) used for this threshold.
- *CMB*: std::map<int,std::string> m_CMBTBVTMap: A map between the TBV (trigger item signal line) number and the corresponding trigger item name.

4.5 XML Definition of MuCTPI Parameters

The default trigger.muctpi.xml configuration file has the following form:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE MuctpiConfig SYSTEM "muctpi.dtd">
<!-- @version: -->
<MuctpiConfig>
   <MUCTPI firstMin="4" secondMin="2" numCand="10" />
</MuctpiConfig>
```

It defines that the first (second) candidate in each sector must pass at least the threshold number four (two) in order to be sent to the RoIB. In addition, the ‘numCand’ attribute of the <MUCTPI> tag defines the maximum number of candidates that can be sent to the RoIB (default 10).

The corresponding DTD file muctpi.dtd is specified in Appendix A.6.

4.6 Configuration Class Structure

The LVL1 trigger configuration is implemented in the offline/Trigger/TrigT1/TrigT1Config package; some classes which are also needed by other packages reside in the TrigT1Interfaces package in order to avoid circular inclusions (the *TriggerThreshold* and *TriggerThreshold-Value* classes and the classes inheriting from these abstract base classes). Figure 5 shows a UML class diagram of the classes involved in the configuration – most of them are directly related to an XML tag already introduced above.

The central class of the configuration is *TriggerMenu*; the one (and only one) instance of this class contains or has access to all the information in the trigger menu XML file. Consequently, the instance has as data member a vector of instances of the *TriggerItem* class – a trigger menu basically IS a collection of trigger items. Each item, in turn, contains a vector of *TriggerCondition* instances – a trigger item is a logical combination of one or more trigger conditions.

In addition to the trigger items, the instance of *TriggerMenu* holds a map which combines std::strings and pointers to the trigger thresholds (class *TriggerThreshold*) defined in the trigger thresholds XML file. *TriggerThreshold* is an abstract class from which several subclasses inherit which are adapted to the needs of the various triggers available in the LVL1 system (muon, electron/photon/tau/hadron, jet, energy). As explained above, a <TriggerThreshold> contains one or several trigger threshold values (class *TriggerThresholdValue*), and also this class is abstract and subclassed by various classes. A factory pattern is used to create the correct *TriggerThreshold* and *TriggerThresholdValue* instances according to the 'type' attribute in the <TriggerThreshold> XML tag.

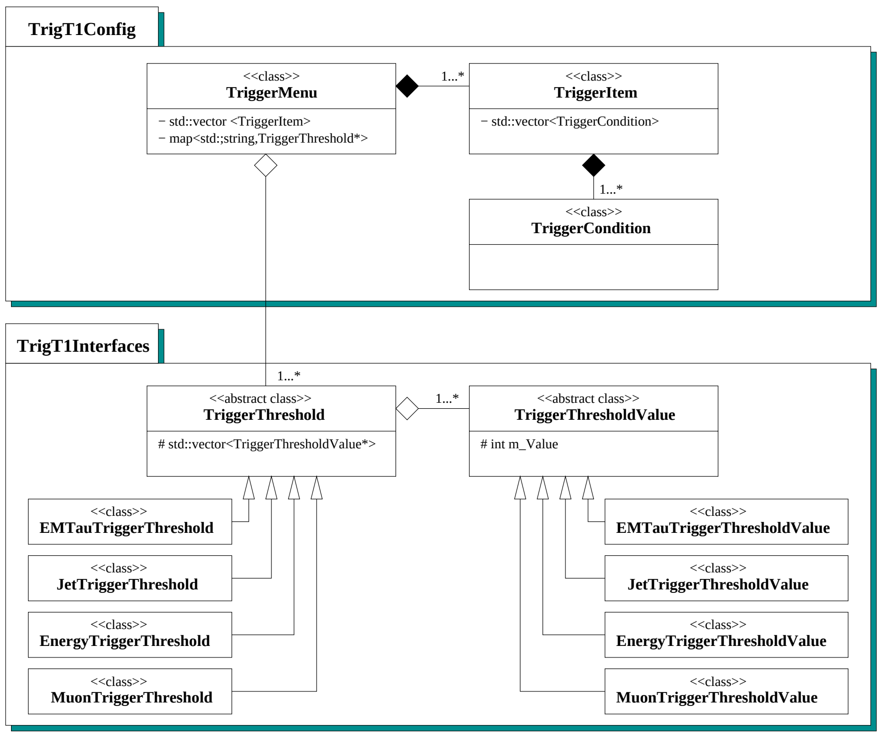


Figure 5: UML class diagram of the trigger configuration software.

The instance of *TriggerMenu* therefore contains the complete logical structure that was defined in the trigger menu and trigger thresholds XML files; the connection between the trigger conditions and the trigger thresholds which are discriminated by the former is made using string comparisons between the 'name' and 'threshold' attributes of the *TriggerThreshold* and *TriggerCondition* instances, respectively.

The DOM[3] parser translates or 'parses' the XML information into a logical tree structure as shown in a simplified manner in Fig. 6 for the trigger menu XML file. Using this structure, which resides in memory, together with values of the data members of the corresponding C++ instances (which are translations of the XML attributes of the respective tags) it is possible to derive a logical value for the trigger menu instance given certain logical values for the trigger condition instances (there are no class counterparts for the <AND>, <OR>, <NOT> tags – this logical information is taken only from the XML tree). The conditions in turn get their logical values from the discrimination process in which the multiplicity delivered for a given trigger threshold is compared to the multiplicity required by the conditions which relate to the trigger threshold in question[4].

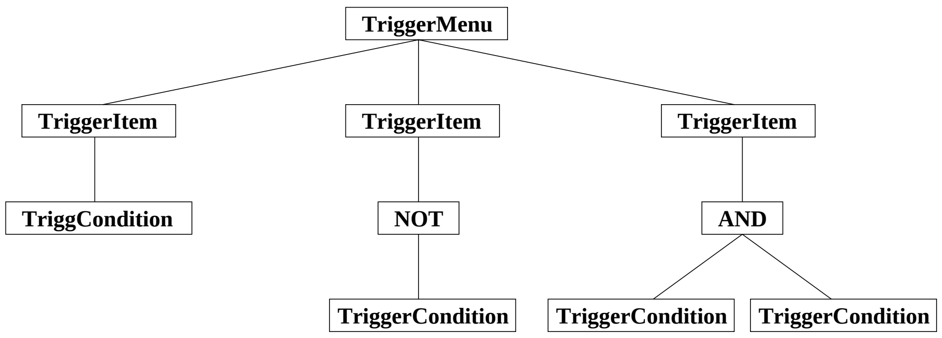


Figure 6: Schematic overview of a simple XML tree built from a trigger-menu file.

The configuration process also comprises configuration of the calorimeter and muon trigger simulations. For this purpose the classes indicated in Fig. 7 have been introduced into the package offline/Trigger/TrigT1/TrigT1Interfaces. There is one instance for each of the classes in the left half of this figure: The *CTPCaloConfig* instance contains the trigger thresholds to be delivered by the calorimeter electron/photon/tau/hadron trigger, the *CTPJetEnergyConfig* instance contains the thresholds for the jet and energy triggers, and the *CTPMuonConfig* contains the thresholds configuring the muon RPC and TGC triggers.

A last bit of simulation is needed to provide configuration data for the MuCTPI; this is achieved using a *MuctpiConfig* object which is created using the information provided in the trigger.muctpi.xml file (see above).

[3]DOM is, besides SAX, the most important API to handle XML documents.

[4]This little excursion into the simulation, rather than configuration, world shows that the configuration of the CTP cannot easily be separated from the simulation of the device: Simulating the CTP response to a given event simply means having certain numbers, namely the input threshold multiplicities, 'run' through the logical tree defined by the XML trigger menu file.

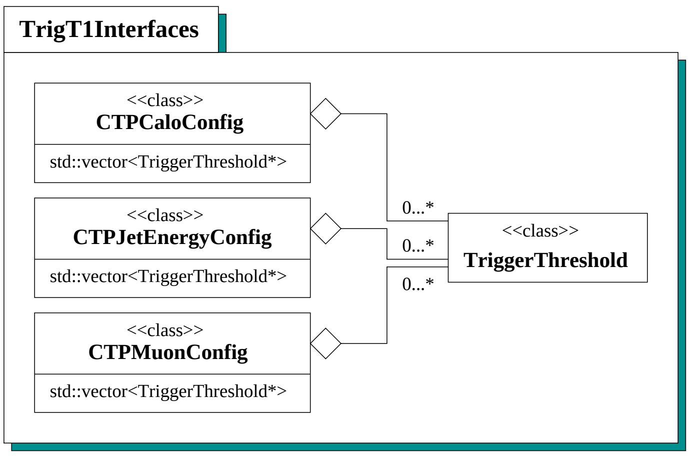


Figure 7: UML class diagram of the configuration for muon and calorimeter triggers.

4.7 Implementation

4.7.1 The *L1Config* Algorithm

Figure 8 shows a sequence diagram for the action of the *L1Config* algorithm from the TrigT1Config package. The algorithm first creates a *TriggerMenu* object and calls its *init* method. In this method, first the XML parsing setup is performed (not shown in the figure), then in three steps *mapThresholds*, *mapThresholdValues* and *mapTriggerMenu* the thresholds and trigger menu XML files are parsed into instances of C++ objects of the classes shown in Fig. 5.

TriggerMenu::mapThresholds creates, for each <TriggerThreshold> XML tag, a *TriggerThreshold* object. A factory pattern involving the *TriggerMenu::createThreshold* method, is used to create the kind of *TriggerThreshold* fitting the 'type' attribute of the XML tag (*EMTau-, Jet-, Energy-, MuonTriggerThreshold*). A pair made from the 'name' attribute of the <TriggerThreshold> tag and a pointer to the *TriggerThreshold* object is pushed back into a data member *map<std::string,TriggerThreshold*> m_TriggerMenuThresholdMap* of the *TriggerMenu* object. Fig. 9 shows, as an example, a sequence diagram for the creation of a *EMTauTriggerThreshold* object.

In a next step, in method *TriggerMenu::mapThresholdValues*, a loop over the member *map<std::string,TriggerThreshold*>* of the *TriggerMenu* class is implemented, calling for each *TriggerThreshold* a method *mapValues* in which the *TriggerThresholdValue* objects of the correct type (again depending on the 'type' attribute of the <TriggerThreshold> tag) are created, again using a factory pattern in the method *TriggerThreshold::createThresholdValue*, see Fig. 10. The *TriggerThresholdValue* pointers are inserted into the *TriggerThreshold::-m_TriggerThresholdValueVector* data member.

In a next step of the *L1Config* algorithm, the *TriggerMenu::mapTriggerMenu* method is used to parse the trigger menu XML file into class instances of *TriggerItem* and *TriggerCondition* objects. For each <TriggerItem> tag encountered, a *TriggerItem* object is created, and it is inserted into the *TriggerMenu::m_TriggerMenuItemVector* data member of the *Trig-*

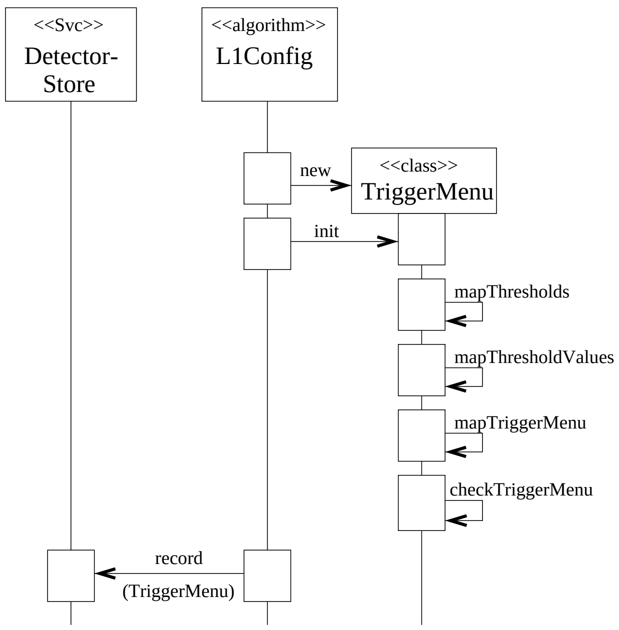


Figure 8: Sequence diagram for the main LVL1 configuration step which sets up the *TriggerMenu* instance. See text for explanations.

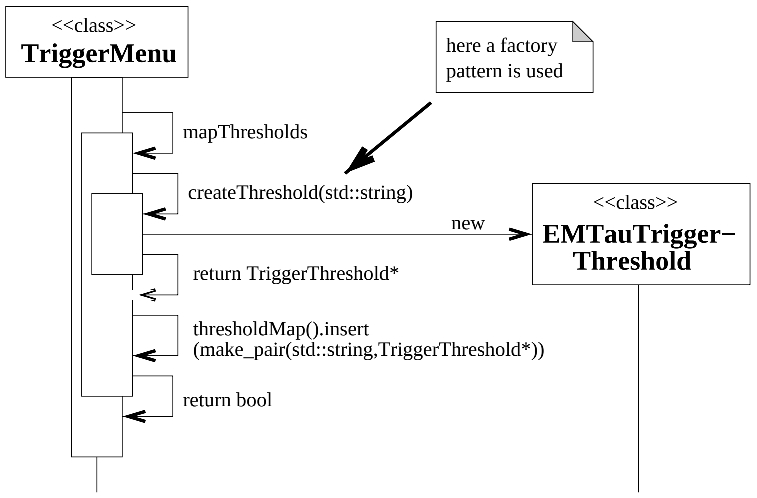
gerMenu object. Similarly, for each <TriggerCondition> tag encountered, a *TriggerCondition* object is created and inserted into the *TriggerItem::m_TriggerItemConditionVector* data member of the corresponding *TriggerItem* object.

Finally, some checks will be performed on the *TriggerMenu* object in order to check its completeness and consistency. This is done in the method *TriggerMenu::checkTriggerMenu* which will not be discussed in detail here.

4.7.2 The *CTPHardware* Algorithm

The task of this algorithm is to introduce into the configuration knowledge about the hardware of the CTP. Figure 11 shows a sequence diagram of the algorithm. In a first step, it retrieves the *TriggerMenu* object that was created by the *L1Config* algorithm from the DetectorStore. Next, the *Hardware::init* method is called which performs several actions:

- It sets all input lines to zero (method *Hardware::resetHardware*).
- It parses the CTP hardware XML file in the usual recursive way, creating *LUT* and *CMB* objects for the corresponding XML tags, and filling the respective data members (method *Hardware::mapHardware*).
- It sets the *m_TriggerThresholdPITStart* member of the *TriggerThreshold* objects contained in (or pointed to by) the *TriggerMenu::m_TriggerMenuThresholdMap* member of the *TriggerMenu* instance. This is achieved by looping over all *LUT* objects and all *TriggerThreshold* pointers and checking whether the number of bits required for the *TriggerThreshold* in question is smaller or at maximum equal to the number of



<<class>>
TriggerMenu
here a factory pattern is used
mapThresholds
createThreshold(std::string)
new
<<class>>
EMTauTrigger−Threshold
return TriggerThreshold*
thresholdMap().insert (make_pair(std::string,TriggerThreshold*))
return bool

Figure 9: Sequence diagram for the *TriggerMenu::mapThresholds* method, showing the creation of an *EMTauTriggerThreshold* object.

input lines (PITs) not yet occupied by other thresholds on this LUT (method *Hardware::buildLUTPITMaps*).

- According to the limited number of input lines to the *LUT* objects, it may happen that not all *TriggerThreshold* objects can be placed on one of the input lines to one of the *LUT* objects. This is checked in the *Hardware::checkConsistencyWithTM* method. For each *TriggerThreshold* not placed on of the *LUT* input lines, an error message is issued which also states the *TriggerItem* objects that contain *TriggerCondition* objects which refer to the affected *TriggerThreshold* objects.
- Finally, the configuration files for the hardware LUTs and programmable devices are calculated and printed in the methods *buildLUTconfigs*, *buildMIOMapList* and *buildCMBconfigs* of the *Hardware* class. These steps will not be treated in detail here, instead a sequence diagram is shown in Fig. 12.

4.8 Job-Options Fragment

The standard job-options fragment for the LVL1 trigger configuration that is included by higher-level job-options files looks like the following:

```
//--------------------------------------------------------------
// TrigT1Config Algorithms Private Options
//--------------------------------------------------------------
ApplicationMgr.DLLs += { "TrigT1Config" };
ApplicationMgr.TopAlg += {"LVL1CTP::L1Config/L1Config"};
ApplicationMgr.TopAlg += {"LVL1CTP::CTPHardware/CTPHardware"};

//--------------------------------------------------------------
// Set output level threshold (1=VERBOSE to 6=FATAL )
```

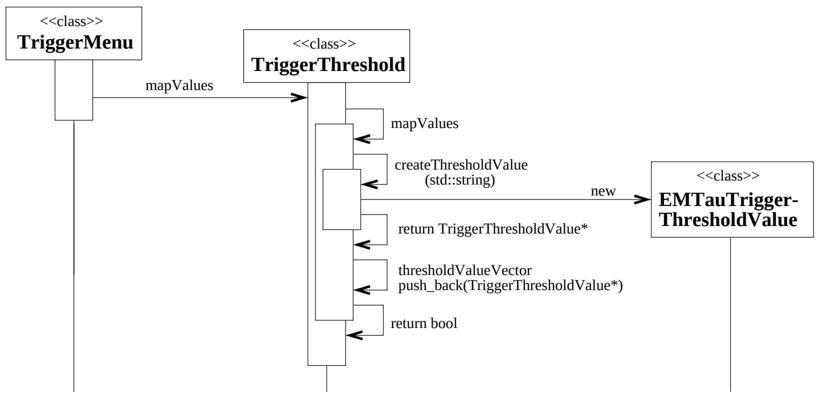


Figure 10: Sequence diagram for the *TriggerThreshold::mapValues* method, showing the creation of an *EMTauTriggerThresholdValue* object.

```
//------------------------------------------------------------
L1Config.triggerMenuFileLocation = "triggermenu.looseLVL1.low.xml";
L1Config.thresholdListFileLocation = "triggerthresholds.looseLVL1.low.xml";
CTPHardware.hardwareXMLFile = "hardware.ctpd.xml";
L1Config.muctpiConfigLocation = "trigger.muctpi.xml";
CTPHardware.printoutLUT = "no";
```

The library for the package (*TrigT1Config*) is loaded, and the top-level Athena algorithms *L1Config* and *CTPHardware* are declared (see Subsection 4.7). Then the locations for the various XML configuration files are set; the *triggermenu.looseLVL1.low.xml* and *triggerthresholds.looseLVL1.low.xml* are the standard trigger menu and trigger thresholds files chosen for Data Challenge 1. The *hardware.ctpd.xml* file actually contains a configuration that is slightly more complicated then what is allowed by the CTP demonstrator hardware. The file *trigger.muctpi.xml* was introduced above. The flag *CTPHardware.printoutLUT*, which may have values "yes" and "no" indicates whether or not the LUT files which are needed to configure the CTP demonstrator hardware should be printed or not. This flag is not yet fully functional, but it should nevertheless be set only if necessary – for each configured LUT (with 16 input lines and 8 outputs) 65536 lines will be printed.

5 The Central Trigger Processor Simulation

5.1 Overview

The CTP combines the input multiplicities provided by the muon and calorimeter triggers and makes the final LVL1 trigger decision. The Decision process is split in several steps:

1. First, the trigger threshold multiplicities are discriminated against the multiplicity requirements (conditions) as implemented in the trigger menu.

2. Then, the conditions are logically combined to trigger items.

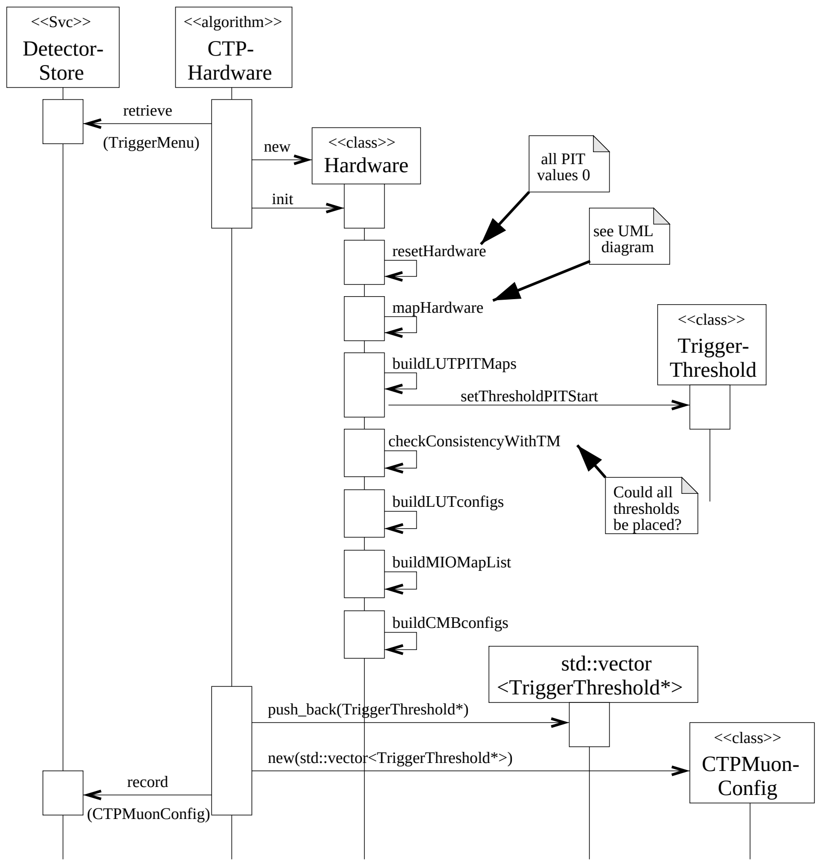


Figure 11: Sequence diagram for the hardware configuration step. The classes involved in the *Hardware::mapHardware* method are visible in the UML class diagram of Fig. 4. Shown is only the configuration step for the muon triggers, involving the *CTPMuonConfig* object, but the principle is the same for the *CTPCaloConfig* and *CTPJetEnergyConfig* objects. See also the text for more explanations.

3. These items are subject to masks, vetos and dead-time algorithms before they are prescaled.
4. The final CTP decision is the logical OR of the logical values of all configured trigger items.
5. The CTP Slink information is built and sent to the RoIB and also to the read-out data path.

The simulation of the CTP follows this scheme closely, with only two minor deviations from the hardware implementation:

- There exists, in the simulation, no implementation of the complex hardware CTP dead-time algorithm which allows to trigger on only N events in a time interval of M μs, with N and M adjustable.

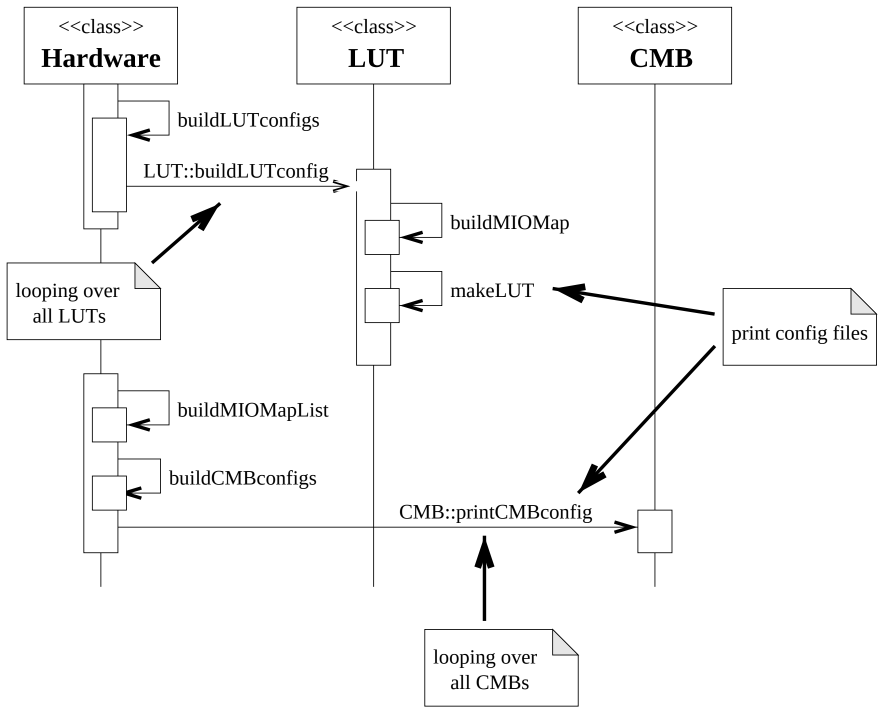


Figure 12: Sequence diagram highlighting the steps taken to get the hardware LUT and CMB configuration files. Please refer to the code for further information.

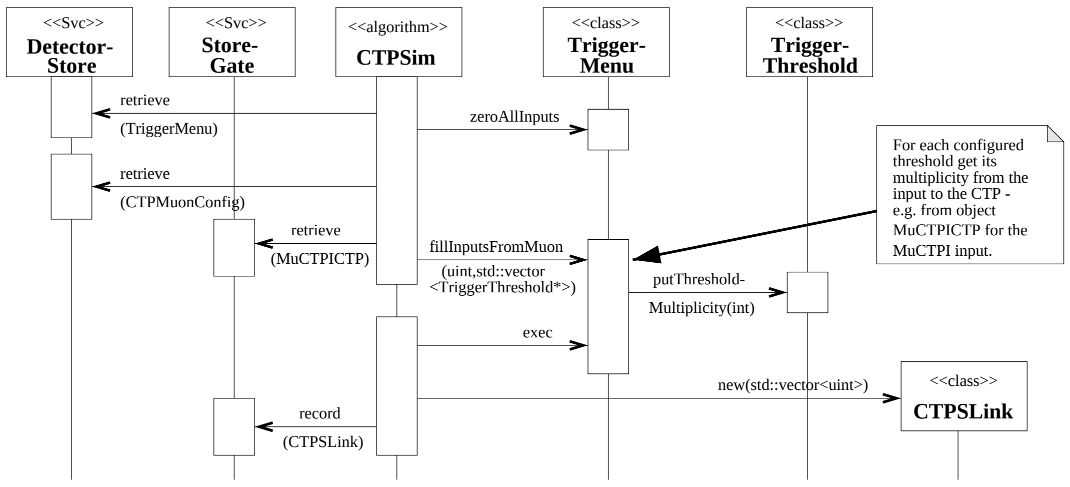
- The output of the CTP to the read-out is not simulated because the CTP hardware delivers the information from 31 time slices around the one that triggered, which is not feasible in a simulation which does not take into account the full timing information. However, the implementation for the triggering time slice should be straight forward since the information equals the information sent to the RoIB in content and format.

5.2 CTP Simulation

The simulation of the CTP is implemented using the *CTPSim* algorithm of the TrigT1CTP package and the *TriggerMenu* class from the TrigT1Config package. The sequence diagram in Fig. 13 gives a rough overview of the simulation process, highlighting only the data relevant for the MuCTPI simulation as inputs. The inputs and configuration data from the calorimeter trigger simulation are treated completely in parallel. First, the main algorithm, *CTPSim*, retrieves the trigger menu object and the *CTPMuonConfig* object from DetectorStore. Then, the event related data from the muon and calorimeter trigger systems are retrieved from StoreGate. The MuCTPI data come as an object of type *MuCTPICTP*, and the calorimeter trigger data are encoded as objects of type *EMTauCTP*, *EnergyCTP* and *JetCTP* for the cluster processor, and the energy and jet triggers from the jet/energy processor, respectively. Figure 14 summarises the various inputs to the CTP simulation.

Once all the input objects to the CTP simulation are collected from StoreGate, methods *TriggerMenu::fillInputsFromXYZ* are called which put the input multiplicities delivered by the muon and calorimeter trigger simulations into the integer data members *TriggerThresh-*

old::m_TriggerThresholdMultiplicity of the corresponding *TriggerThreshold* objects. Then the CTP logic simulation is executed in the call to the *TriggerMenu::exec* method. In this step, which is explained below, all the constituents of the Slink the CTP sends to the RoIB are built and put into a newly created CTPSLink object which is stored in StoreGate.



<<Svc>> Detector-Store
<<Svc>> Store-Gate
<<algorithm>> CTPSim
<<class>> Trigger-Menu
<<class>> Trigger-Threshold
retrieve (TriggerMenu)
zeroAllInputs
For each configured threshold get its multiplicity from the input to the CTP - e.g. from object MuCTPICTP for the MuCTPI input.
retrieve (CTPMuonConfig)
retrieve (MuCTPICTP)
fillInputsFromMuon (uint,std::vector <TriggerThreshold*>)
putThreshold-Multiplicity(int)
exec
new(std::vector<uint>)
<<class>> CTPSLink
record (CTPSLink)

Figure 13: Sequence diagram for the simulation of the CTP. See text for details.

The *TriggerMenu::exec* methods wraps all steps of the CTP logic simulation, which are shown in Fig. 15. The figure does not show the full algorithmic content, but with the exception of the *TriggerMenu::calculateAndSetItemValues* method the methods are pretty simple and understandable.

- The *calculateAndSetItemValues* method loops over all *TriggerItem* objects in the vector of trigger items held by the *TriggerMenu* object and, for each of the items, calls its *TriggerItem::calculateItemValue* method. In this method, the logical value of the trigger item is calculated by calculating recursively the logical values of all constituting trigger conditions from the trigger threshold input multiplicities, and then calculating the item value taking into account the logical operations dictated by the trigger menu. The final item values is stored as data member of the *TriggerItem* objects. The *calculateTBVWord* derives the three 32-bit words required in the CTP Slink for encoding the trigger values before masks and vetos (words 15 to 17 of the CTP Slink).

- The *evaluateMasks* method checks whether the ‘mask’ attribute of the <TriggerItem> tags is set to “on” or “off”; if “off” is chosen, the item value is set to zero. Then, the *calculateMaskWord* derives words 25 to 27 which contain a bit pattern indicating the masks that were set.

- The *evaluateVetoAndDeadTime* method checks whether dead time should be applied (which would result in setting all trigger item values to zero). The method *calculateTAVWord* is then used to derive the words 18 to 20 of the CTP Slink which contain the trigger decision after the application of masks, veto and dead-time.

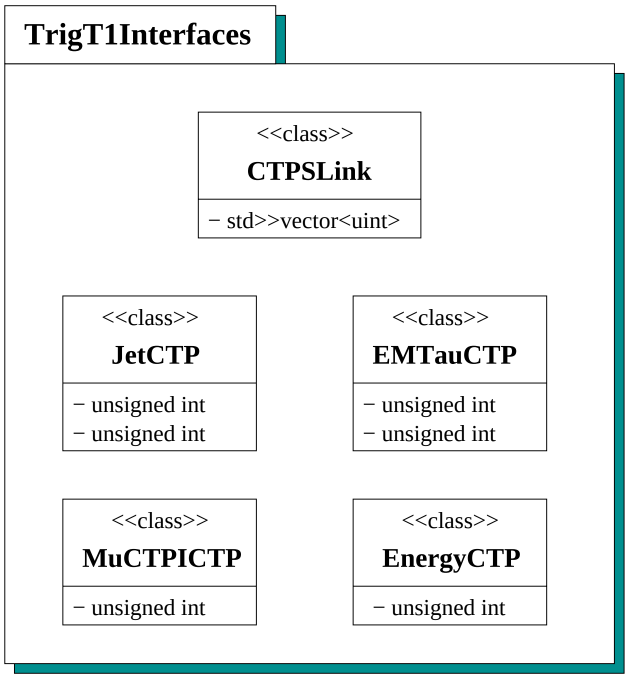


Figure 14: UML class diagram for the inputs to the CTP simulation. See text for details.

- Finally, the prescale mechanism is applied in the method *evaluatePrescales*, setting item values to zero if necessary. The result of this operation is encoded in words 21 to 23 of the CTP Slink using the *calculateTAPWord* method.

5.3 Job-Options Fragment

The standard job-options fragment for the CTP simulation that is included by higher-level job-options files looks like the following:

```
//----------------------------------------------------------
// TrigT1CTP Algo
//----------------------------------------------------------
ApplicationMgr.DLLs += { "TrigT1CTP" };
ApplicationMgr.TopAlg += { "LVL1CTP::CTPSim/CTPSim" };
//ApplicationMgr.TopAlg += { "LVL1CTP::CTPTester/CTPTester" };
CTPSim.ApplyDeadtime = "no";
CTPSim.RandomVersusZero = "zero";
```

After the library declaration and the declaration of the *CTPSim* algorithm an additional *CTPTester* algorithm can be included which has not been treated in detail so far. The task of this algorithm is to extract from StoreGate the result of the CTP simulation, to perform simple tests on it and to print it. It is commented out because it is not foreseen for the usual CTP simulation running, but rather for debugging purposes.

There are two flags for the *CTPSim* algorithm: *ApplyDeadTime* ("yes" or "no") steers whether or not the CTP simulation masks out four events after each accepted event. This

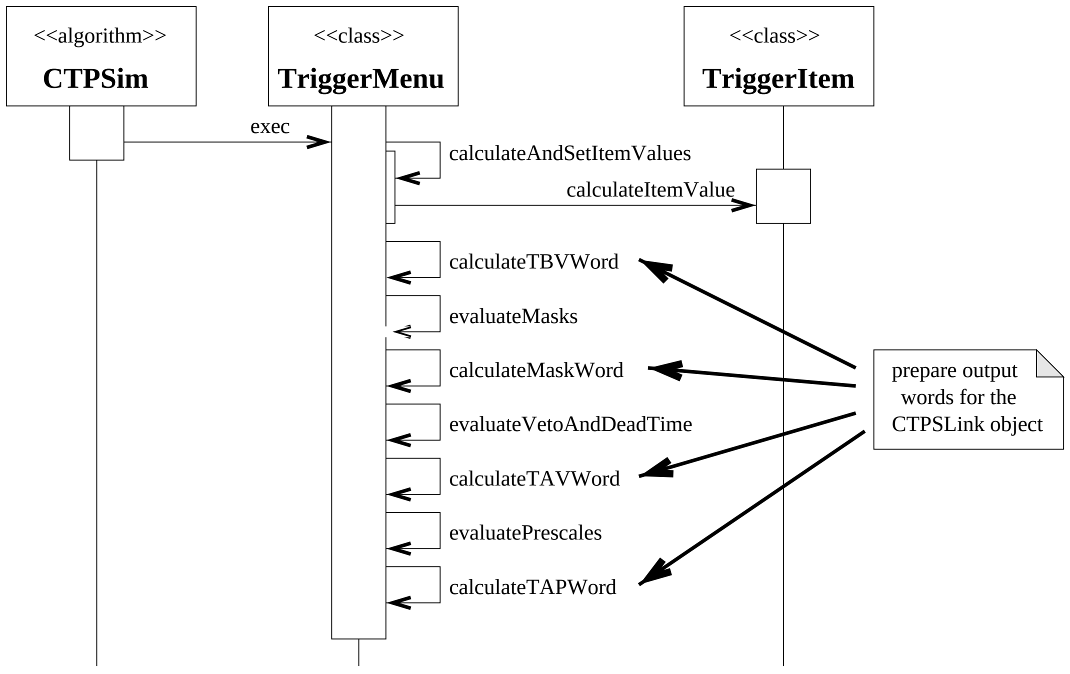


Figure 15: Sequence diagram for the *TriggerMenu::exec* step of the CTP simulation. See text for details.

is the simple dead-time mechanism implemented in the CTP demonstrator hardware[5]. The *RandomVersusZero* flag ("random" or "zero") decides whether missing input information to the CTP simulation is replaced with zeros or with random numbers. Default is "zero".

6 The LVL1 RDO

The raw data object (RDO) is the object view of the LVL1 result – as opposed to the bytestream format which cannot easily be interpreted by the human observer. There exist two versions of the RDO. The chronologically first RDO implementation was that of the raw RDO (the doubling-up of the word 'raw' is rather unfortunate), see Section 6.1. The newer implementation, and the one that is currently used for the seeding of the HLT, is the reconstructed (or interpreted) RDO, see Section 6.2.

6.1 The Raw RDO

The raw RDO is implemented using the classes of the TrigT1Result package, see Fig. 16, and follows closely the Slink structure connecting the different parts of the LVL1 trigger to the RoIB. There is a top-level class *RoIBResult* which contains the objects that correspond to the Slinks: one object *MuCTPIResult* for the MuCTPI result, one object *CTPResult* for the CTP

[5]There exists also a more complex dead-time algorithm which introduces dead-time as soon as there are more then N triggered events in an interval of M μs, with N and M programmable. This algorithm however cannot be implemented in the CTP simulation software.

Slink, four *EMTauResult* objects for the four Slinks from the calorimeter cluster processor to the RoIB, and two *JetEnergyResult* objects for the two Slinks from the jet/energy processor to the RoIB. In order to follow the data format of the hardware as closely as possible, the *XXXResult* objects have as members *Header* and *Trailer* objects which contain the data passed in the headers and trailers of the Slinks. The data format version chosen for the implementation is v2.2 [10].

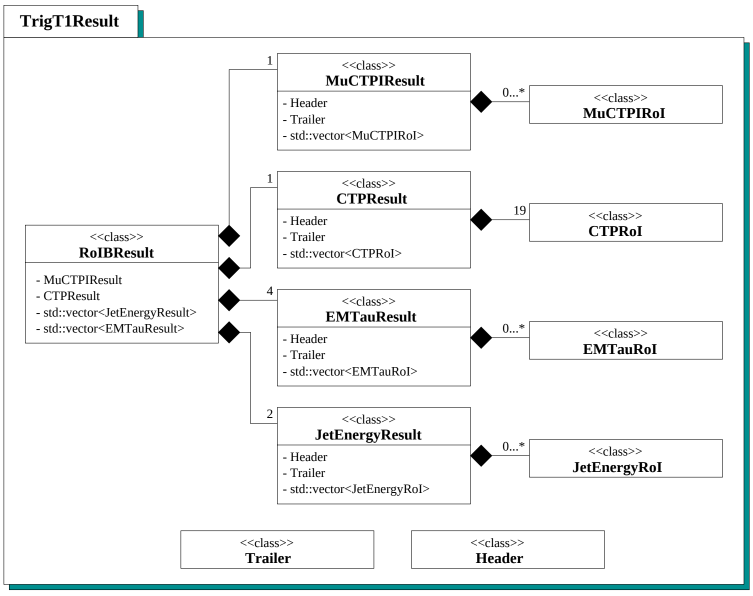


Figure 16: UML class diagram of the classes involved in the LVL1 raw RDO.

The actual data content of the *RoIBResult* object is in the *XXXRoI* classes (XXX=MuCTPI, CTP, EMTau, or JetEnergy) which contain (in case of the calorimeter and muon triggers) the 32-bit RoI words or (in case of the CTP) the corresponding information of the CTP processing. The number of RoIs clearly is not constant but depends on the event properties; only in case of the CTP is the RoI number defined and fixed to 19 32-bit words containing input bit patterns, masks, the LVL1 result and some more information.

The primary purpose of the *RoIBResult* class is to represent the LVL1 result after its assembly by the RoIB simulation (see Section 7). In this simulation, the various partial results from the muon and calorimeter triggers and from the CTP are mapped onto the *XXXResult* members of the *RoIBResult* instance created by the *RoIBuilder* algorithm (package TrigT1RoIB).

The raw RDO as implemented in the *RoIBResult* class was initially also used in the implementation of the HLT steering code. The steering code retrieved the RDO from StoreGate (where it had been stored by the *RoIBuilder* algorithm of the TrigT1RoIB package) and accessed the RoIs. However, the (raw) RoIs contain only 32-bit words indicating with bit patterns the thresholds that were passed or the location in the detector, whereas the steering code requires the RoI to be present in terms of the η and ϕ coordinates and the threshold value (in GeV). Therefore, some reconstruction code had to be run within the HLT steering code which also had to access the LVL1 configuration in order to give the necessary interpretations of the RoIs. This approach seemed inelegant and introduced unnecessary dependencies of the HLT code on the LVL1 trigger code. In order to remove these dependencies, the reconstructed RDO was introduced.

6.2 The Reconstructed RDO

The idea behind the reconstructed RDO as implemented in the *RecRoIBResult* class of the TrigT1Result package is that the interpretation of the RoIs necessary for the HLT steering code can more elegantly be done in the conversion step from bytestream, assuming that the input to the HLT process will always be bytestream, be it transient or persistent:

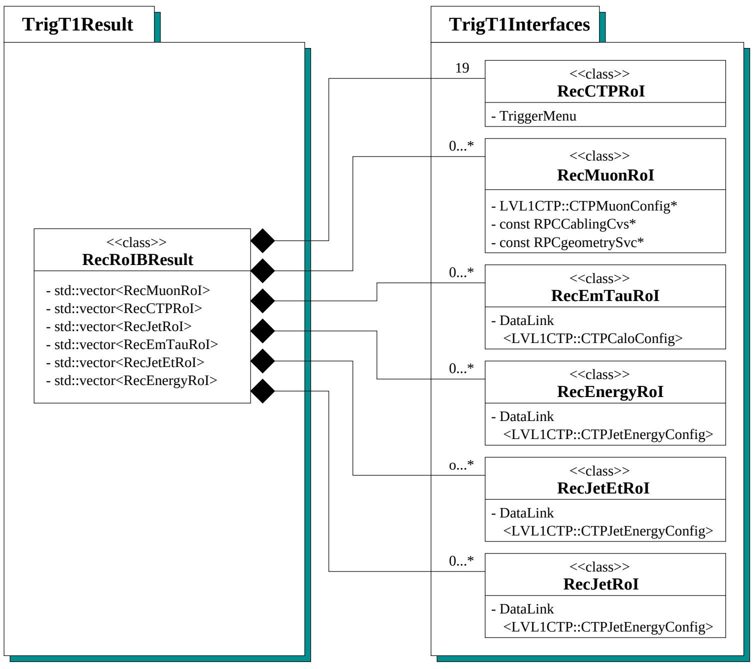


Figure 17: UML class diagram of the classes involved in the LVL1 reconstructed RDO.

The important difference to the *RoIBResult* class is that that the various RoIs contained in the *RecRoIBResult* object have access to configuration data, which in most cases is used during the construction of the object to fill data members representing η, ϕ and the threshold value or to calculate these quantities later on during program execution. In case of the calorimeter RoIs, the configuration is accessed through the *CTPCaloConfig* or *CTPJetEnergyConfig* objects which are a result of the configuration process. In case of the muon RoIs, the *CTPMuonConfig* object together with the relevant RPC services (cabling and geometry) have to be provided. The configuration data are used to give values to data members which represent the threshold value (in GeV), and the coordinates of the RoI in η–ϕ space.

It should be noted that currently (offline version 6.0.3) the class *RecRPCRoI* is not yet implemented, and that the *RecRoIBResult* only consists of vectors of *RecMuonRoI*, *RecEmTauRoI* and *RecJetRoI* class instances – the use of the *RecEnergyRoI* and *RecJetEtRoI* classes still has to be implemented.

7 The RoIB Simulation

7.1 The RoIBuilder Algorithm

The *RoIBuilder* algorithm collects the various pieces of LVL1 results from the calorimeter and muon trigger simulations and from the CTP simulation and creates an instance of the *RoIBResult* class from them. A detail of the process, taking into account only the MuCTPI part and neglecting the calorimeter and CTP Slinks, is shown in Fig. 18. It can be seen that first the object corresponding to the MuCTPI-to-RoIB Slink is retrieved from StoreGate (class *MuCTPIToRoIBSLink*), the constituting vector of unsigned integers is extracted and looped over, and for each integer corresponding to a muon RoI a *MuCTPIRoI* is created and pushed back into a vector. This vector, together with instances of the *Header* and *Trailer* class shown in Fig. 16, is used to create a *MuCTPIResult* object, which in turn is used to create the *RoIBResult* object.

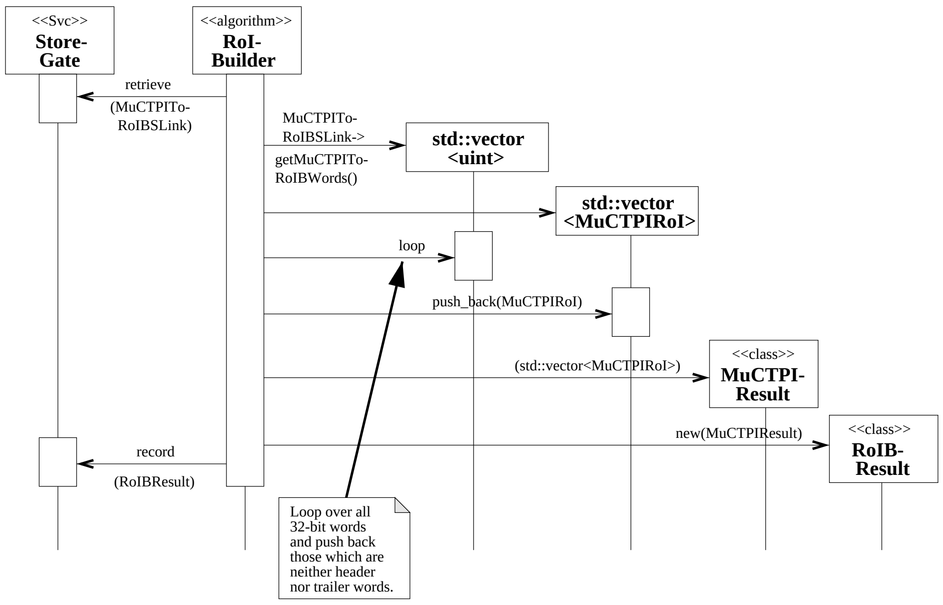


Figure 18: Sequence diagram for the *RoIBuilder* algorithm, specifying only the part which handles the MuCTPI part of the LVL1 result. The calorimeter and CTP results are treated in very similar ways.

In order to really build the *RoIBResult* object, also the partial LVL1 results for the six calorimeter Slinks and the one CTP Slink have to be created from objects which are similar, but not completely equal, to the *MuCTPIToRoIBSLink* object used for the MuCTPI Slink. Since the differences are only in naming conventions and data member names, only these other names will be given here (the corresponding classes are also shown in the UML diagram of Fig. 19):

- The CTP Slink comes through an object of type *CTPSLink*. From this, RoIs of type *CTPRoI* are filled, which in turn serve to create an object *CTPResult*.

- The calorimeter cluster processor information is transmitted on four Slinks which are simulated using four *DataVector<SlinkWord>* objects. From these, *EMTauRoI* objects are filled which serve to fill four *EMTauResult* objects.

- Similarly, the two jet/energy processor Slinks arrive at the RoIB simulation via two *DataVector<SlinkWord>* objects from which *JetEnergyRoI* are created which enter the two *JetEnergyResult* objects.

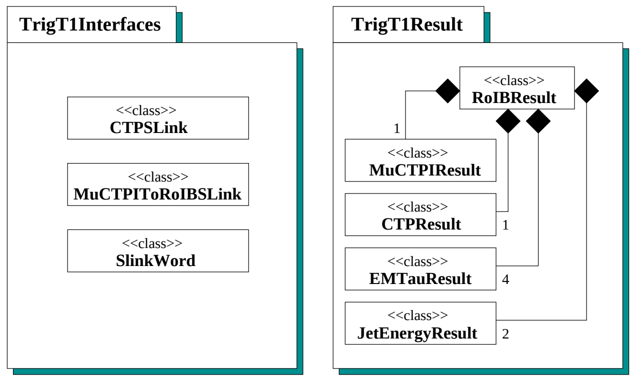


Figure 19: UML class diagram for the inputs to the RoIB simulation and the creation of the LVL1 RDO. See text for more details.

7.2 The Tester Algorithms

The two algorithms *RoIBTester* and *RecRoIBTester*, which are not included during normal LVL1 simulation, serve for debugging and testing purposes and for tests of the byte-stream (BS) conversion mechanism via the *ReadTrigT1BSExample_jobOptions.txt* file explained in detail in Section 9.4. Since the algorithms are fairly simple, they will not be explained explicitly here.

7.3 Job-Options Fragment

The standard job-options fragment for the RoIB simulation that is included by higher-level job-options files looks like the following:

```
//-----------------------------------------------------
// Trigt1RoIB Algo
//----------------------------------------------------
ApplicationMgr.DLLs += { "TrigT1RoIB" };
ApplicationMgr.TopAlg += { "ROIB::RoIBuilder/RoIBuilder" };
//ApplicationMgr.TopAlg += { "ROIB::RoIBTester/RoIBTester" };
```

It consists of the library include statement (‘ApplicationMgr.DLLs’) and declarations of the RoIBuilder and *RoIBTester* algorithms. The latter algorithm retrieves the LVL1 RDO that is built by the *RoIBuilder* algorithm from StoreGate, performs tests and prints it. For usual, non-debugging running it is not used, except for the job-options file *Read-TrigT1BSExample_jobOptions.txt* which exercises the reading of bytestream and is explained in Section 9.4.

8 The Bytestream Conversion

Figure 20 shows a schematic overview of the conversion process involved in transferring the LVL1 result to LVL2. The LVL1 result (object of type *RoIBResult*) is stored in Store-Gate. From there it is retrieved by the AlgTool *RoIBResultByteStreamTool* of the package TrigT1ResultByteStream. The tool has two *convert* methods, one to convert the raw RDO object to bytestream (implemented in the *RoIBResultByteStreamTool.h* file), and one to convert the BS back to the object of type *RoIBResult* (file *RoIBResultByteStreamTool.cxx*). The converters, the fragments of which were provided by H. Ma[6], use the *eformat* library and the definition of data formats version 2.2.0 [10].

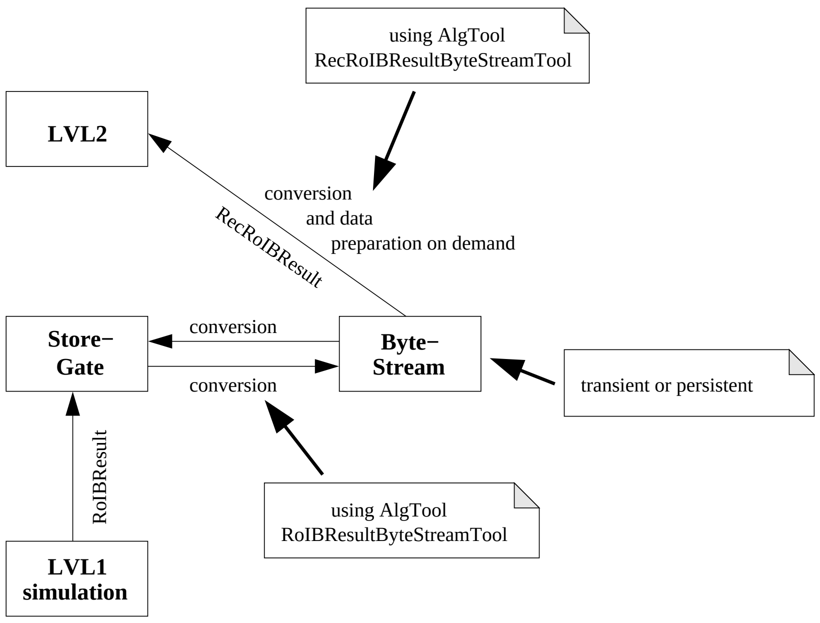


Figure 20: Overview of the conversion steps needed for the LVL1-LVL2 communication. See the text for details.

The second of the converters mentioned can be used to translate the BS (be it transient – in memory – or persistent – as a file on a disk) back to the raw LVL1 RDO. This is, however, not to be used anymore in the future, although it found wide use in the preparation phase towards the HLT Technical Design Report [3], where the recreated raw LVL1 RDO of type *RoIBResult* was used to seed the HLT steering process. In doing so, the raw RoIs (32-bit

[6]In case of questions of change requests concerning the converter package TrigT1ResultByteStream, please contact H. Ma or G. Comune who also has experience with conversion packages, in his case for the LVL2 result. The author of this note admits to never really have fully understood the concepts of the converters and the AlgTools used for implementing them.

words) had to be interpreted during the HLT steering process one by one – a procedure that was considered inelegant and time-consuming. In addition, it was felt that this way of doing it led to too much dependence of the HLT trigger software on LVL1 software.

The approach that was chosen instead is also shown in Fig. 20. It consists in a direct conversion of the BS to the reconstructed, instead of the raw, RDO. This conversion step is implemented in the *RecRoIBResultByteStreamTool* AlgTool in package TrigT1ResultByteStream. It requires the creation of reconstructed, rather than raw, RoIs (classes *RecMuonRoI* etc.), and therefore the use of LVL1 configuration data in the construction phase of the RDO (see Fig. 17 for an overview of the reconstructed RDO).

Section 9.4 will explain how to run a job that creates and reads back BS.

9 Installation and Running

In this section, various methods to use (parts of) the LVL1 simulation code are introduced.

9.1 Installation using an Offline Release

The simplest way to the make use of the LVL1 simulation, or of parts of it, is to not check out any specific packages from the CVS repository, but to completely rely on an offline release. The following explains this option, assuming running of the simulation chain

configuration → calorimeter trigger → CTP simulation → RoIB simulation → BS creation → reading of BS and RDO (re)creation.

Also the reading of the BS and its interpretation are discussed. The offline release used is 6.0.3, and we are using the TestRelease package to build the code.

First, the user has to make sure to use the correct offline release. This is achieved by two lines in the main requirements file:

```
macro ATLAS_DIST_AREA "/afs/cern.ch/atlas/software/dist"
macro ATLAS_RELEASE "6.0.3"
```

Doing (in the home directory)

```
%> source setup.sh
```

configures CMT and prepares the user to check out packages, to compile, and to run.

The TestRelease package has to be checked out in the private working area (which is assumed to be a directory ‘athena’ directly below the home directory of the user):

```
%> cd athena
%> cmt co TestRelease
```

The requirements file of the TestRelease package should contain the following ‘use’ statements, in addition to the usual content of the file:

```
use AtlasPolicy AtlasPolicy-*
use TrigT1Calo       TrigT1Calo-*        Trigger/TrigT1
use TrigT1Config     TrigT1Config-*      Trigger/TrigT1
use TrigT1CTP        TrigT1CTP-*         Trigger/TrigT1
use TrigT1RoIB       TrigT1RoIB-*        Trigger/TrigT1
use TrigT1Result     TrigT1Result-*         Trigger/TrigT1
use TrigT1ResultByteStream TrigT1ResultByteStream-* Trigger/TrigT1
use TrigT1Muctpi     TrigT1Muctpi-*         Trigger/TrigT1
use MagneticFieldAthena MagneticFieldAthena-* MagneticField
use xKalmanppAthena     xKalmanppAthena-* Reconstruction/xKalmanpp
use RecExCommon RecExCommon-* Reconstruction/RecExample
use GeneratorModules    GeneratorModules-* Generators
```

Change to the 'cmt' subdirectory of the TestRelease package, configure it, and compile everything:

```
%> cd ~/athena/TestRelease/TestRelease-*/cmt
%> cmt broadcast cmt config
%> source setup.sh
%> cmt broadcast "rm -rf ../i686*"
%> cmt broadcast gmake clean
%> cmt broadcast gmake
```

Section 9.4 will explain in more detail how to run the compiled code.

9.2 Installation using Single Packages

In many cases the user might want to develop a certain package and for this reason has to check out its HEAD version from the CVS repository (not specifying a specific version):

```
%> cmt co TrigT1CTP Trigger/TrigT1/TrigT1CTP
```

In case the user wants to edit a package that is not explicitely mentioned in the requirements file presented in the section above, it should be included via a use statement.

It may also happen that the user wants a specific version of a given package, but not the one that is used in the release. In this case, the version in question needs to be checked out using the '-r' option:

```
cmt co -r TrigT1CTP-AB-CD-XY Trigger/TrigT1/TrigT1CTP
```

where AB-CD-XY represents the specific tag the user is requiring. In this case the specific tag in question should also be entered into the requirements file to make sure CMT picks up the correct version of the code:

```
use TrigT1CTP TrigT1CTP-AB-CD-XY Trigger/TrigT1
```

In both cases mentioned above, the same steps as in the foregoing subsection are required to compile the code.

9.3 Installation as a Stand-Alone Program

At the time of offline release 6.0.3 there was no functioning stand-alone version of the LVL1 configuration and CTP simulation code. Nevertheless in this section the principles of getting a stand-alone version of the code will be demonstrated.

The *makefile* (under construction!) that is supposed to result in a stand-alone version of the trigger configuration and CTP simulation code is contained in the 'misc' directory of the TrigT1CTP package, together with *main.cxx* file needed for the stand-alone code. The important point about the *makefile* is that it does not try to use CMT and its facilities, but rather expects a checked-out version of the packages TrigT1Interfaces, TrigT1CTP and TrigT1Config. From these checked-out versions it tries to build an executable. In doing so, the use of the 'STANDALONE' compilation flag activates the use of other include files and different code sections in the implementation files than is the case for standard Athena running. As an example, an extract from the header file of the *TriggerCondition* class from the TrigT1Config package is shown:

```
#ifndef TRIGGERCONDITION_H
#define TRIGGERCONDITION_H
#include <vector>
#include <string>
#ifndef StandAlone
#include "dom/DOM.hpp"
#include <iostream>
#include <strstream>
#include "TrigStore/MessageSvcProvider.h"
#else
#include <dom/DOM.hpp>
#endif
namespace LVL1CTP
{
class TriggerCondition
{
public:
/* Constructor takes name and the XML DOM node */
TriggerCondition(std::string,int,DOM_Node&);
T̃riggerCondition();
...
#ifndef StandAlone
MessageSvcProvider m_messageSvcProvider;
std::string m_thisName;
#endif
};
} // End of namespace LVL1CTP bracket
#endif
```

The main point in having the compilation flag 'STANDALONE' is that the Athena concept of the MessageSvc cannot be used in stand-alone running and has to be replaced by standard 'cout' statements. In addition, the path to many include files, for example those needed for the XML parsing parts of the code, is different for Athena and stand-alone running.

In order to be able to use the stand-alone code, therefore, the packages mentioned above have to be checked out, and the *makefile* has to be adapted to the specific situation of the user in question. The required changes should be obvious. Before the *makefile* can be run with the usual 'gmake' command, the running environment has to be set up properly. The aim is to use the gcc compiler version used for the offline release 6.0.3, namely 2.95. The following script may be used to achieve the correct setup[7]:

```
#!/bin/zsh
echo ''Running runscript to set up for DOM examples.''
export XERCESCROOT= /4xml
export LD_LIBRARY_PATH=$XERCESCROOT/lib:$LD_LIBRARY_PATH
export PATH=$PATH:$XERCESCROOT/bin:$XERCESCROOT/standalone/bin
echo ''XERCESCROOT = '' $XERCESCROOT
echo ''LD_LIBRARY_PATH = '' $LD_LIBRARY_PATH
echo ''PATH = '' $PATH
echo ''Runscript to set up for DOM examples ready.''
```

Here, only the XERCESCROOT variable has to be adapted to the specific situation. In addition, a complete XERCESC setup in terms of the required XML XERCESC libraries has to be provided, see [9]. After running the *makefile* successfully, the stand-alone code can be run doing

```
%> CTPSim [menufilename] [thresholdsfilename] [muctpiconfigfile]
```

The program will then start using XML configuration files specified in the arguments to the 'CTPSim' command. These files have to be provided in the 'run' directory.

The goal of running the configuration and CTP simulation stand-alone, outside of the computing framework Athena, clearly is to provide a possibility for thorough tests of the CTP hardware and to run the simulation and the hardware with the same configuration and input data and thus cross-check the two against each other. This is not yet achieved. The steps to be taken in order to arrive at this goal are the following:

1. Get the configuration and CTP simulation running in stand-alone mode as described above on lxplus.

2. Port the code to the Dsy-Srv on which the code running and steering the CTP hardware can be found (contact: R. Spiwoks).

3. Use the LUT configuration files and the VHDL code files generated by the configuration step of the stand-alone running to configure the CTP hardware. This step also involves translating the VHDL code into CPLD configuration files, a step which has to be done very carefully in order to avoid later hardware damages due to faulty configurations.

[7]The correct compiler version gcc-2.95 can be selected by executing the 'source setup.sh' in the home directory with offline release 6.0.3 selected in the requirements file.

4. Provide, for example via plain asci files, input data to the simulation and to the CTP hardware.

5. Compare the outputs of the simulation and of the hardware.

None of the above mentioned steps is trivial, and they all have to be done in close collaboration with the people working on the CTP or CTPD hardware, especially R. Spiwoks.

9.4 Running the Simulation

In this section, various simple job-option files that exercise the LVL1 simulation are discussed. In all cases, only the calorimeter branch of the LVL1 trigger is considered, with the extensions necessary to include also the muon RPC (and later TGC) simulations being straightforward and obvious.

9.4.1 Running the LVL1 Simulation without BS

The file General_jobOptions.txt (from the 'share' directory of the TrigT1Config package) is given in Appendix 10.1. This file, after some initialisation and setting up Athena for reading LVL1 TDR data in Zebra format, includes the three lines:

```
#include "$TRIGT1CONFIGROOT/share/TrigT1ConfigJobOptions.txt"
#include "$TRIGT1CONFIGROOT/share/L1Sim_CaloSetup_jobOptions.txt"
#include "$TRIGT1CONFIGROOT/share/L1Sim_jobOptions.txt"
```

The first line includes the job options for the setup of the LVL1 trigger configuration. The second file is included in order to prepare the calorimeter trigger simulation, and the third line calls the algorithms that actually perform the simulations of the calorimeter (and later also muon) trigger, and of the CTP and RoIB simulations:

```
//-----------------------------------------------------------
// RPC stuff
//-----------------------------------------------------------
//#include "$TRIGT1RPCSTEERINGROOT/share/TrigT1RpcJobOptions.txt"

//-----------------------------------------------------------
// TrigT1Muctpi Algorithms Private Options
//-----------------------------------------------------------
//#include "$TRIGT1MUCTPIROOT/share/TrigT1Muctpi_jobOptions.txt"

//----------------------------------------------------
// TrigT1Calo Algos
//----------------------------------------------------
#include "$TRIGT1CALOROOT/share/TrigT1CaloJobOptions.txt"

//----------------------------------------------------
// TrigT1CTP Algos
//----------------------------------------------------
```

```
#include "$TRIGT1CTPROOT/share/TrigT1CTPJobOptions.txt"

//--------------------------------------------------------
// TrigT1RoIB Algos
//--------------------------------------------------------
#include "$TRIGT1ROIBROOT/share/TrigT1RoIBJobOptions.txt"
```

The command to start the simulation is simply:

```
%> athena General_jobOptions.txt
```

9.4.2 Running the LVL1 Simulation and Creating the LVL1 Result BS

A more complicated exercise is to produce the LVL1 bytestream (BS) and write it out to a file (which is typically called *RawEvent.re*). This can be achieved using the following job-options file *WriteTrigT1BSExample_jobOptions.txt* from the 'share' directory of the TrigT1ResultByteStream package:

```
#include "$ATHENACOMMONROOT/share/Atlas_ZebraTDR.UnixStandardJob.txt"
#include "$BYTESTREAMCNVSVCROOT/share/WriteByteStream_jobOptions.txt"
#include "$TRIGT1RESULTBYTESTREAMROOT/share/WriteTrigT1ResultBS_jobOptions.txt"
EventSelector.directConversion = true;
EventSelector.readHits = false;
EventSelector.readDigits = false;
EventSelector.calos = false;
EventSelector.muons = false;
EventSelector.trt = false;
EventSelector.sct = false;
EventSelector.pixel = false;
EventSelector.mdt = false;
EventSelector.rpc = false;
EventSelector.tgc = false;
ApplicationMgr.DLLs +=  "GaudiAud" ;
AuditorSvc.Auditors +=  "ChronoAuditor" ;
AuditorSvc.Auditors +=  "MemStatAuditor" ;
MemStatAuditor.OutputLevel = 4;
MessageSvc.OutputLevel = 2;
ApplicationMgr.EvtMax = 10;
ApplicationMgr.EvtSel = "FILE ZEBRA.P";
```

This in turn calls the file *WriteTrigT1Result_jobOptions.txt*:

```
#include "$TRIGT1CONFIGROOT/share/TrigT1ConfigJobOptions.txt"
#include "$TRIGT1CONFIGROOT/share/L1Sim_CaloSetup_jobOptions.txt"
#include "$TRIGT1CONFIGROOT/share/L1Sim_jobOptions.txt"
ApplicationMgr.DLLs +=  "TrigT1ResultByteStream" ;
StreamBS.ItemList +=  "6000#*" ;
StreamBS.RequireAlgs += "RoIBuilder";
```

The command to start the simulation now is:

```
%> athena WriteTrigT1BSExample_jobOptions.txt
```

9.4.3 Reading back the BS

In order to check and test the content of the BS file RawEvent.re that was written using the job options given in the preceeding section, the following piece of job options can be used (file *ReadTrigT1BSExample_jobOptions.txt* from the TrigT1ResultByteStream/share directory):

```
#include "$BYTESTREAMCNVSVCROOT/share/ByteStreamSelector_jobOptions.txt"
#include "$TRIGT1RESULTBYTESTREAMROOT/share/ReadTrigT1ResultBS_jobOptions.txt"
#include "$TRIGT1RESULTBYTESTREAMROOT/share/ReadRecTrigT1ResultBS_jobOptions.txt"
MessageSvc.OutputLevel = 2;
ApplicationMgr.EvtMax = 20;
```

The included file *ReadTrigT1ResultBS_jobOptions.txt* recreates the raw RDO from the BS (see Section 6.1), and the second file *ReadRecTrigT1ResultBS_jobOptions.txt* aims at creating the reconstructed RDO (Section 6.2) which serves to seed the HLT steering software.

The result of recreating the raw RDO from BS is tested in the above job options via the running of the *RoIBTester* algorithm. An example for the output of this algorithm is shown in Appendix C (taking into account the fact that the actual data content of the *RoIBResult* structure depends on the configuration and on the data file that were used in creating the BS file). The output provided by the creation of the reconstructed RDO depends very much on the kind of RoIs found in the BS file and will not be shown here explicitly.

9.5 Typical Configuration and Simulation Output

Depending on the output level chosen for the Athena MessageSvc, more or less output will be given for the LVL1 configuration and the simulations of the CTP and the RoIB. Moreover, the output clearly depends on the configuration chosen and (for the simulation) on the data file used. However, a typical job should always produce lines similar to the ones shown in Appendix D for the LVL1 configuration step during the initialisation phase of the job. Also shown should be the initialisation messages of the *CTPSim* and *RoIBuilder* algorithms:

```
CTPSim                INFO ========================================
CTPSim                INFO Initialisation for CTPSim algorithm.
CTPSim                INFO ========================================
CTPSim                INFO
RoIBuilder            INFO
RoIBuilder            INFO ========================================
RoIBuilder            INFO Initialisation for RoIBuilder algorithm.
RoIBuilder            INFO ========================================
RoIBuilder            INFO
```

In addition, a typical event with output level lower than default should show lines corresponding to the ones shown in Appendix E, where of course the numbers given there again depend on the actual configuration and data file that was chosen.

10 Things To Do

This section contains several lists of things to be done in order to improve the overall LVL1 trigger simulation or the parts of it which are treated in this note. First, Subsection 10.1 assembles points of general importance to the simulation. Then, Subsections 10.2 through 10.7 concentrate on the various single packages of the simulation and things to be done for them.

10.1 General issues

- Adapt LVL1 simulation to offline release 6.1.0 and following, taking into account change of compiler from gcc 2.95 to 3.2. This also implies use of a different Xerces-C version 2.2.0. Note: There was a discussion recently in ATLAS about the need of DOM and about replacing it by other parsers such as SAX, etc. C. Arnault is the right person to ask.
- Introduce use of secondary RoIs in trigger configuration and CTP simulation. A first simple implementation might be achieved by setting the required multiplicity of the supposed secondary RoI to a very high number (say 9999). In the future, a flag in the <TriggerThreshold> XML tag (an attribute 'RoI' with values "primary" or "secondary" might be necessary. This would then require asking the trigger thresholds in the CTP simulation whether they are primary or not, and taking only the primary ones into account in the trigger decision.
- In close collaboration with the calorimeter and muon trigger software developers the complete simulation should be tested for the complete data flow chain. This has happened for most parts of the calorimeter trigger, and the test procedure showed several bugs which were then repaired. However, especially the jet and energy triggers and all of the muon trigger simulation is not yet fully tested in the combination with the CTP and RoIB simulations.
- Strict rule checking, at least for the 'required' ATLAS coding rules, should be applied. This has not been done thoroughly so far, and lots of rule violations are present. In addition to this a number of compilation warnings (mainly of the 'discards qualifier' type are present in most of the packages, possibly hinting at design flaws.
- The core parts of the LVL1 simulation have been tested for memory leaks only using the 'top' method – i.e. running over 100 or 200 events and looking whether the memory consumption is rising. This gave satisfactory results. However, a more thorough investigation using a tool like *valgrind* is necessary.
- The whole simulation suffers from the use of hardcoded numbers in a lot of places. It would be very valuable to go through the code and replace them by static variables that are defined in one common location, probably in the TrigT1Interfaces package.
- The code does not use exceptions, and also return codes are not used broadly throughout the simulation. This might be changed in order to get a better error control.

- The combined running of the calorimeter and muon RPC trigger simulations still has to be achieved, as has the integration of the muon TGC simulation.
- The standalone version of the configuration and CTP simulation code is not finished yet, see Section 9.3.
- The use of the *MessageSvcProvider* as provided by the TrigStore package is not fully understood; in particular it seems difficult or impossible to set the output level. Discuss with Simon George, the author of the thing. The *MessageSvcProvider* is currently used in the TrigT1Config package. There are some places in classes of the TrigT1Interfaces package where its use might also be appropriate.

10.2 Details for TrigT1Interfaces

- The data member 'thresholdHA2Isol' of the *EMTauTriggerThresholdValue* class should be renamed to something like 'thresholdHAVeto' (see Section 10.3).
- The naming convention for the trigger threshold types (data member 'thresholdType' of the *TriggerThreshold* class and attribute 'type' of the <TriggerThreshold> tags) do not follow the scheme adopted in the LVL1 TDR [1].
- The *TMUtil* class should be checked thoroughly for uncatched errors, etc. In addition, it should be synchronised with the *TMUtil* class in the *TrigSteer/TrigConfig* package (the better version of the code is probably in *TrigT1/TrigT1interfaces*).
- In analogy to the *RecMuonRoI* etc. a class *RecCTPRoI* is needed in order to have the energy and CTP information available for the reconstructed RDO. For the *RecCTPRoI* class methods should be provided that extract, using the LVL1 configuration, a list of fired triggers items, the multiplicities of all trigger thresholds, and the LVL1 accept signal.

10.3 Details for TrigT1Config

- The attribute 'haisol2' of the <TriggerThresholdValue> tag in the trigger thresholds XML file should rather be named 'haveto' since it defines the maximum amount of energy allowed in the hadronic section of the calorimeters behind an electromagnetic candidate. This change would have to be reflected in the *TriggerMenu::mapThresholds()* method. Consequently, also the corresponding data member of the *EMTauTriggerThresholdValue* class should be renamed from 'thresholdHA2Isol' to something like 'thresholdHAVeto'.
- The naming convention for the trigger threshold types (data member 'thresholdType' of the *TriggerThreshold* class and attribute 'type' of the <TriggerThreshold> tags) do not follow the scheme adopted in the LVL1 TDR [1]. Changing this would affect also several methods of the *TriggerMenu*, *CTPHardware* and *Hardware* classes.

- In the *TriggerMenu::checkTriggerMenu* method, make sure that
  - the up to six muon thresholds that are defined in the trigger thresholds XML file really have monotonically rising threshold values (this is necessary for the interpretation of the data);
  - the multiplicity delivered for energy thresholds is less than two (an energy threshold is either passed or not);
  - the number of bits used for forward jet thresholds ('FL', 'FR') is 2, and the number of bits used for energy thresholds is one (this test might also go somewhere in the *Hardware* class).
- From a design point of view one might want to consider creating classes for the <PIT>, <MIO> and <TBV> tags of the CTP hardware configuration XML file. This might possibly make the code cleaner, especially the internal algorithmic part of the *LUT* and *CMB* classes.
- The 'priority' attribute of the <TriggerItem> tag in the trigger menu XML files is currently not yet used. The purpose of this flag, if set to 'high' instead of 'low', would be to override, under certain circumstances, the CTP dead-time algorithms. Such an override mechanism would have to be implemented in the *TriggerMenu* class. This has, however, no high priority since the simple dead-time mechanism that is built into the CTP simulation is usually not used (see Section 5.3 for a discussion of this mechanism).
- For some purposes it might be helpful to have somewhere (maybe in *TriggerMenu*?) a map that relates the integer number of a trigger threshold (one to six for the muon trigger, one to 16 for the electron/photon trigger, etc.) to the trigger threshold objects. This would facilitate the translation of the bits indicating the number of the passed threshold in the RoIs to the threshold value, which is accessible via the trigger threshold object.
- Another design change would be to avoid the double definition of <TriggerCondition> tags in the trigger menu XML files by having in the <TriggerItem> tags not the <TriggerCondition> tags, but only references to them, and only after the definition of all trigger items define all trigger conditions. This, however, probably makes it more complicated to define complex logical structures because it is not a priori clear how to introduce tags for the logical operations AND, OR, NOT.
- It would be cleaner to try to get rid of the third argument in the call to the method *TriggerItem::calculateItemValue*. This third argument is a reference to the trigger conditions vector, but this clearly is a member of the trigger item anyhow and might always be accessed recursively by walking the XML tree. The proposed change would affect the *TriggerMenu.cxx* and *TriggerItem.cxx* files in the TrigT1Config package.
- In analogy to the *RecMuonRoI* class (and other similar classes, classes *RecCTPRoI* and *RecEnergyRoI* are needed in order to have the energy and CTP information available for the reconstructed RDO. These classes should be implemented in the TrigT1Interfaces

package, and it has to be agreed with the calorimeter trigger people who is responsible for the *RecEnergyRoI* class and its content. See also Section 10.5.

- Currently the connection between *TriggerMenu* and the various *TriggerThresholds* and between *TriggerThresholds* and the various *TriggerThresholdValues* is done using pointer. On the other hand, the *TriggerMenu* holds the *TriggerItems* directly, as does a *TriggerItem* with the respective *TriggerConditions*. One might consider using DataLinks here – I never bothered about these, and things work, but I guess they would be safer and more elegant. E. Moyse has some experience with them from his *TrigT1Calo* work.

- There is currently no real connection implemented between instances of the *TriggerCondition* class and the *TriggerThreshold* instances that it is supposed to discriminate - the connection runs, using many *for*-loops, over the one *TriggerMenu* instance. Maybe this is not the best way to do it. On the other hand, introducing additional dependencies and links makes the thing even more complicated ...

- In order to use the code in a stand-alone version and together with the CTP or CTPD hardware, the configuration code needs to generate the hardware configuration files (LUT files and VHDL code for the programmable devices). This should be achieved using the 'ofstream' command.

- Currently, in order to get the threshold value for a given threshold bit, a loop over all thresholds in the trigger menu threshold map or in the vector of trigger thresholds in the *CTPCaloConfig* and the other configuration objects has to be performed. This is time-consuming, so it might be better to provide objects which contain only threshold of one class (say, only forward-jet thresholds).

10.4 Details for TrigT1CTP

- I had once the impression that the first bit of the jet words delivered from the the calorimeter trigger simulation to the CTP is wrong, having a '1' where it should have a '0'. Check with the TrigT1Calo responsible!

- Simulation of the read-out data path, and provision of the respective BS conversion. Simple because the data content and format is exactly that of the data that the CTP sends also to the RoIB.

10.5 Details for TrigT1Result

- In order for the LVL2 code to run properly, the L1ID must be contained in the RDO or equivalent data. This is already the case for the raw RDO (class RoIBResult), but not for the reconstructed RDO, class *RecRoIBResult*. In this class the L1ID has to be introduced as a data member somehow.

- In order to allow the use of the CTP and energy information in LVL2, additional members have to be introduced. See also Sections 10.3 and 10.2.
- The use of the *RecCTPRoI*, *RecEnergyRoI* and *RecJetEtRoI* classes in the *RecRoIBResult* object has to be implemented.

10.6 Details for TrigT1RoIB

- In order to have a successful test of the reconstructed RDO in the class *RecRoIBTester* all necessary methods to extract threshold values (in GeV) and similar information from the RecXXXRoI classes have to be defined. This is the responsibility of the TrigT1Calo and TrigT1RPC/TGC authors, but one should check with them and try to use the newest and most complete version of these methods to do the tests in *RecRoIBTester* (currently the concern is mainly about extracting the energy threshold for the total (missing) E_T and for the summed jet E_T).
- In the *RecRoIBTester* method (and in general in any method that aims at using the reconstructed RDO) it has to be made sure that the configuration used for interpretation is the same as the one used for creating of the BS. In particular one has to be careful not to try to access trigger thresholds that are not provided by the interpreting configuration – this leads to a crash. Therefore, a test should be introduced – something like (if(passedThreshold.size() > CTPJetEnergyConfif.size()) then warning (and no crash) or similar.

10.7 Details for TrigT1ResultByteStream

- Provide a RMS byte stream converter for the use in the test beds. Contact Werner Wiedenmann about this.

10.8 Details for the Stand-alone Version of the Code

See Subsection 9.3.

Acknowledgements

I would like to thank all my colleagues who have worked with me on the LVL1 simulation to make this a successful project, in particular N. Ellis for his reading of the manuscript of this note and E. Moyse who made a huge effort on the calorimeter trigger simulation and with whom I could discuss many technical items which I would not have been able to solve alone.

References

[1] ATLAS Collaboration, **First-Level Trigger Technical Design Report**, CERN/LHCC/98-14.

[2] E. Moyse, **TrigT1Calo**, http://hepwww.ph.qmul.ac.uk/ moyse/atrig/index.php;
E. Moyse, PhD thesis to be submitted to University of London;
E. Moyse and A. Watson, **Performance and Validation of TrigT1Calo, the Offline Level-1 Calorimeter Trigger Simulation**, ATL-COM-DAQ-2003-010. A. di Mattia, **The LVL1 offline simulation of the muon RPC trigger**, documentation to be provided;
M. Ishino, **The LVL1 offline simulation of the muon TGC trigger**, documentation to be provided;
T. Wengler, **The LVL1 offline simulation of the MuCTPI**, documentation to be provided.

[3] ATLAS Collaboration, **High-Level Trigger, Data Acquisition and Controls Technical Design Report**, CERN/LHCC/2003-022.

[4] A. Watson, **Updates to the Level-1 e/gamma and tau/hadron Algorithms**, ATL-DAQ-2000-046.

[5] ATLAS Collaboration, **Muon Spectrometer Technical Design Report**, CERN/LHCC/97-22;
A. D Mattia and L. Luminari, **Performance of the Level-1 Trigger System in the ATLAS Muon Barrel Spectrometer**, ATL-DAQ-2002-008;
http://atlas.web.cern.ch/Atlas/GROUPS/MUON/layout/muon_layout_P.html.

[6] http://atlas.web.cern.ch/Atlas/GROUPS/SOFTWARE/OO/architecture/General/index.html

[7] T. Schörner-Sadenius and T. Wengler, **Formats and Interfaces in the Simulation of the ATLAS First Level Trigger**, ATL-DA-ES-0029.

[8] M. Abolins *et al.*, **Specification of the LVL1/LVL2 trigger interface**, ATL-D-ES-0003.

[9] http://xml.apache.org/xerces-c

[10] C. Bee *et al.*, **The raw event format in the ATLAS Trigger & DAQ**, ATL-DAQ-98-129, v2.2.

A Examples for XML Configuration Files

In this appendix, a consistent set of trigger menu and trigger threshold files for the very simple case of only two EM thresholds defined is shown, together with the corresponding DTD files.

A.1 Example for a trigger menu file

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<!DOCTYPE TriggerMenu SYSTEM "trigger.dtd" [
   <!ENTITY % entities SYSTEM "entities.dtd">
   %entities;
]>

<TriggerMenu TM_ID="lowlumi">

   <TriggerItem TI_ID="iEM1" mask="on" priority="low" prescale="1">
         <TriggerCondition triggerthreshold="EM01" mult="2" />
   </TriggerItem>

   <TriggerItem TI_ID="iEM2" mask="on" priority="low" prescale="1">
         <TriggerCondition triggerthreshold="EM02" mult="1" />
   </TriggerItem>

</TriggerMenu>
```

A.2 Example for a trigger threshold file

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE TriggerThresholdList SYSTEM "trigger2.dtd">

<!-- @version: -->

<TriggerThresholdList>

<TriggerThreshold name="EM01" type="EM" bitnum="3">
   <TriggerThresholdValue thresholdval="10" emisolation="4"
                           haisolation1="2" haisolation2="2"
                           phimin="0" phimax="360" etamin="-5" etamax="5" />
</TriggerThreshold>
<TriggerThreshold name="EM02" type="EM" bitnum="3">
    <TriggerThresholdValue thresholdval="15" emisolation="4"
                           haisolation1="2" haisolation2="2"
                           phimin="0" phimax="360" etamin="-5" etamax="5" />
</TriggerThreshold>

</TriggerThresholdList>
```

A.3 The *trigger.dtd* file

```
<?xml version="1.0" encoding="UTF-8"?>

<!ELEMENT TriggerMenu (TriggerItem)+>
<!ATTLIST TriggerMenu
     TM_ID  %menu_ID;>

<!ELEMENT  TriggerItem (AND|OR|NOT|TriggerCondition)>
<!ATTLIST  TriggerItem
     TI_ID        %item_ID;
     priority     %priority_value;
     mask         %mask_value;
     prescale     %prescale_value;>

<!ELEMENT AND (AND|OR|NOT|TriggerCondition)+>

<!ELEMENT OR (AND|OR|NOT|TriggerCondition)+>

<!ELEMENT NOT (AND|OR|NOT|TriggerCondition)>

<!ELEMENT TriggerCondition EMPTY>
<!ATTLIST TriggerCondition
     triggerthreshold %threshold_choice;
     mult %multiplicity_value;>
```

A.4 The *entities.dtd* file

```
   <?xml version="1.0" encoding="UTF-8"?>
<!ENTITY % priority_value '(low | high) "low"'>
<!ENTITY % mask_value '(on | off) "off"'>
<!ENTITY % prescale_value 'CDATA "10000"'>
<!ENTITY % threshold_choice 'CDATA #REQUIRED'>
<!ENTITY % multiplicity_value 'CDATA #REQUIRED'>
<!ENTITY % menu_ID 'ID #REQUIRED'>
<!ENTITY % item_ID 'ID #REQUIRED'>
```

A.5 The *trigger2.dtd* file

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- @version: -->

<!ELEMENT TriggerThresholdList (TriggerThreshold)+>

<!ELEMENT TriggerThreshold (TriggerThresholdValue)*>
<!ATTLIST TriggerThreshold
     name         ID        #REQUIRED
     type         CDATA     #REQUIRED
     bitnum       CDATA     #REQUIRED
     OPL          CDATA     "NO"
     confirm      CDATA     "0"       >

<!ELEMENT TriggerThresholdValue EMPTY>
<!ATTLIST TriggerThresholdValue
     thresholdval CDATA     #REQUIRED
     emisolation  CDATA     "5"
     haisolation1 CDATA     "10"
     haisolation2 CDATA     "20"
     window       CDATA     "2"
     phimin       CDATA     #REQUIRED
     phimax       CDATA     #REQUIRED
     etamin       CDATA     #REQUIRED
     etamax       CDATA     #REQUIRED >
```

A.6 The *muctpi.dtd* file

```
<?xml version="1.0" encoding="UTF-8"?>

<!ELEMENT MuctpiConfig (MUCTPI)>

<!ELEMENT MUCTPI EMPTY>
<!ATTLIST MUCTPI
     firstMin  CDATA #REQUIRED
     secondMin  CDATA #REQUIRED
     numCand  CDATA #REQUIRED
>
```

A.7 The CTP hardware XML files

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<!DOCTYPE HARDWARE SYSTEM "hardware.dtd" >

<HARDWARE configuration_id="TEST_1">

<LUT lut_id="LUT_1">
    <PIT range_begin="0" range_end="17" />
    <MIO range_begin="0" range_end="7" />
</LUT>

<LUT lut_id="LUT_2">
    <PIT range_begin="18" range_end="35" />
    <MIO range_begin="8" range_end="15" />
</LUT>

<CMB cmb_id="CMB_1">
    <MIO range_begin="0" range_end="15" />
    <TBV range_begin="0" range_end="15"/>
</CMB>

<CMB cmb_id="CMB_2">
```

```
    <MIO range_begin="0" range_end="15" />
    <TBV range_begin="16" range_end="31"/>
</CMB>

</HARDWARE>
```

A.8 The *hardware.dtd* file

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- The global element is the hardware itself. -->
<!ELEMENT HARDWARE (LUT*,CMB+)>
<!ATTLIST HARDWARE
    configuration_id   ID      #REQUIRED >

<!-- These are the physical lines (cables etc.).
     We take ranges to which we assign several MULTIPLICITYs. -->
<!ELEMENT PIT EMPTY>
<!ATTLIST PIT
    range_begin        CDATA   #REQUIRED
    range_end          CDATA   #REQUIRED >

<!-- We need LUTs. -->
<!ELEMENT LUT (PIT+,MIO)>
<!ATTLIST LUT
    lut_id             ID      #REQUIRED >

<!-- We need lines connecting LUTs and CMBs. -->
<!ELEMENT MIO EMPTY>
<!ATTLIST MIO
    range_begin        CDATA   #REQUIRED
    range_end          CDATA   #REQUIRED >

<!-- We need combinational devices. -->
<!ELEMENT CMB ((PIT|MIO)+,TBV+)>
<!ATTLIST CMB
    cmb_id             ID      #REQUIRED >

<!-- We need output lines from the CMBs. -->
<!ELEMENT TBV EMPTY>
<!ATTLIST TBV
    range_begin        CDATA      #REQUIRED
    range_end          CDATA      #REQUIRED >
```

B The *General_jobOptions.txt* file

```
   #include "$ATHENACOMMONROOT/share/Atlas_ZebraTDR.UnixStandardJob.txt"
ApplicationMgr.EvtSel = "FILE ZEBRA.P";
EventSelector.runs = 1, 30000;
EventSelector.calos = false;
EventSelector.muons = false;
EventSelector.trt = false;
EventSelector.sct = false;
EventSelector.pixel = false;
// Use auditors ApplicationMgr.DLLs +=  "GaudiAud" ;
AuditorSvc.Auditors +=  "ChronoAuditor";
AuditorSvc.Auditors +=  "MemStatAuditor" ;
MemStatAuditor.OutputLevel = 4 ;
MessageSvc.OutputLevel = 2;
ApplicationMgr.EvtMax = 10;
// Include the L1 simulation
#include "$TRIGT1CONFIGROOT/share/TrigT1ConfigJobOptions.txt"
#include "$TRIGT1CONFIGROOT/share/L1Sim_CaloSetup_jobOptions.txt"
#include "$TRIGT1CONFIGROOT/share/L1Sim_jobOptions.txt"
```

C Output of the *RoIBTester* Algorithm

In this section an example is given for the output created by the *RoIBTester* method in recreating the raw RDO (class *RoIBResult*) from a BS file. In the example given here, a single electromagnetic RoI a30003 is found which caused the event to be triggered. In addition, in the first of the two jet/energy Slinks, some transverse energy is reported (RoIs 30000007, 3400000e, 38000051, see [7] for an interpretation of the RoIs):

```
RoIBTester          DEBUG ---- new CTP slink ----
RoIBTester          DEBUG ee1234ee
RoIBTester          DEBUG 8
RoIBTester          DEBUG 2020000
RoIBTester          DEBUG 7400
RoIBTester          DEBUG 4
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG RoI = 9
RoIBTester          DEBUG RoI = 0
RoIBTester          DEBUG RoI = 0
RoIBTester          DEBUG RoI = 0
RoIBTester          DEBUG RoI = 0
RoIBTester          DEBUG RoI = 0
RoIBTester          DEBUG RoI = 2
RoIBTester          DEBUG RoI = 0
RoIBTester          DEBUG RoI = 0
RoIBTester          DEBUG RoI = 2
RoIBTester          DEBUG RoI = 0
RoIBTester          DEBUG RoI = 0
RoIBTester          DEBUG RoI = 2
RoIBTester          DEBUG RoI = 0
RoIBTester          DEBUG RoI = 0
RoIBTester          DEBUG RoI = 1
RoIBTester          DEBUG RoI = 3
RoIBTester          DEBUG RoI = 0
RoIBTester          DEBUG RoI = 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 2
RoIBTester          DEBUG 19
RoIBTester          DEBUG 1
RoIBTester          DEBUG ---- new egamma slink ----
RoIBTester          DEBUG ee1234ee
RoIBTester          DEBUG 8
RoIBTester          DEBUG 2020000
RoIBTester          DEBUG 7200
RoIBTester          DEBUG 4
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG RoI = a30003
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 2
RoIBTester          DEBUG 1
RoIBTester          DEBUG 1
RoIBTester          DEBUG ---- new egamma slink ----
RoIBTester          DEBUG ee1234ee
RoIBTester          DEBUG 8
RoIBTester          DEBUG 2020000
RoIBTester          DEBUG 7201
RoIBTester          DEBUG 4
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 2
RoIBTester          DEBUG 0
RoIBTester          DEBUG 1
RoIBTester          DEBUG ---- new egamma slink ----
RoIBTester          DEBUG ee1234ee
RoIBTester          DEBUG 8
RoIBTester          DEBUG 2020000
RoIBTester          DEBUG 7202
RoIBTester          DEBUG 4
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 2
RoIBTester          DEBUG 0
RoIBTester          DEBUG 1
RoIBTester          DEBUG ---- new egamma slink ----
RoIBTester          DEBUG ee1234ee
RoIBTester          DEBUG 8
RoIBTester          DEBUG 2020000
RoIBTester          DEBUG 7203
RoIBTester          DEBUG 4
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 2
RoIBTester          DEBUG 0
RoIBTester          DEBUG 1
RoIBTester          DEBUG ---- new jetenergy slink ----
RoIBTester          DEBUG ee1234ee
RoIBTester          DEBUG 8
RoIBTester          DEBUG 2020000
RoIBTester          DEBUG 7300
RoIBTester          DEBUG 4
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG RoI = 30000007
RoIBTester          DEBUG RoI = 3400000e
RoIBTester          DEBUG RoI = 38000051
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 2
RoIBTester          DEBUG 3
RoIBTester          DEBUG 1
RoIBTester          DEBUG ---- new jetenergy slink ----
RoIBTester          DEBUG ee1234ee
RoIBTester          DEBUG 8
RoIBTester          DEBUG 2020000
RoIBTester          DEBUG 7301
RoIBTester          DEBUG 4
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 2
RoIBTester          DEBUG 0
RoIBTester          DEBUG 1
RoIBTester          DEBUG ---- new MuCTPI slink ----
RoIBTester          DEBUG ee1234ee
RoIBTester          DEBUG 8
RoIBTester          DEBUG 2020000
RoIBTester          DEBUG 7500
RoIBTester          DEBUG 4
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 0
RoIBTester          DEBUG 2
RoIBTester          DEBUG 0
RoIBTester          DEBUG 1
```

D Typical Output of the LVL1 Configuration Step

```
L1Config            INFO ========================================
L1Config            INFO Initialisation for L1Config algorithm.
L1Config            INFO ========================================
L1Config            INFO
TriggerMenu         INFO
TriggerMenu         INFO  ================================
TriggerMenu         INFO Initialising TriggerMenu stuff.
TriggerMenu         INFO  ================================
TriggerMenu         INFO -------------------------------------------------
TriggerMenu         INFO  XML file parsing in TriggerMenu.cxx successful !
TriggerMenu         INFO -------------------------------------------------
TriggerMenu         INFO
TriggerMenu         INFO ===============================================
TriggerMenu         INFO  The threshold map has the following entries :
TriggerMenu         INFO ===============================================
TriggerMenu         INFO
TriggerMenu         INFO ========================
TriggerMenu         INFO  Mapping trigger menu !
TriggerMenu         INFO ========================
TriggerMenu         INFO Creating trigger item iEM1 with prescale 1, priority l
ow and mask on.
TriggerMenu         INFO   Creating trigger condition EM01 which requires a mul
tiplicity of 2.
TriggerMenu         INFO Creating trigger item iEM2 with prescale 1, priority l
ow and mask on.
TriggerMenu         INFO   Creating trigger condition EM02 which requires a mul
tiplicity of 1.
TriggerMenu         INFO
TriggerMenu         INFO ----------------------------
TriggerMenu         INFO Starting trigger menu checks!
TriggerMenu         INFO ----------------------------
TriggerMenu         INFO Number of trigger items test:
TriggerMenu         INFO  PASSED. Number of trigger items: 2.
TriggerMenu         INFO Threshold number tests:
TriggerMenu         INFO  PASSED! Number of transverse energy thresholds: 0.
TriggerMenu         INFO  PASSED! Number of missing transverse energy threshold
s: 0.
TriggerMenu         INFO  PASSED! Number of jet sum thresholds: 0.
TriggerMenu         INFO  PASSED! Number of jet thresholds: 0.
TriggerMenu         INFO  PASSED! Number of muon thresholds: 0.
TriggerMenu         INFO  PASSED! Number of EM thresholds: 2.
TriggerMenu         INFO  PASSED! Number of HA thresholds: 0.
TriggerMenu         INFO  PASSED! Number of forward left jet thresholds: 0.
TriggerMenu         INFO  PASSED! Number of forward right jet thresholds: 0.
TriggerMenu         INFO Trigger threshold existence test:
TriggerMenu         INFO  PASSED! All necessary thresholds defined.
DetectorStore      DEBUG Recorded object 1
 of type LVL1CTP::MuctpiConfig(CLID 6027)
object modifiable when retrieved
DetectorStore      DEBUG retrieve(default): Retrieved const handle to default o
bject
 of type LVL1CTP::MuctpiConfig(CLID 6027)
```

```
L1Config             INFO The MuctpiConfig object has values 4 2 10
DetectorStore       DEBUG Recorded object /Run/L1TriggerMenuLocation
 of type LVL1CTP::TriggerMenu(CLID 6020)
object modifiable when retrieved
CTPHardware          INFO ========================================
CTPHardware          INFO Initialisation for CTPHardware algorithm.
CTPHardware          INFO ========================================
CTPHardware          INFO
DetectorStore       DEBUG Retrieved const handle to object /Run/L1TriggerMenuLoc
ation  of type LVL1CTP::TriggerMenu(CLID 6020)
Hardware             INFO  Distributing EM thresholds over PITs.
Hardware             INFO     - placing threshold EM01 on PITs starting at 0.
Hardware             INFO     - placing threshold EM02 on PITs starting at 3.
Hardware             INFO  Distributing MU thresholds over PITs.
Hardware             INFO  Distributing HA thresholds over PITs.
Hardware             INFO  Distributing JT thresholds over PITs.
Hardware             INFO  Distributing FL thresholds over PITs.
Hardware             INFO  Distributing FR thresholds over PITs.
Hardware             INFO  Distributing SM thresholds over PITs.
Hardware             INFO  Distributing ET thresholds over PITs.
Hardware             INFO  Distributing TM thresholds over PITs.
Hardware             INFO
Hardware             INFO ------------------------------------
Hardware             INFO  Checking consistency hardware - TM
Hardware             INFO ------------------------------------
Hardware             INFO Threshold / item on PIT test
Hardware             INFO
Hardware             INFO -------------------------------
Hardware             INFO    Building LUT config files.
Hardware             INFO -------------------------------
Hardware             INFO
Hardware             INFO ------------------------------------------
Hardware             INFO    Re-arranging MIOMap for use in CMBs.
Hardware             INFO ------------------------------------------
Hardware             INFO
Hardware             INFO -------------------------------
Hardware             INFO    Building CMB config files.
Hardware             INFO -------------------------------
CTPHardware          INFO  =================================
CTPHardware          INFO   Starting trigger configuration.
CTPHardware          INFO  =================================
CTPHardware          INFO
CTPHardware          INFO
CTPHardware          INFO  ================================
CTPHardware          INFO   Printing egamma configuration.
CTPHardware          INFO  ================================
CTPHardware          INFO
CTPHardware          INFO 3 PITs from 0: 3 PITs from 3:
CTPHardware          INFO  ==============================
CTPHardware          INFO   Printing muon configuration.
CTPHardware          INFO  ==============================
CTPHardware          INFO
CTPHardware          INFO
CTPHardware          INFO
CTPHardware          INFO  ======================================
CTPHardware          INFO   Printing jet / energy configuration.
CTPHardware          INFO  ======================================
CTPHardware          INFO
CTPHardware          INFO
DetectorStore       DEBUG Recorded object /Run/CaloTrigConfig
 of type LVL1CTP::CTPCaloConfig(CLID 6010)
object modifiable when retrieved
DetectorStore       DEBUG Recorded object /Run/MuonTrigConfig
 of type LVL1CTP::CTPMuonConfig(CLID 6011)
object modifiable when retrieved
DetectorStore       DEBUG Recorded object /Run/JetEnergyTrigConfig
 of type LVL1CTP::CTPJetEnergyConfig(CLID 6012)
object modifiable when retrieved
```

E Typical Output of the CTP and RoIB Simulations

```
CTPSim              DEBUG ==============================
CTPSim              DEBUG Execution of CTPSim algorithm.
CTPSim              DEBUG ==============================
CTPSim              DEBUG
DetectorStore       DEBUG Retrieved const handle to object /Run/L1TriggerMenuLoc
ation  of type LVL1CTP::TriggerMenu(CLID 6020)
CTPSim              DEBUG ==========================
CTPSim              DEBUG Filling of threshold bits.
CTPSim              DEBUG ==========================
CTPSim              DEBUG
DetectorStore       DEBUG Retrieved const handle to object /Run/MuonTrigConfig
of type LVL1CTP::CTPMuonConfig(CLID 6011)
StoreGateSvc        ERROR retrieve(default): No valid proxy for default object
 of type LVL1::MuCTPICTP(CLID 6070)
CTPSim              DEBUG WARNING retrieving MuCTPICTP object from StoreGate !
CTPSim              DEBUG Setting muon inputs to CTP to zero !
DetectorStore       DEBUG Retrieved const handle to object /Run/CaloTrigConfig
of type LVL1CTP::CTPCaloConfig(CLID 6010)
StoreGateSvc        DEBUG Retrieved const handle to object CaloTriggerDataLocati
on/EmTauCTP  of type LVL1::EmTauCTP(CLID 6253)
CTPSim              DEBUG EMTauCTP object has cable word 0 = 2 and cable word 1
= 33554432
DetectorStore       DEBUG retrieve(default): Retrieved const handle to default o
bject
 of type LVL1CTP::CTPJetEnergyConfig(CLID 6012)
StoreGateSvc        DEBUG retrieve(default): Retrieved const handle to default o
bject
 of type LVL1::JetCTP(CLID 6252)
CTPSim              DEBUG JetCTP object has cable word 0 = 536870912 and cable w
ord 1 = 536870912
StoreGateSvc        DEBUG retrieve(default): Retrieved const handle to default o
bject
 of type LVL1::EnergyCTP(CLID 6254)
CTPSim              DEBUG EnergyCTP object has cable word 0 = 0
CTPSim              DEBUG ==========================
CTPSim              DEBUG Evaluation of L1 decision.
CTPSim              DEBUG ===========================
CTPSim              DEBUG
CTPSim              DEBUG ===============================
CTPSim              DEBUG Creation/storage of CTP slink.
CTPSim              DEBUG ===============================
CTPSim              DEBUG
StoreGateSvc        DEBUG Recorded object /Event/CTPSLinkLocation
 of type LVL1CTP::CTPSLink(CLID 6013)
object modifiable when retrieved
RoIBuilder          DEBUG
RoIBuilder          DEBUG =============================
RoIBuilder          DEBUG Execution of RoIB algorithm.
RoIBuilder          DEBUG =============================
RoIBuilder          DEBUG
StoreGateSvc        DEBUG retrieve(default): Retrieved const handle to default o
bject
 of type EventInfo(CLID 2101)
StoreGateSvc        DEBUG Retrieved const handle to object /Event/CTPSLinkLocati
on  of type LVL1CTP::CTPSLink(CLID 6013)
RoIBuilder          DEBUG CTP RoI = 9 1 0
RoIBuilder          DEBUG CTP RoI = 10 0 0
RoIBuilder          DEBUG CTP RoI = 11 0 0
RoIBuilder          DEBUG CTP RoI = 12 0 0
RoIBuilder          DEBUG CTP RoI = 13 0 0
RoIBuilder          DEBUG CTP RoI = 14 0 0
RoIBuilder          DEBUG CTP RoI = 15 0 0
RoIBuilder          DEBUG CTP RoI = 16 0 0
RoIBuilder          DEBUG CTP RoI = 17 0 0
RoIBuilder          DEBUG CTP RoI = 18 0 0
RoIBuilder          DEBUG CTP RoI = 19 0 0
RoIBuilder          DEBUG CTP RoI = 20 0 0
RoIBuilder          DEBUG CTP RoI = 21 0 0
RoIBuilder          DEBUG CTP RoI = 22 0 0
RoIBuilder          DEBUG CTP RoI = 23 0 0
RoIBuilder          DEBUG CTP RoI = 24 0 0
RoIBuilder          DEBUG CTP RoI = 25 3 0
RoIBuilder          DEBUG CTP RoI = 26 0 0
RoIBuilder          DEBUG CTP RoI = 27 0 0
RoIBuilder          DEBUG L1 Accept = 0
StoreGateSvc        DEBUG Retrieved const handle to object CaloTriggerDataLocati
on/EmTauSlink0  of type DataVector<LVL1CTP::SlinkWord>(CLID 6250)
StoreGateSvc        DEBUG Retrieved const handle to object CaloTriggerDataLocati
on/EmTauSlink1  of type DataVector<LVL1CTP::SlinkWord>(CLID 6250)
StoreGateSvc        DEBUG Retrieved const handle to object CaloTriggerDataLocati
on/EmTauSlink2  of type DataVector<LVL1CTP::SlinkWord>(CLID 6250)
RoIBuilder          DEBUG Found EmTau RoI in slink 2 with value a650001and again
 a650001
StoreGateSvc        DEBUG Retrieved const handle to object CaloTriggerDataLocati
on/EmTauSlink3  of type DataVector<LVL1CTP::SlinkWord>(CLID 6250)
StoreGateSvc        DEBUG Retrieved const handle to object CaloTriggerDataLocati
on/JEPSlink0  of type DataVector<LVL1CTP::SlinkWord>(CLID 6250)
StoreGateSvc        DEBUG Retrieved const handle to object CaloTriggerDataLocati
on/JEPEnergySlink  of type DataVector<LVL1CTP::SlinkWord>(CLID 6250)
```

```
RoIBuilder          DEBUG energy RoI word = 805339148
RoIBuilder          DEBUG energy RoI word = 872448019
RoIBuilder          DEBUG energy RoI word = 939524164
StoreGateSvc        DEBUG Retrieved const handle to object CaloTriggerDataLocati
on/JEPSlink1  of type DataVector<LVL1CTP::SlinkWord>(CLID 6250)
StoreGateSvc        ERROR retrieve(default): No valid proxy for default object
 of type L1MUINT::MuCTPIToRoIBSLink(CLID 6103)
RoIBuilder          DEBUG Problem retrieving MuCTPI result from store!
RoIBuilder          DEBUG Creating empty MuCTPI RDO part!
StoreGateSvc        DEBUG Recorded object RoIBResult
 of type ROIB::RoIBResult(CLID 6000)
object modifiable when retrieved
```