Algorithms for the
ATLAS High Level Trigger

THE PESA CORE ALGORITHMS GROUP
(editor Steve Armstrong)
# Contents

1 Introduction 4

2 Conceptual Overview of the Software Framework 6
   2.1 Basic Concepts ................................................. 6
   2.2 The Seeding Mechanism of Algorithms .................. 8
   2.3 Guidance and Control of Algorithms: Steering .......... 9
   2.4 Data Access within Algorithms: The PESA Data Access Paradigm .... 10
      2.4.1 RegionSelection ........................................ 12
      2.4.2 The London Scheme ................................... 14
   2.5 HLT-specific Tools for Algorithms ....................... 16
      2.5.1 SpacePoint Formation ................................ 16

3 Concrete Examples using the Software Framework 19
   3.1 Steering .................................................. 19
   3.2 A Basic Example of the use of RegionSelector ........... 20

4 Common Components 22
   4.1 Raw Data Model and Data Model Core Software ........... 22
      4.1.1 Inner Detector ....................................... 23
      4.1.2 The Calorimeters ..................................... 27
      4.1.3 The Muon Spectrometer ............................... 31
   4.2 Identifiers ................................................ 32
   4.3 Reconstruction Input Objects (RIOs) ....................... 33
      4.3.1 Inner Detector ....................................... 33
      4.3.2 The Calorimeters ..................................... 33
      4.3.3 The Muon Spectrometer ............................... 35
   4.4 Reconstruction Output Physics Objects (ROPOs) .......... 35
      4.4.1 Track ................................................ 35
   4.5 Offline Core Software ...................................... 37
      4.5.1 StoreGate ........................................... 37
      4.5.2 IdentifiableContainer (IDC) .......................... 37
5 Coding Guidelines
5.1 General Coding Rules ........................................ 38
5.2 Level 2 Guidelines ........................................... 38
  5.2.1 General Design and Coding Issues .................... 39
  5.2.2 Athena/GAUDI Design and Coding Issues .......... 41
  5.2.3 Testing ................................................... 43
  5.2.4 Parameters ............................................... 43
5.3 Event Filter Guidelines ...................................... 44
  5.3.1 General Design and Coding Issues .................. 44
  5.3.2 Athena/GAUDI Design and Coding Issues .......... 45
  5.3.3 Testing ................................................... 46
  5.3.4 Parameters ............................................... 46

6 System Performance Measurement Methods .................. 47
  6.1 Tuning and Analysis Utilities (TAU) .................... 47
  6.2 Network Application Logger (NetLogger) ............... 48
  6.3 GAUDI Chrono and Auditor Services ................... 48
  6.4 TDAQ TimeStamp (TS) Library ......................... 49
  6.5 T2TimerSvc ............................................... 50

7 Inventory of Algorithms ....................................... 51
  7.1 Level 2 Algorithms ........................................ 51
    7.1.1 SiTree ............................................... 51
    7.1.2 IDSCAN ............................................... 51
    7.1.3 PixTrig/SiTrack .................................... 52
    7.1.4 TRT-LUT ............................................. 53
    7.1.5 TRT-Kalman ......................................... 54
    7.1.6 T2Calo .............................................. 55
    7.1.7 muFast .............................................. 55
    7.1.8 muComb ............................................... 56
    7.1.9 WuppertalVertexing .................................. 57
  7.2 Event Filter Algorithms .................................. 58
    7.2.1 xKalman++ ........................................... 58
    7.2.2 iPatRec .............................................. 59
7.2.3 LArClusterRec .............................................. 61
7.2.4 egammaRec .................................................. 61
7.2.5 Moore ......................................................... 62

List of Figures

1 The ATLAS Software universe from HLT perspective. ...................... 5
2 Package diagram of PESA Software ...................................... 7
3 Examples for relations between EDM classes during processing .......... 8
4 UML class diagram form HLTAlgo. ........................................ 11
5 The London Scheme ................................................................ 14
6 A UML class diagram of the LVL-2 specific SpacePoint class. .......... 17
7 The ATLAS Raw Data Flow ..................................................... 22
8 A UML class diagram of Calorimeter RIOs. ................................. 34
9 A UML class diagram of the LVL2 Track Class. ......................... 36

List of Tables

1 Example of a RegionSelector DetectorDescription data file. ............. 12
2 Execution time of RegionSelector for various $\Delta \eta \times \Delta \phi$ .............. 13
3 An example of a Sequence table in XML format. .......................... 20
4 An example of a Signature table in XML format. ............................ 21
5 The content of the Pixel, SCT, and TRT ROD Fragments. ............... 24
6 The content of the LAr and Tile ROD Fragments. ........................ 28
7 General Coding Rule Categories ............................................ 38
8 TimeStamp library compiler flags ............................................ 49
1 Introduction

The ATLAS High Level Trigger (HLT) consists of a software-based framework in which guided algorithms validate or invalidate specific trigger hypotheses. The HLT consists of two distinct stages, the Level 2 (LVL2) Trigger and the Event Filter (EF).

Algorithms running in the LVL2 environment must confront a variety of challenges. Algorithms must request a restricted portion of data from the detector via the Region-of-Interest (RoI) mechanism and execute relevant portions of their code within the 10 ms average latency required to generate a LVL2 decision.\(^1\) Furthermore, to utilize fully the HLT processing limited resources, code will be run at LVL2 in a multithreaded environment. Event Filter algorithms are anticipated to operate in a manner closer to that of the Offline environment. They potentially have full access to all detector data. Furthermore, they have a more generous average latency of 1 s in which to make a decision.

This document is intended to provide a conceptual review and practical guide for HLT algorithm development. Section 2 provides a brief overview of the HLT software framework, referred to as the PESA Core Software. This is followed in Section 3 with concrete examples that algorithm developers may follow in order to integrate their code within this framework in order that they become part of the suite of PESA Core Algorithms. Furthermore, relevant software components which are shared in common between the PESA Core Software Framework, PESA Core Algorithms, and Offline Reconstruction Programs (ORPs) are reviewed in Section 4 since no adequate or unified documentation exists elsewhere. Figure 1 provides simple view of the relationships between PESA Core Algorithms, PESA Core Software, Common software Components, and other items relevant to the HLT.

Coding guidelines which are vital to successful integration of algorithms into the LVL2 and EF environments are discussed in Section 5. Methods that may be used to profile the system performance of algorithms are reviewed in Section 6. Finally, Section 7 provides a comprehensive inventory of all algorithms for LVL2 and EF that are known to be under active development at the time of writing this document. An Appendix is also provided to define terms and provide a reference for the often confusing variety of acronyms used in the HLT world.

\(^1\)The CPU time component of the latencies quoted in this document for both LVL2 and EF algorithms are for 4 GHz machines.
Figure 1: The ATLAS Software universe from the perspective of an HLT algorithm developer.
2 Conceptual Overview of the Software Framework

The requirements and a conceptual design of the PESA Core Software are discussed in detail elsewhere [1, 2].

In the following subsections, the anticipated interactions Algorithms will have with other components of the PESA Core Software is discussed from an conceptual standpoint. Concrete examples are given in Section 3.

Figure 2 shows a package diagram illustrating the current structure of the HLT Software found under the Trigger area of the ATLAS CVS repository. HLT Algorithms exist within the TrigAlgorithms package container; an inventory of these algorithms is given in Section 7. Guidance and Control of Algorithms is accomplished by the PESA Core Software Steering found in the TrigSteer package container and discussed in Section 2.3. PESA Core Software components relevant to the data access paradigm are found in the TrigDataAccess package container and are discussed in Section 2.4. Tools specific to HLT Algorithms are found in the TrigTools package container and are discussed in Section 2.5. Details of the Level 1 Simulation within the TrigT1 package container are discussed in detail elsewhere [3].

2.1 Basic Concepts

A knowledge of the following concepts is essential for an understanding of the HLT PESA Software from a top-level perspective and in the following subsections.

- **Seed and Seeding.** At the heart of the philosophy of the PESA design is the concept of *seeding*. Algorithms functioning in the context of the High Level Trigger do not operate in a general-purpose or exclusive sense; rather, they must validate or reject Trigger Element hypotheses formed at a previous stage in the triggering process.

- **Trigger Element.** A Trigger Element (TE) is a tag for either a Reconstruction Output Physics Object (ROPO) or another TE linked together in a navigable way within a Store. A TE can be stored in StoreGate like any other object, but the algorithm developer has no control over this. The sole responsible for TE creation is the Steering (see Section 2.3). An algorithm can only (de)activate TEs in the Store. Trigger Elements are discussed in more detail elsewhere [1, 2, 4].

---

2 Since the time these documents were written, many of the components have been revised during their implementation within PESA Software Releases.

3 Note that only the *package containier* s directly under the Trigger area of the CVS repository shown in Figure 2 comprise the current HLT Software. Older packages which are not package containers are obsolete and deprecated. For example, one should use Trigger/TrigDataAccess/TrigStore and not Trigger/TrigStore.

4 This is an object with a label; currently this is a string, but in future releases will support integers in future.
Figure 2: A package diagram illustrating the structure of the PESA Software within the Trigger area of the ATLAS CVS repository.

- **Sequence** A transformation of one or more (input) TEs via a set of HLT Algorithms into a new (output) TE. The Steering obtains information about the specific HLT Algorithm to execute for a given Seed from the Sequence, and each Algorithm in a Sequence is configured with a relevant parameter set [2].

- **Signature.** An element in the Menu Table containing collections of required TEs, a prescaling factor, and a forced accept rate [2].

- **Menu Table.** A collection of Signatures that could be validated in a given Step [2].

The HLT algorithms are the building blocks of the reconstruction that provides the data to derive the trigger decision. Logically the Trigger processing is done starting from a LVL1 RoI using predefined Sequences of algorithms. In practice each of these LVL1 RoI objects is decorated by a Trigger Element, because these are the special objects upon which the Steering is acting. For each of these Trigger Elements the Steering executes the required algorithms as defined in the Sequence table. This implies that a given algorithm may be executed N times per event. This is conceptually different the Offline reconstruction where any algorithm acts upon the full event.
2.2 The Seeding Mechanism of Algorithms

A seeding mechanism is needed in order to guide the reconstruction to the data fragment of the event upon which it should work. The Trigger Elements characterize the abstract physics objects with their label (e.g., “electrons” or “jets”). This is beneficial because one does not couple directly the Steering of the trigger selection to the details of the Event Data Model used by the algorithms. No micro sequencing is done on the level of (e.g., single Calorimeter Cells or Inner Detector Clusters). These complex event details are better handled by the algorithms themselves. Other than the label, a Trigger Element does not have any properties or states of its own. Such a list of possible states or properties is not well-defined a priori. Instead it is the uses relation by which the Trigger Element is associated to the concrete event data, that should provide all necessary information to the algorithms. A Trigger Element is thereby an entry point for the algorithm into the event.

Figure 3 shows an illustrative example of the Seeding Mechanism. Figure 3(a) illus-
trates an example of an electromagnetic RoI object from the LVL1 calorimeter trigger. The LVL1Conversion of the Steering provides an “EM” Trigger Element that has a “uses” relation to the RoI object. At the first reconstruction step the Steering executes the algorithm(s) for a Sequence that matches this input Trigger Element. The Steering creates the expected output Trigger Element as specified in the Sequence. The output Trigger Element has a “seeded by” relation to the input Trigger Element as shown in Figure 3(b). The task of the algorithm(s) is to validate this hypothesis. The Steering invokes the HLTAlgo method `execHLTAlgorithm(TriggerElement* outputTE)` that any HLT algorithm has to implement. In this sense the output Trigger Element is the “seed”.

The algorithm is seeded by the output Trigger Element, but it needs to access the detailed event information, in this case the RoI object. Therefore it navigates via the “seeded by” and “uses” relation to obtain the RoI object and then does its reconstruction and analysis work. The result of the reconstruction in this example is a calorimeter cluster that uses several calorimeter cells. As shown in Figure 3(b), the cluster gets linked to the output Trigger Element by a “uses” relation. Thereby the subsequent algorithm can access the cluster found for this RoI. The second task of the algorithm is the analysis or decision if it was successful in validating the hypothesis of the output Trigger Element. If so, it activates the Trigger Element. Without activation the Steering drops the Trigger Element and it would not be considered to validate any physics signature, presumably leading to a negative trigger decision.

In our “electron” example, the physics hypothesis is tested in the next step using tracking information. As before, the Steering generates the output Trigger Elements and executes the algorithm with the new “seed”. For the tracking algorithm it is beneficial not to use the position of the initial RoI, but to used the more precise information of the cluster obtained in the previous step (see Figure 3(c)). Hence one navigates via the “seeded by” and “uses” links to the cluster. Then the track is reconstructed and linked into the output Trigger Element. Finally, if successful, the output Trigger Element is again activated.

The example illustrates the principle concept of the “seeding” mechanism. The “seed” provides an entry point into the already available event fragment that corresponds to an initial RoI. The navigation allows to extract specific information from existing objects (RoI, Cluster, Track, etc.) as input for the next reconstruction step. It is up to the algorithm implementation to decide, which input information it uses from previous steps.

### 2.3 Guidance and Control of Algorithms: Steering

The primary concept motivating the Steering component of the HLT Software is the need for a fast and early rejection of uninteresting events in a flexible and configurable manner.\(^5\) This is realized in a way that permits full control of the Algorithms executing within the

---

\(^5\)The Steering is also motivated by a need for a pre-scale/forced-accept of some events.
HLT processing flow with the simple modification of XML configuration files.

The Step Controller (SC) of the Steering software replaces the Athena Event Loop Manager and has the responsibility of calling HLT Algorithms. Two XML files encode Sequences and Signatures that in turn instruct the Steering on when and how to run an algorithm and if a physics signature is fulfilled. The HLT processing flow is disaggregated into Steps. The decision to go further in the process is taken at every new Step by the comparison between active Trigger Elements in the Store and the corresponding Configuration Signature.

An event is accepted if all its constituent Sequences have been executed and at least one of the corresponding Configuration Signatures has been satisfied. Examples of Signatures and Sequence Tables are provided for some example test cases and can be found in every PESA software release.

In order to use the Steering, the only strong requirement imposed upon developers is that their algorithms inherit from the HLTAlgo base class that augments the Athena Algorithm base class with some HLT Navigation helper functions. This HLTAlgo class is found in the package Trigger/TrigSteer/TrigSteering. Figure 4 shows a UML class diagram for HLTAlgo and associated classes.

### 2.4 Data Access within Algorithms: The PESA Data Access Paradigm

Of course, Algorithms must have event data upon which to work to reconstruct physics objects of interest. Within the HLT environment, the data access paradigm is different than that of the Offline.

At LVL2, event data reside within ROBs until actively requested. Based on information from the LVL1 trigger, it is intended that the LVL2 algorithms consider only relevant Regions-of-Interest when validating the LVL1 decision. This allows the LVL2 algorithms to request and process only a small fraction of event data from ROBs, representing a substantial reduction in the network and computation resources required. The first step in this process is the conversion of a geometrical region (e.g., a cone with an extent $\eta$ and $\phi$) into Identifiers; this is accomplished with a RegionSelector described in Section 2.4.1. Interactions with the Data Collection system are hidden from the Algorithm behind a call to StoreGate as described in Section 2.4.2.

In the EF, the data access paradigm is closer to that of the Offline environment since there is the possibility of full access to event data.

10
Figure 4: A UML class diagram showing HLTAlgo and associated classes.
Table 1: An illustrative example of an RegionSelector ASCII file containing DetectorDescription data. For subdetectors with multiple layers, there would also be radius information of each element.

![Table 1](image)

2.4.1 RegionSelection

The HLT RegionSelector [5] translates geometrical regions\(^6\) within the fiducial volume of the detector into a set of Identifiers. Presently these Identifiers are IdentifierHashes corresponding to elements of appropriate granularity in each sub-detector, usually a DetectorElement. These Identifiers are used to request and organize the event data.

The RegionSelector uses ATLAS DetectorDescription information during its initialization phase. In the current implementation, an ASCII data file exists for each sub-detector\(^7\) that provides information needed to build an internal map of Identifier lists, referred to as the EtaPhiMap. An illustrative example of an ASCII data file containing DetectorDescription information is shown in Table 1.

An EtaPhiMap exists for each layer (or disk) of a subdetector and is essentially a two-dimensional matrix in \(\eta\) and \(\phi\). Each element consists of a list of IdentifierHash; the column indices are \(\phi\) floating point numbers while a range \((\eta_{\text{min}}, \eta_{\text{max}})\) specifies row indices.

This requires two different sub-maps: one that associates \(\phi\) with the IdentifierHash (\(\phi \rightarrow\)IdentifierHash map) and the other that associates the IdentifierHash with the \(\eta\) range (IdentifierHash\(\rightarrow\) \((\eta_{\text{min}}, \eta_{\text{max}})\) map). This dual map approach improves system performance and insures that the EtaPhiMap does not duplicate IdentifierHashes inside the IdentifierHash\(\rightarrow\) \((\eta_{\text{min}}, \eta_{\text{max}})\) map.

The input to RegionSelector API is the sub-detector under consideration (i.e., Pixel, SCT, TRT, LAr, Tile, MDT, RPC, CSC, or TGC) and the extent of the geometrical region. Given the vastly different designs of each subdetector, a subdetector-dependent procedure is used. The algorithm immediately considers if the positive, negative, or both

---

6In the current implementation, this abstract geometrical region is a cone in \(\eta\) and \(\phi\). The significant spread in the \(z\) position of the primary vertex is imbedded into the scheme for the innermost subdetector (i.e., Pixel and SCT). In future implementations, additional types of regions will be supported (e.g., helical roads).

7In future implementations this will be replaced by dynamically generated tables drawing directly from the ATLAS DetectorDescription services.
regions are being considered and uses only the relevant maps.

With knowledge of the layers and/or disks in the region, the algorithm searches the \( \phi \rightarrow \text{IdentifierHash} \) map which will give a set of \text{IdentifierHash} is relevant in \( \phi \) region. The last step of the algorithm is to validate each \text{IdentifierHash} inside the \text{IdentifierHash} \rightarrow (\eta_{\text{min}}, \eta_{\text{max}}) \) map.

An emphasis on optimal system performance (\textit{i.e.}, fast execution time) is vital due to the short LVL2 latency. Obviously, if the RegionSelector spends more time providing Identifiers for a given Region than the time required to read-out and process the entire detector, it would defeat the purpose for its existence.

The mean execution time to retrieve a region on each subdetector in a \( \Delta\eta \times \Delta\phi \) regions of various sizes is specified in Table 2 on a 1 GHz Pentium 3 LXPLUS Linux machine. There is an active effort to minimize these times, especially for the sub-detectors with large numbers of Identifiers (\textit{i.e.}, SCT and TRT). Note that system performance is largely independent of the extent in \( \Delta\eta \) since the bulk of the time spent in lookup for the \text{EtaPhiMap} is first spent determining the location for the given \( \phi_{\text{min}} \) and \( \phi_{\text{max}} \) as discussed above.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
\text{Region} & \text{0.1} & \text{0.2} & \text{0.5} \\
\hline
Pixel & 0.2 & 0.2 & 0.2 \\
SCT & 0.6 & 0.6 & 0.6 \\
TRT & 0.7 & 0.7 & 0.7 \\
LAr & 0.04 & 0.04 & 0.04 \\
Tile & 0.04 & 0.04 & 0.04 \\
\hline
\end{tabular}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
\text{Region} & \text{0.1} & \text{0.2} & \text{0.5} \\
\hline
Pixel & 0.3 & 0.3 & 0.3 \\
SCT & 0.9 & 0.9 & 1.0 \\
TRT & 1.1 & 1.1 & 1.1 \\
LAr & 0.05 & 0.05 & 0.05 \\
Tile & 0.05 & 0.05 & 0.05 \\
\hline
\end{tabular}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
\text{Region} & \text{0.1} & \text{0.2} & \text{0.5} \\
\hline
Pixel & 0.6 & 0.6 & 0.6 \\
SCT & 1.9 & 1.9 & 2.0 \\
TRT & 2.4 & 2.4 & 2.4 \\
LAr & 0.07 & 0.07 & 0.07 \\
Tile & 0.09 & 0.09 & 0.09 \\
\hline
\end{tabular}
\end{table}

Table 2: Preliminary execution time of RegionSelector in milliseconds on a 1 GHz Pentium 3 Linux machine. Work is in progress to improve the performance for SCT and TRT. Note that performance is roughly independent of the extent in \( \Delta\eta \).
2.4.2 The London Scheme

The London Scheme\(^8\) is the foreseen data access paradigm that HLT algorithms are required to use to access event data. The basic idea of the scheme is to hide the data access details (i.e., interaction with the Data Collection system) from the HLT algorithms behind a call to StoreGate [6]. Refer to Figure 5 for a diagramatic representation of the London Scheme.

Central to the scheme is that the data in StoreGate are organized in a specific way. The event data of a single detector are contained in one IdentifiableContainer (IDC) [7, 8]. Collections within the IDC contain a certain species of event data (e.g., RIOs, SpacePoints, or RDOs). The collections are tagged with an identifier. Presently this identifier is a IdentifierHash corresponding to an element of appropriate granularity within each sub-detector.

When an HLT algorithm is called by the StepSequencer in the Steering, the following steps occur:

LS.1 The algorithm is called with one or more TriggerElements in its parameter list. Assume in this example that only one TriggerElement is passed, and that this TriggerElement has a navigable link to a RoI. The RoI contains, among other informa-

---

\(^8\)The scheme was invented in a meeting at Royal Holloway near London in July 2002, hence the name.
tion, its position \((\eta, \phi)\).

LS.2 The HLT algorithm typically needs to access the event data that pertains to a region in \(\Delta \eta, \Delta \phi\) around the center of the RoI position for at least one sub-detector. The algorithm calls the RegionSelector with parameters of this region as arguments and also specifies which sub-detector is of interest. The RegionSelector returns a list of the Offline Identifiers of RIOs (\(n.b., \) the following description applies equally for RDOs) that are contained within the region.

LS.3 The HLT algorithm retrieves from StoreGate the relevant IDC or IDCs and, by means of the list of Offline Identifiers, requests from each IDC the corresponding collections of RIOs. The IDCs then requests those RIOs from StoreGate. Two cases can occur:

(a) The RIOs are already available in StoreGate. In this case, StoreGate returns the RIOs to the IDC which returns them to the HLT algorithm; or

(b) The RIO is not yet available in StoreGate. In this case, to make the requested RIO objects available in StoreGate, the data must be retrieved from the Data Collection system. To this end, the \texttt{IOpaqueAddress} internally used by StoreGate has been extended to include an Offline Identifier and the identifier of the ROB that holds the data of these RIOs. Crucial for this is a map between Offline Identifier(s) and corresponding ROB identifier(s) that is loaded in the initialization step. Note that a ROB can hold the data of a number of DetectorElements.\(^9\) However, the smallest amount of data that can be provided by the data flow system in one request is the full data content of a single ROB.

LS.4 StoreGate calls the ByteStreamConverter with the \texttt{IOpaqueAddress} as an argument.

LS.5 The ByteStreamConverter retrieves from the ROB DataCollector (or its emulation for Monte Carlo) the ByteStream data from the ROB that corresponds to the DetectorElement in question.

LS.6 The ByteStreamConverter extracts from the ByteStream the 32-bit words that correspond to event data and converts those 32-bit words into either RDOs or, by invoking a DataPreparation \texttt{AlgTool}, into RIOs. The knowledge about the precise header and footer configuration in the ByteStream that is necessary to identify the event data related 32-bit words is contained in the EventFormatLibrary \([9]\).

LS.7 The thus obtained RDO or RIO objects are stored in StoreGate.

\(^9\)The exceptions to this are the TRT and LAr. In the case of the TRT endcap DetectorElements, a single DetectorElement spans three ROBs due to the 1/32 of 2\(\pi\) \(\phi\) segmentation of the former and the 1/96 of 2\(\pi\) \(\phi\) segmentation of the latter. In the case of LAr, the concept of DetectorElement does not exist, but it should; instead the concept of TriggerTower is used.
The London Scheme foresees data buffering which is beneficial in the following sense. If the HLT algorithm requests only the data corresponding to one DetectorElement, the data corresponding to one full ROB must be retrieved from the Data Collection system. The ByteStreamConverter allows to pass the full ROB data to StoreGate for transient storage. In that case, when the HLT algorithm requests a second DetectorElement from the same ROB, it is not necessary to invoke the Data Collection system and request the full ROB ByteStream a second time.

2.5 HLT-specific Tools for Algorithms

2.5.1 SpacePoint Formation

LVL2-specific clusters (i.e., SCT_Cluster and PixelCluster in the InnerDetector/InDetRecEvent/InDetRecInput package) give the position of the Clusters within the local 1 (2)-dimensional coordinate system of the SCT (Pixel) DetectorElement. These are converted to 3-dimensional coordinates in the ATLAS global coordinate system using the AlgTools SCT_SpacePointTool and PixelSpacePointTool. These tools accept as input a STL vector of pointers to Cluster Collections of the appropriate type, SCT_ClusterCollection or PixelClusterCollection, and return a STL vector of objects of the class TrigSiSpacePoint. A UML class diagram of the LVL2-specific SpacePoint class TrigSiSpacePoint and associated InDetRecInput classes is shown in Figure 6.

The geometry information required for this transformation is supplied by the class TrigSiGeoSvc, which provides an interface to the Detector Description.

For the Pixels, the creation of SpacePoints consists of combining the local coordinates of Clusters with information on the position and orientation of the DetectorElement to give the global coordinates.

The process for the SCT is more complicated since a single SCT detector provides only a one-dimensional measurement. However, an SCT module, consisting of two detectors in a stereo-pair, provide 2-dimensional information. One species of SCT DetectorElement, phi-layer, has strips orientated parallel to the beam axis, the other, u or v layer, is rotated by ±40 mRad with respect to the phi-layer DetectorElements. The formation of SpacePoints consists of the following steps:

- Associate each phi-layer Cluster Collection\(^{10}\) with the corresponding stereo-layer Cluster Collection;
- For each pair of Collections (phi + stereo), take each phi-layer Cluster and search for associated stereo-layer Clusters. If there is more than one associated stereo layer Cluster, a SpacePoint is formed for each (in this case one, at most, will be a correct

\(^{10}\)There is a Cluster Collection per DetectorElement.
measurement, the others will form "ghost" points). If no associated stereo-layer hit is found, a point is created from the phi-layer information alone:

- Calculate the second coordinate (z for the barrel, or R for the end-caps);
- Using information on the position and orientation of the DetectorElement transform to global coordinates.

Note that for the LVL2 SpacePoints some simplifications are made in the interest of speed, as follows:

- No attempt is made to form SpacePoints from Tracks passing close to the edge of a module, where the corresponding stereo-layer Cluster is in a different module.
- Since the stereo and phi layers are separated by a small distance, the trajectory of the track will influence the measurement of the second coordinate. Since the trajectory
is not known at the time that SpacePoints are created, there will be a corresponding increase in the uncertainty in the measurement in the second coordinate ($R$ or $z$).
3 Concrete Examples using the Software Framework

The purpose of this Section is to provide extremely explicit and specific examples of the use of the PESA Core Software framework to Algorithm developers.

More text here to explicitly follow the TrigAlgExample.

3.1 Steering

As Section 2.3, a developer needs to set-up or at least update a Sequence and a Signature list to instruct the Steering on what to do with an algorithm. A simple example Sequence is

\[
\text{TE\textunderscore input} \quad \text{AlgName} \quad \text{TE\textunderscore output}
\]

The Steering would read this Sequence, and, based on it, look for active TEs with the label \text{TE\textunderscore input} in the Store. If any is found, then the Steering creates a TE with label \text{TE\textunderscore output} and then executes the Algorithm with name \text{AlgName}. The pointer to \text{TE\textunderscore output} is passed as an argument to the Algorithm that in turn can setup navigational links to existing Reconstructed Objects with it.

In this scheme, the \text{TE\textunderscore output} is eventually seen as a seed for the next Algorithm through another sequence. An example of a complete Sequence is

\[
\begin{align*}
\text{LVL2:} & \quad \text{TE1 Alg1 TE2} \\
\text{LVL2:} & \quad \text{TE2 Alg2 TE3} \\
\text{LVL2:} & \quad \text{TE3 Alg3 TE4} \\
\text{EF:} & \quad \text{TE4 Alg4 TE5}
\end{align*}
\]

where the LVL2 and EF tags have been added since it is possible to select at which HLT level the Algorithm should be run. The Algorithm name itself has the form

\[
\text{AlgName/AlgInstance/ParameterFileName}
\]

where \text{AlgName} is the algorithm name as known to the developer(s) (e.g., \text{T2Calo}); \text{AlgInstance} is the name of the algorithm created by the Steering\textsuperscript{11}; \text{ParameterFileName} is potentially the name of a file that is available to the algorithm and in principle accessed and parsed by it to read in some extra configuration information.\textsuperscript{12}

An example of a Sequence table in XML format is provided in Table 3. An example of a Signature table in XML format is provided in Table 4.

\textsuperscript{11}In principle there can be several instances of the same algorithm running for different sequences.

\textsuperscript{12}The content of this file is completely left to the discretion of the developer or empty/non-existent if not needed.
### 3.2 A Basic Example of the use of RegionSelector

When using the RegionSelector, an algorithm must give as input to the API the sub-detector and the region in terms of η and φ ranges. An example of this is

```cpp
// Example RegionSelector inputs
float etamin=0.1, etamax=0.2, phimin=0.1, phimax=0.2;

// Declaration of the Algorithm Factory
static const AlgFactory<RegionSelectorEx> Factory;
const IAlgFactory& RegionSelectorExFactory = Factory;

RegionSelectorEx::RegionSelectorEx(const std::string& name, ISvcLocator* pSvcLocator)
    : Algorithm(name, pSvcLocator), m_pRegionSelector(0) {
    // declare properties
    declareProperty( "RegionSelectorTool", m_RegionSelectorTool = "RegionSelector" );
}

StatusCode RegionSelectorEx::execute() {
    std::vector<int> List_ID = m_pRegionSelector->DetIDList(PIXEL, etamin, etamax, phimin, phimax);
    return StatusCode::SUCCESS;
}
```

Table 3: An example of a Sequence table in XML format.
<table>
<thead>
<tr>
<th>Signature ID</th>
<th>Prescale</th>
<th>Acceptance</th>
</tr>
</thead>
<tbody>
<tr>
<td>g40i</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>gmu</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>newelec</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>gammaelecmu</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>g40x2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>g40x3</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4: An example of a Signature table in XML format.

The output is a standard STL vector of integers with unique IdentifierHashes corresponding to the region.
4 Common Components

4.1 Raw Data Model and Data Model Core Software

This subsection summarizes aspects of ByteStream Raw Data, Raw Data Objects (RDOs), and Converters between the two for each sub-detector.

Figure 7 illustrates the role of LVL2 and EF Algorithms within a simplified view of the ATLAS Raw Data Flow from Ref [10] which emphasizes the simulation chain. This view of the ATLAS Raw Data Flow is still considered to be under development and discussion.

ByteStream Raw Data is Read-Out Buffer (ROB)-formatted data produced by the ATLAS detector or its simulation. It is defined by a set of hierarchical fragments, where only the bottom level, the ROD fragment, is defined by the sub-detector group. The format of the ByteStream has not yet been formally defined. Hence, preliminary “best guesses” have been made as to its structure which may undergo changes in the future.

A Raw Data Object (RDO) is uncalibrated Raw Data converted into an object representing a set of readout channels. Historically this has been referred to as a Digit. It is the representation of Raw Data which is put into the Transient Event Store (TES) and
is potentially persistifiable.

The purpose of the RDO converters is dual: first a Raw Data ByteStream file can be created by taking the information from the already filled RDOs (in the transient store, from ZEBRA); second, this ByteStream file can then be read back by the converters to fill the RDOs (or the RIOs for LVL2). Since the RDOs are a representation of the specific detector output, its content can change with the life time of the sub-detectors.

In general, an algorithm developer does not need to deal with these Raw Data Model issues; they would work instead with Reconstruction Input Objects (RIOs) or even Reconstruction Output Physics Objects (ROPOs) derived from the Raw Data. However, for the sake of completeness as well as for developers of tools, sub-algorithms (e.g., clustering schemes), or ultra-specialized algorithms which may need direct access to RDOs, they are described here.

4.1.1 Inner Detector

The implementation of the RDOs for the Inner Detector is based on a design reflective of the possibility that the content of RDOs may evolve in time. Hence there is the possibility of having several concrete classes of RDOs. A user who wishes to create a RDO presumably knows which data taking mode the subdetector is in, so the right concrete class will get instantiated. Downstream users who wish to access already filled RDOs can access the base class instead of the specific concrete classes.

Another important point about the implementation of these objects is that access to the content of the word that constitutes the raw information is done through member function that apply the right bit field masks to the word. Algorithm code should not have to apply masks, as this information can change with time, as already stated.

Finally, the implementation of the RDOs makes use of the IdentifiableContainer (IDC) base class for a quick access to the stored collections, which, for the Inner Detector, has a granularity corresponding to DetectorElements in each sub-detector.

The Pixel Detector

Components of the Pixel Detector Raw Data Model have been implemented in package InnerDetector/InDetEventCnv/InDetRawDataByteStream. This package contains the converters needed to write ByteStream from RDO, and, conversely, to read Bytestream from RDO.

The conversion of Pixel RDOs into ByteStream is provided by the AlgTool PixelRawContByteStreamTool while (PixelRawContByteStreamCnv is the Converter class). The tool iterated on collections of RDOs within the PixelRDO_Container (an IdentifiableContainer class). Each collection corresponds to a Pixel Detector Element. The ByteStream to RDO conversion is performed by the AlgTool class PixelRawCollByteStreamTool which converts the words of the ROD fragment (see Ta-
### Pixel Detector

<table>
<thead>
<tr>
<th>Source ID</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Subdetector ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 = Barrel</td>
<td></td>
</tr>
<tr>
<td>101 = Forward A</td>
<td></td>
</tr>
<tr>
<td>102 = Forward C</td>
<td></td>
</tr>
<tr>
<td>103 = B-Layer</td>
<td></td>
</tr>
</tbody>
</table>

| ROB ID | (0, 119) |

<table>
<thead>
<tr>
<th>Data Element</th>
<th>Array of RawDataWord = :Module:*:FrontEnd:Row:Column:tot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module</td>
<td>5 bits module wafer number in ROD</td>
</tr>
<tr>
<td>*</td>
<td>2 bits information on EndOfEvent</td>
</tr>
<tr>
<td>FrontEnd</td>
<td>4 bits front end chip number</td>
</tr>
<tr>
<td>Row</td>
<td>8 bits row number (0, 160)</td>
</tr>
<tr>
<td>Column</td>
<td>5 bits column number (0, 23)</td>
</tr>
<tr>
<td>Tot</td>
<td>8 bits time of trigger</td>
</tr>
</tbody>
</table>

| Trailer |

### SCT

<table>
<thead>
<tr>
<th>REID</th>
<th>unsigned int</th>
<th>raw identifier for each module</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAW DATA WORD</td>
<td>32-bit word</td>
<td>the strip number and raw data</td>
</tr>
</tbody>
</table>

### TRT data block format

<table>
<thead>
<tr>
<th>Header</th>
<th>Block start code + group number + block number</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Up to 26 data words</th>
<th>Data word format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel address</td>
<td>bits 27 – 31</td>
</tr>
<tr>
<td>TS1 HT</td>
<td>bit 26</td>
</tr>
<tr>
<td>TS1 LT</td>
<td>bits 18 – 25</td>
</tr>
<tr>
<td>TS2 HT</td>
<td>bit 17</td>
</tr>
<tr>
<td>TS2 LT</td>
<td>bits 9 – 16</td>
</tr>
<tr>
<td>TS3 HT</td>
<td>bit 8</td>
</tr>
<tr>
<td>TS3 LT</td>
<td>bits 0 – 7</td>
</tr>
</tbody>
</table>

| Trailer | Block end code + number of hit channels |

Table 5: The content of the Pixel, SCT, and TRT ROD Fragments.
ble 5) into RDOs and put them together into collections. The content of each Pixel ROD fragment is given in Table 5.

For each of these conversions, an Offline to Online Identifier map is required. This is accomplished by the PixelHid2RESrcID class which also provides ByteStream address objects for the collections. Note that the current Pixel mapping assumes all RODs belonging to the same subdetector are aggregated as into a Read-Out-Crate.\(^\text{13}\) Two files for the Barrel Pixel detector (and two for the End-caps) provide the mapping between each Offline Identifier corresponding to an individual DetectorElement and an Online Identifier corresponding to its position and number in the read-out tree. The mapping files are contained in the InnerDetector/InDetEventCnv/SiTrackerByteStreamCnv area. In PixelBarrelMUR2RESrcIdMapping.dat, the mapping between

\[
\text{semi-barrel considered} \rightarrow \text{semi-stave number} \rightarrow \text{ROD} \rightarrow \text{ROS}
\]

is provided. In PixelBarrelMUR2TEMapping.dat, the mappings are set between

\[
\begin{align*}
\text{semi-stave number} & \rightarrow \text{detector layer} \\
\text{detector layer} & \rightarrow \eta_{\text{min}} \text{ of module} \\
\eta \text{ of module} & \rightarrow \text{semi-barrel considered} \\
\text{detector layer} & \rightarrow \eta \text{ of module} \rightarrow \text{number of modules in } \phi
\end{align*}
\]

Similar files exist for the endcaps.

**SCT**

The package InnerDetector/InDetEventCnv/InDetRawDataByteStream provides conversion from SCT RDOs (SCT_RDORawData) in ByteStream format and vice versa. The composition of each SCT ROD fragment is given in Table 5.

Each pair of one REID and one RAW DATA WORD corresponds to one RDO.

The mapping from SCT modules to RODs is provided in SCTDigitAccessor and based on Ref. [11]. The mapping service uses raw identifiers (REID) to describe the connection of the SCT module to a Minimal Unit of Readout (MUR) and Transient Identifiers (TEID) to describe the module in terms of the physical location in the SCT. The bitpattern of the REIDs and TEIDs is based on Ref. [12]. Mapping files are contained in the InnerDetector/InDetEventCnv/SiTrackerByteStreamCnv area: SCTBarrelMUR2RESrcIdMapping.dat, SCTEndcapMUR2RESrcIdMapping.dat, SCTBarrelMUR2TEMapping.dat, and SCTEndcapMUR2TEMapping.dat. These .dat files are read in by SCTDigitAccessor during the initialization phase.

**TRT**

To understand fully the TRT Raw Data Model, a brief discussion of the nature of the

\(^\text{13}\)This ROC definition is technically obsolete.
TRT readout is required due to its dual purpose as a tracking and particle identification
detector and the nature of reading out a wire/straw detector.

The TRT ByteStream consists of data collected in a 75 ns window following the start
of and LHC bunch-crossing which generates a LVL1 accept. Hence, the readout period
is three times the nominal 25 ns time-slice separation between LHC beam crossings. Two
species of readout data exist: a tracking output using a 200 eV discriminator thresh-
old (referred to as Low Threshold) and a particle identification output using a 5000 eV
discriminator threshold (referred to as High Threshold).

In recording the low threshold signal, eight bits for each of three consecutive time-slices
are used. Each 25 ns time-slice is divided into eight 3.125 ns wide bins. A bit is set to one
if the low threshold discriminator was on during the corresponding time bin. This leads to
24 bits being used to record the low threshold data. In recording the high threshold signal
one bit for each time-slice is used. The bit is set to one if the high threshold discriminator
was on at any time during the time-slice. This leads to a total 3 bits being used to record
the high threshold data.

The resulting set of 27 bits is referred to as ”full encoding” and contains data for
three time-slices with the earliest data occupying the highest order bits as show in the
Raw Data rows in Table 5.

The TRT detector uses a total of 256 RODs operating in parallel to process the TRT
data. The data received by a single ROD comes from either a stack of three barrel modules
or parts of many end-cap wheels. Each ROD is connected to a φ segment of the detector
with different φ granularities for the barrel and endcaps. The barrel is divided into 32 φ
readout segments for each end of the detector. The φ segment reads out a set of three
modules (i.e., type 1, type 2, and type 3) and contains 329 + 520 + 793 = 1642 channels.
The end-cap is divided into 96 readout segments for each end of the detector. Each of
segment spans a wedge-shaped region in all layers at one end of the TRT. For the type A
and B wheels, the 1/96 of 2π straw layer consists of 8 channels, and for the type C wheels
it is 6 channels. Since there are 6 type A wheels with 16 layers each, 8 type B wheels
with 8 layers each and 4 type C wheels with 16 layers each, the φ segment contains 1664
channels.

The content of a ROD fragment is shown in Table 5. A single ROD supplies data to a
single ROB, therefore the TRT uses the same number (i.e., 256) of ROBs as RODs. The
ROB output buffer format is a single encapsulation of the ROD output buffer data. It
consists of a header, 4 × 16 = 64 blocks of information, and a trailer. The ROB output
buffer serves as the input to the LVL2 trigger processor.

Further details can be found in Ref. [13].
4.1.2 The Calorimeters

Liquid Argon

The package LArCalorimeter/LArCnv/LArByteStream provides conversion of LArRawChannels into ATLAS raw data in ByteStream format, and conversion from BS to LArRawChannels or LArCells.

The encoding to ByteStream from LArRawChannel is implemented in the class LArROD_Encoder. It can take a group of LArRawChannels in one ROD and turn them into an ROD block. Conversely, LArROD_Decoder class provides conversion from a ROD fragment to either LArRawChannels or LArCells. The content of each ROD fragment is given in Table 6.

Each ROD is assigned a ROD Identifier. The format of the Identifier is defined by TDAQ group. The assignment of the Identifier is done by the class Hid2RESrcID. The current mapping is that all front-end boards (FEBs) in one Feedthrough is put into one ROD. This class also provides transformation of ROD Identifier to ROS Identifier, as it is needed for assembling higher-level Fragments, with FullEventAssembler in ByteStreamCnvSvc package.

The LAr RDO, LArRawChannels, are stored in LArRawChannelCollection. These collections can be accessed through LArRawChannelContainer, a subclass of IdentifiableContainer. Currently, the LArRawChannels in a LArRawCollection is ordered according to temporary definition of LAr Trigger Tower (TT) Identifier as defined in LArIdentifier package. This provides access to RoIs expressed as a group of TT. To access all LArRawChannels in a RoI, the user can use the selector class, LArTT_Selected in LArRawUtils package, given a LArRawChannelContainer, and a vector of TT Identifiers.

Even though in the standard Offline data model LArCells are stored in CaloCellContainer in granularity of subdetectors (i.e., EM, HEC, FCAL), a special Collection/Container similar to that for LArRawChannels is provided for LVL2 algorithms since it is desirable to create LArCell directly from ByteStream for efficiency reasons. The calibration of LArCells, normally applied by algorithms in LArCellRec packages, have to be applied. This is achieved by defining a LArCellCorrection base class, a subclass of AlgTool, in LArRecUtils package, and the converter will retrieve a set of these AlgTools from ToolSvc to apply the corrections to the LArCells. This guarantees the LArCells created by the ByteStream converter is identical to the LArCells created in the offline data processing sequence.

There are a few converters in this package. There are two converters for the containers: LArRawChannelContByteStreamCnv and LArCellIDCByteStreamCnv. There is one template Tool class for the Collections: both LArRawChannel and LArCell. LArRawChannelContByteStreamCnv converts all LArRawChannels in all Collections into RODFragments. A template collection converter is provided in ByteStreamCnvSvc package, and each detector is supposed to provide an AlgTool, to be used by the templated collection converter. The AlgTool for LAr is LArRawChannelCollByteStreamTool.
Liquid Argon

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEB_ID</td>
<td>Identifier for FEB</td>
</tr>
<tr>
<td>WordCount</td>
<td>32 bits</td>
</tr>
<tr>
<td>Map1</td>
<td>$4 \times 32$ bits one bit for each of the 128 channels for the presence of t and Q data</td>
</tr>
<tr>
<td>Map2</td>
<td>$4 \times 32$ bits one bit for each of the 128 channels for the presence of Energy data</td>
</tr>
<tr>
<td>Block of RawDataWords</td>
<td>RawDataWord = :E:t:Q:</td>
</tr>
<tr>
<td>E</td>
<td>16 bits (if not suppressed)</td>
</tr>
<tr>
<td></td>
<td>1 sign bit/13 bits/2 range bits</td>
</tr>
<tr>
<td></td>
<td>range 0: LSB 1 MeV,</td>
</tr>
<tr>
<td></td>
<td>range 1: LSB 8 MeV,</td>
</tr>
<tr>
<td></td>
<td>range 2: LSB 64 MeV,</td>
</tr>
<tr>
<td></td>
<td>range 3: LSB 512 MeV</td>
</tr>
<tr>
<td>t</td>
<td>16 bits (if not suppressed)</td>
</tr>
<tr>
<td></td>
<td>unsigned int (LSB 5 ps)</td>
</tr>
<tr>
<td>Q</td>
<td>1 bits</td>
</tr>
<tr>
<td></td>
<td>unsigned int</td>
</tr>
</tbody>
</table>

Tile

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WordCount</td>
<td>32 bits</td>
</tr>
<tr>
<td>Frag type+ID</td>
<td>32 bits</td>
</tr>
<tr>
<td></td>
<td>upper 16 bits: frag type</td>
</tr>
<tr>
<td></td>
<td>0 - raw digits (a la testbeam)</td>
</tr>
<tr>
<td></td>
<td>3 - energy,time,quality (this sub-fragment)</td>
</tr>
<tr>
<td></td>
<td>lower 16 bits is frag ID:</td>
</tr>
<tr>
<td></td>
<td>8 bits - ROS number (1-4)</td>
</tr>
<tr>
<td></td>
<td>8 bits - module number (0-63)</td>
</tr>
<tr>
<td>Map</td>
<td>$2 \times 32$ bits one bit for each of the 48 channels for the presence of this channel in fragment</td>
</tr>
<tr>
<td>Block of RawDataWords</td>
<td>RawDataWord = :E:t:Q:</td>
</tr>
<tr>
<td>E</td>
<td>16 bits</td>
</tr>
<tr>
<td></td>
<td>1 sign bit/1 gain bit/14 data bits</td>
</tr>
<tr>
<td></td>
<td>gain = 0 - low gain</td>
</tr>
<tr>
<td></td>
<td>gain = 1 - high gain</td>
</tr>
<tr>
<td></td>
<td>scale factor = 16</td>
</tr>
<tr>
<td></td>
<td>full range $\pm 1023$ roughly:</td>
</tr>
<tr>
<td></td>
<td>800 GeV for low gain (LSB 50 MeV)</td>
</tr>
<tr>
<td></td>
<td>12.5 GeV for high gain (LSB 0.8 MeV)</td>
</tr>
<tr>
<td>t</td>
<td>16 bits</td>
</tr>
<tr>
<td></td>
<td>signed short</td>
</tr>
<tr>
<td></td>
<td>scale factor = 256 (LSB 2-8 ns)</td>
</tr>
<tr>
<td></td>
<td>full range $\pm 127$ ns</td>
</tr>
<tr>
<td>Q</td>
<td>16 bits</td>
</tr>
<tr>
<td></td>
<td>signed short</td>
</tr>
<tr>
<td></td>
<td>scale factor = 32768 (LSB 2-15)</td>
</tr>
<tr>
<td></td>
<td>full range $\pm 1$</td>
</tr>
</tbody>
</table>

Table 6: The content of the LAr and Tile ROD Fragments.
which creates LArRawChannels or LArCells in one Collection from ROBfragments. Using a mapping class, Hid2RESrcID, LArRawChannelContByteStreamCnv also provides ByteStreamAddress objects for the Collections. The conversion is implemented in AlgTool classes, such that it can be used outside of the converters.

Tile Calorimeter

This subsection summarizes the status of ByteStream output for the Tile Calorimeter Raw Data Object (RDO). The package TileCalorimeter/TileSvc/TileByteStream provides conversion of TileRawChannels into ATLAS raw data in ByteStream format, and conversion from ByteStream to TileRawChannels or TileCells.

The granularity of the Tile Calorimeter from readout point of view is the following. There are 4 independent systems (ROS) which correspond to 2 halves of barrel and 2 extended barrels. Every such a cylinder contains 64 modules (or half-modules in case of barrel) and output from every 8 modules goes into one ROD unit. Therefore one ROD fragment contains 8 sub-fragments which correspond to 8 modules. Also, there is a possibility to have different kind of sub-fragments, for example sub-fragment with raw digits (25 ns time slices) and sub-fragment with reconstructed quantities like energy, time, quality. There is no limit on total number of sub-fragments in one Tile ROD fragment: they are put one after another and identified by 32-bit words. Upper half of this word is fragment type and lower half is fragment ID The fragment ID itself can be split into 2 parts: 8 bits ROS number (range 1-4) and 8 bits module number (range 0-63).

For the moment only one type of sub-fragment is implemented, namely sub-fragment which contains amplitude, time and quality. All these values are member elements of TileRawChannel. The amplitude in TileRawChannel is expressed in ADC counts (10 bits range), time is in nanoseconds and quality is the number in the range 0-1. Although it might not be necessary, all variables are treated and signed. The content of the this fragment is given in Table 6.

The encoding to ByteStream from TileRawChannel is implemented in the class TileROD_Encoder. It can take a group of TileRawChannels in one ROD and turn them into an ROD block. Conversely, TileROD_Decoder class provides conversion from a ROD fragment to either TileRawChannels or TileCells.

Each ROD is assigned a ROD ID. The format of the ID is defined by TDAQ group. The assignment of the ID is done by the class TileHid2RESrcID. This class also provides transformation of ROD ID to ROS ID, as it is needed for assembling higher level Fragments, with FullEventAssembler in ByteStreamCnvSvc package. As it was said before, the current mapping is that every 8 modules are put into one ROD and there are 4 ROSes in TileCal with 8 RODs per ROS.

The Tile RDO, TileRawChannels, are stored in TileRawChannelCollection. These collections can be accessed through TileRawChannelContainer, a subclass of
IdentifiableContainer. There are 256 Collections in one Container in total, one Collection holds only channels which correspond to one Tile Calorimeter extended barrel module or one half of barrel module.

Even though in the standard Offline data model TileCells are stored in CaloCellContainer, a special Collection/Container similar to that for TileRawChannels is provided for LVL2 algorithms since it is desirable to create TileCell directly from ByteStream for efficiency reasons. The calibration of TileCells, normally applied by TileRawChannelToCell algorithm in TileRecAlg package, have to be applied. A temporary solution for calibration is to hold all calibration constants in the TileInfo class. TileInfo is stored in TDS and then retrieved by any algorithm when needed. Both TileRawChannelToCell and TileROD_Decoder are using the same constants from TileInfo class. This guarantees the TileCells created by the ByteStream converter is identical to the TileCells created in the offline data processing sequence.

There is one currently unresolved issue. TileCells are built from two TileRawChannels. In the case of the D0 cell (i.e., the central cell of last barrel sampling) two TileRawChannels for this cell belong to different collections. For the moment, two halves of the D0 cell will be created in two different Collections.

Another problem is correspondence between TileCells and Trigger Towers. Almost all cells of D-sampling (with $\Delta \eta = 0.2$) are split in two trigger towers (with $\Delta \eta = 0.1$ granularity). One channel of such cells belongs to one trigger tower, another one to another trigger tower. For the moment, only one of two channels is taken into account when the "TileCell to CaloTriggerTower" map is created.

There are a few converters in the TileByteStream package. There are two converters for the containers: TileRawChannelContByteStreamCnv and TileCellIDCByteStreamCnv. There is one template Tool class for the Collections, both TileRawChannel and TileCell.

TileRawChannelContByteStreamCnv converts all TileRawChannels in all Collections into ROD-Fragments. A template collection converter is provided in ByteStreamCnvSvc package, and each detector is supposed to provide an AlgTool, to be used by the templated collection converter. The AlgTool for Tile is TileRawChannelCollByteStreamTool, which creates TileRawChannels or TileCells in one Collection from ROB-fragments. Although there are 8 sub-fragments in one ROD-fragment and these sub-fragments can be unpacked in 8 Collections, only one Collection is filled in TileRawChannelCollByteStreamTool on every request.

Using a mapping class, TileHid2RESrcID, TileRawChannelContByteStreamCnv also provides ByteStreamAddress objects for the Collections.

The conversion is implemented in GAUDI AlgTool classes, such that it can be used outside of the converters.
4.1.3 The Muon Spectrometer

Resistive Plate Chambers (RPC)

The definition of RDOs for the RPC is complicated by the way the RPC read-out is organized. The RPCs are trigger chambers, and the organization of their read-out is oriented toward this task: the read-out structure does not reflect any easily identifiable geometrical structure, like for example, a single RPC chamber with its strips.

The primary task of the RPC trigger chambers is to record hits in the \textit{pivot plane} (i.e., the middle one of the three RPC planes) that coincide in time either with hits in the RPC plane closest to the Interaction Region (“confirm plane low $p_T$”) or with hits in the RPC plane farthest from the Interaction Region (“confirm plane high $p_T$”). The RPC strips are read out via coincidence matrices (CMs). The \( \eta \) strips of the pivot plane and the \( \eta \) strips of the low $p_T$ (high $p_T$) confirm plane are connected to the low $p_T$ (high $p_T$) CMs and likewise for the \( \phi \) strips. The CMs are grouped together in PADs. Each PAD reads out all coincidences between RPC planes in a 3-dimensional volume. Neighboring PADs overlap in \( \eta \) on the two confirm planes but not on the pivot plane. There, a PAD covers half a RPC chamber. Per logical sector, there can be up to 8 PADs. The cabling of the RPC strips to the CMs is such that one CM reads out either the \( \eta \) or the \( \phi \) strips in half a PAD. Consequently, one PAD reads out two low $p_T \ \eta$ CMs, two low $p_T \ \phi$ CMs, two high $p_T \ \eta$ CMs and two high $p_T \ \phi$ CMS. A more detailed description is available elsewhere [14].

The RPC ByteStream format reflects the read-out subdivision in PADs that are subdivided in CMs that read-out fired CM channels. Detailed documentation on the RPC ByteStream Format is available elsewhere [15]. Note that there is no simple one-to-one correspondence between RPC strips and CM channels. In order to determine which RPC strip fired given which channel fired in a, say, low $p_T \ \eta$ CM, a cable map and the information in the high $p_T \ \phi$ CM of the same PAD is needed. The latter information is necessary to resolve the ambiguities caused by the fact that more than one RPC strip can be connected to the same CM read-out channel.

There will be three different types of RPC RDOs:

1. Bare RDOs;
2. RDOs with prepared data, first variety;
3. RDOs with prepared data, second variety (“digits”).

The bare RDOs are the minimal object representation of the information contained in the bytestream. Three classes are used: \texttt{RpcPad}, \texttt{RpcCoinMatrix}, and \texttt{RpcFiredChannel}; these are in the CVS repository under: \texttt{offline/MuonSpectrometer/MuonDigitContainer}.

\texttt{RpcPad} is a \texttt{DataVector} of \texttt{RpcCoinMatrix} which, in turn, is a \texttt{DataVector} of \texttt{RpcFiredChannel}. \texttt{RpcFiredChannel} contains the online identifiers of the CM chan-
nels (not the RPC strips) that fired. Objects of type **RpcPad** are the collections that are stored in an IDC. An object of type **RpcPad** is the smallest unit of information a HLT algorithm can retrieve from StoreGate (see Sections 2.4.2 and 4.5.1 with a single request. Accordingly, an object of type **RpcPad** is uniquely identified by means of an Offline Identifier as defined in Ref. [16]. These Offline Identifiers follow the geometrical RPC structure, not the read-out structure. Since the projections of the PADs onto the pivot plane don’t overlap, the location where the PAD intercepts the pivot plane can be identified uniquely with an Offline Identifier and is used to identify an object of type **RpcPad**.

The class hierarchy in the first variety of the RDOs with prepared data is identical to the one in the bare RDOs. The only difference is that **RpcFiredChannel** contains the information which RPC strips fired (their Offline Identifier). To arrive at the RPC strip that fired access to the cable map (*i.e.*, to information external to the ByteStream), is needed. The HLT algorithm muFast (see Section 7.1.7) will make use mainly of this type of RDOs.

The second variety of RDOs with prepared data corresponds to RPC digits and are used by EF algorithms like Moore. For more information see Ref. [17].

### 4.2 Identifiers

In a the most general sense, an Identifier provides a label for a region of the ATLAS Detector and provides an organizing principle for event data associated with it. There are various types of Identifiers.

- **Offline Identifier** There are three kinds of Offline Identifiers.
  
  - **Identifier**: Also referred to as an *Extended Identifier*. A hierarchical vectors of short ints.
  
  - **Identifier32**: Also referred to as an *Compact Identifier*. These are packed 32-bit representations of Identifiers.

  - **IdentifierHash**: A Hash representation of an Identifier or Identifier 32. It encodes a 32-bit index which can be used to look-up *Identifiable* objects stores in a simple vector. It is intended to be a continuous hash (*i.e.*, it runs from 0 to \(N-1\), where there are \(N\) different possible values for an Identifier(32) within a specific context) [18]. This allows constant-time look-ups.

- **Online Identifier** Assumed to be a ROB-ID unless otherwise indicated.

It is intended that the HLT RegionSelector described in Section 2.4.1 return **IdentifierHashes** within a RoI. Also, via a Manager class, DetectorElements are accessible from the DetectorDescription via **IdentifierHashes**.
4.3 Reconstruction Input Objects (RIOs)

RIOs are the most upstream objects that algorithms typically interact with (as opposed to RDOs or RodInputDigits). For example, the Pixel and SCT RIOs are Clusters (see this document’s Appendix for definitions) and the TRT RIO is the drift circle of a straw.

4.3.1 Inner Detector

Similarly to the RDOs, the implementation of the RIOs makes use of the IdentifiableContainer base class, and the collections are also according to the granularity of DetectorElements.

For Pixel and SCT, there are currently two implementations of the Cluster class: one used for EF and Offline and one used for LVL2. The one used at EF has Pixel and SCT sharing the same class. For LVL2 there is a common structure for Pixel, SCT and TRT, but they all have their own concrete classes. For Pixel and SCT there is a base class used for LVL2. There is also an Extended class which could potentially be used at EF (which inherits from the LVL2 base class) in the future. Both LVL2 and EF set of cluster classes contain a list of RDO identifiers from which the cluster is built. The number of member functions is limited in both set of classes and the member functions follow the InnerDetector Requirements [19]. It is assumed that in the future there will be only one set of RIO classes to be used for LVL2, EF, and Offline.

In the case of the TRT, the same classes are used for LVL2, EF, and Offline: those classes are the DriftCircle classes part of the set of classes that are also used at LVL2 for Pixel and SCT. The granularity of the TRT RIO is the same as for the RDO: that of a straw, thus the RIO contains an identifier which is the offline identifier for a straw. In the case of the RDO the straw information is uncalibrated and is just the direct content of the detector output, while in the case of the RIO the straw information is calibrated: out of the drift time, a drift radius is obtained. For now, the drift function applied is the same for all straws. In the future the constants that go into the parametrization of this drift function will come from the Interval of Validity Service [20].

Given the lack of convergence in the Offline environment for SpacePoints, a LVL2-specific SpacePoint class has been developed and is described in Section 2.5.1.

4.3.2 The Calorimeters

For the Calorimeters, the RIOs are calibrated calorimeter cells (LArCells and TileCells), imported from the offline reconstruction.

Both LArCells and TileCells have CaloCell as a common base class which represents the basic nature of an observation in the ATLAS calorimeters an energy, position, time and quality. A CaloCell has been calibrated so that energy() returns the physical energy deposit in the cell with units of GeV, but without any kind of leakage corrections.
Figure 8: A UML class diagram of the Calorimeter RIOs LArCell and TileCell along with their common base class CaloCell.

Time represents when the feature extraction thinks the deposit occurred, in nanoseconds, relative to the trigger. It ought to be zero for good hits. Quality reflects how well the input to the feature extraction system matched the signal model on which the feature extraction algorithm is based. It is a number, from zero to 1, giving the significance of the hypothesis that the actual signal is a sampling of the signal model (i.e., it is the integral of a probability distribution from – inf up to an observed value of a test statistic and ought to be uniformly distributed from [0,1] if the hypothesis is correct) [21].

A UML class diagram of LArCell and TileCell is given in Figure 8.
4.3.3 The Muon Spectrometer

For the barrel Muon Spectrometer it was found expedient to use in part RDOs instead of RIOs as input to the HLT muon selection algorithms. The RDOs are organized inside the transient event store as identifiable collections and can be accessed in the same way ROIs can be accessed.

For the MDTs, both for LVL2 and Event Filter one RDO corresponds to a DetectorElement which is a MDT chamber. A RDO contains the information of a collection of MDT read-out channels. The information per read-out channel, i.e. per MDT tube, is the time of the leading and falling edge of the MDT tube pulse, from which the uncalibrated drift time can be calculated.

In the case of the barrel trigger chambers, the RPCs, LVL2 uses RDOs and the Event Filter RIOs.

The definition of RPC RDOs is complicated by the fact that the RPCs are trigger chambers. Their read-out is optimized for the trigger task and does not reflect any easily identifiable geometrical structure, as for example a RPC chamber with its strips. Consequently, the class structure of RPC RDOs follows the read-out structure ordering by PADs and CMAs. A RPC RDO corresponds to a PAD, i.e. is a collection of CMAs which in turn are collections of fired CMA channels. Each PAD contains the information of all coincidences between the inner-our outermost RPC station with the pivot plane within a three-dimensional volume. Neighboring volumes overlap in eta in the inner- and outermost layers, but not in the pivot plane. This allows to assign a unique identifier to each PAD, i.e. RDO.

One RPC RIO corresponds to a DetectorElement which is a RPC chamber. A RPC RIO contains the information of a collection of RPC strips that fired. There is no simple correspondence between RPC strips and RPC read-out channels. In order to translate a fired RPC read-out channel, which is a CMA read-out channel, into a RPC strip, a cable map and processing of the information of the CMAs for the opposite view is required.

More information can be found in Ref. [22].

4.4 Reconstruction Output Physics Objects (ROPOs)

4.4.1 Track

A track is, in general, an object containing a parameterization of a hypothesized particle trajectory through space relating groups of RIOs and/or SpacePoints together. A Track trajectory consists of three position, two direction, and one curvature\textsuperscript{14} parameters. If a

\textsuperscript{14}The use of curvature assumes a homogenous magnetic field in which case this quantity is constant. For ATLAS and its significantly inhomogenous magnetic field in the endcap region of the Inner Detector and in the Muon Spectrometer, this parameter may be replaced by an invariant quantity such as $\text{charge}/p$. 
Figure 9: A UML class diagram of the LVL2 specific Track class `TrigInDetTrack` and associated classes.

A proposed uniform Track class exists for LVL2 algorithms, the `TrigInDetTrack` class [23]; it can be found in the CVS repository under `Trigger/TrigEvent/TrigInDetEvent`. A UML class diagram of `TrigInDetTrack` and associated classes is shown in Figure 9. No such uniform Track class yet exists in the Offline environment.\(^{15}\)

\(^{15}\)There are of course Track classes defined internally within ORPs such as iPatRec and xKalman++. 

Track is evaluated at an intersecting surface, there are five parameters and a covariance matrix.
4.5 Offline Core Software

4.5.1 StoreGate

StoreGate is discussed in detail elsewhere [6].

4.5.2 IdentifiableContainer (IDC)

IdentifiableContainer (IDC) is discussed in detail elsewhere [7]. Since IDC provides the view and organizing principle for event data, system performance is a vital issue. Some system performance aspects of IDC have been measured [8] which suggest acceptable performance at LVL2 for the numbers of collections anticipated by making use of the RoI mechanism. More refined tests must wait until it operates within a full vertical HLT slice within the testbeds.
5 Coding Guidelines

5.1 General Coding Rules

General coding rules that all ATLAS software source code should comply with are described elsewhere [24]. Table 7 summarizes the categories of these rules. Code in the Trigger area of the CVS repository will have an automatic RuleChecker run on them routinely and a list of the number of occurrences of a rule violation summarized in a spreadsheet.

<table>
<thead>
<tr>
<th>Rule Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NF</td>
<td>File naming</td>
</tr>
<tr>
<td>NM</td>
<td>General naming</td>
</tr>
<tr>
<td>NI</td>
<td>Illegal naming</td>
</tr>
<tr>
<td>NC</td>
<td>Naming conventions</td>
</tr>
<tr>
<td>NC</td>
<td>Naming conventions</td>
</tr>
<tr>
<td>CO</td>
<td>Code organization</td>
</tr>
<tr>
<td>CF</td>
<td>Control flow</td>
</tr>
<tr>
<td>CL</td>
<td>Object life cycle</td>
</tr>
<tr>
<td>CC</td>
<td>Conversions</td>
</tr>
<tr>
<td>CI</td>
<td>Class interface</td>
</tr>
<tr>
<td>CN</td>
<td>New and Delete</td>
</tr>
<tr>
<td>CS</td>
<td>Static and Global objects</td>
</tr>
<tr>
<td>CB</td>
<td>Object oriented programming</td>
</tr>
<tr>
<td>CE</td>
<td>Assertions and errors</td>
</tr>
<tr>
<td>CH</td>
<td>Error handling</td>
</tr>
<tr>
<td>CA</td>
<td>Parts of C++ to avoid</td>
</tr>
<tr>
<td>CR</td>
<td>Readability and maintenance</td>
</tr>
<tr>
<td>CP</td>
<td>Portability</td>
</tr>
<tr>
<td>CT</td>
<td>Templates</td>
</tr>
<tr>
<td>SG</td>
<td>General style</td>
</tr>
<tr>
<td>SC</td>
<td>Comments</td>
</tr>
</tbody>
</table>

Table 7: A summary of the categories for the general coding rules all ATLAS software source code should comply. The complete set of rules is available elsewhere [24].

5.2 Level 2 Guidelines

When porting algorithms from the Offline environment or when developing new code, a number of guidelines should be followed to facilitate the integration with the HLT framework and testbeds. Obviously code should be as bug-free as possible to minimise
the time lost during integration. However, this requirement is absolutely essential due to the non-trivial nature of debugging in the Online environment, the complexity of the setup, and especially the multithreaded nature of the LVL2 Processing Unit (L2PU).

5.2.1 General Design and Coding Issues

L2CG.1.0 Data Requests. Data Requests are extremely time-consuming due to the limited network bandwidth and resources available at LVL2. Raw detector data contained in ROBs/RODs that define a RoI should be requested at once. For most algorithm development purposes, this is handled behind-the-scenes via the London Scheme (see Section 2.4.2).

L2CG.1.1 Access to Configuration Data. There is no access to configuration data during a LVL2 run. Thus, algorithms, tools, and services must not try to scan configuration parameters at run-time. There should be a clear separation between code executed once before the start of a run and multiple times within the Event Loop. Most importantly, no initialization should take place within the Event Loop, especially for the first event. File and database access (e.g., calibration) is prohibited within the Event Loop. All services, algorithms, and tools should be “touched” during initialization of the L2PU.

L2CG.1.2 Thread Safety. Algorithms for LVL2 will be run in the L2PU which is a multithreaded environment.\(^\text{16}\) Hence, all algorithms, tools, or services must be thread-safe. All code should be

\begin{itemize}
  \item L2CG.1.2.1 without global variables;
  \item L2CG.1.2.2 without singletons [25];
  \item L2CG.1.2.3 with the keyword const as much as possible, mainly in the declaration of methods and parameters passed to methods. For example, use
\end{itemize}

```cpp
class A { public:
  int get_int (void) const; //GOOD!
  ...
}
```

instead of

```cpp
class A { public:
  int get_int (void); //BAD!
  ...
}
```

\(^{16}\)To have a better understanding of multithreaded programming, refer to http://cern.ch/rabello/work/cern/threads/index.html.
when you are not altering an object of type `class A`. Also, use

```cpp
float do_something (const A& object); //GOOD
```

instead of

```cpp
float do_something (A& object); //BAD
```

so that the compiler guarantees that the `const` property of the objects.

L2CG.1.3 Debug and Status Information. The inclusion of debug and status information during a run may quickly hamper the performance. Use the GAUDI services for debugging, control, and error reporting; these will be diverted to the corresponding Online Services. The use of `cout/cerr`, or any other file or database access is prohibited. Algorithms should only processes data during run phase.

L2CG.1.4 Data Copying. Data copying should be minimized. Data copying (even in memory) is time-consuming. Pass data by reference, and reduce the number of data copies as much as possible. A few guidelines are

L2CG.1.4.1 Pass by `const` reference and avoid pointers because they are error-prone;

L2CG.1.4.2 When doing data conversion, use references to the starting objects instead of transforming the data and restoring which takes time;

L2CG.1.4.3 When filling `vectors`, make a `vector.reserve()` before proceeding into `push_back();` this saves execution time by avoiding reshuffling of data because of insufficient space allocation;

L2CG.1.4.4 Use caution when using other `std::containers` since they are often problematic.

L2CG.1.5 New Memory Allocation. Calling `new/malloc` takes a considerable amount of time. Hence, avoid allocating objects on the heap. Minimize as much as possible the calls to these routines. Instead, make use of local scope variables and only exchange minimal information between objects and methods. Avoid exchanging heap allocated objects. Note that objects allocated as local variables are allocated on the stack during the program initialisation or library loading and do not represent a large overhead.

---

17Recent measurements indicate a rate for a `memcpy()` operation of around 400 MBytes/s (with cutting-edge technology). Scaling this to the 10 ms average LVL2 latency precludes copying more than 4 MBytes of data per event. However, even this reduced estimate is not realistic since we do not have cutting-edge computers and the vast majority of algorithmic processing would occur elsewhere.
**L2CG.1.6 Configuration.** Configuration is problematic. For the TDR, it has been suggested that algorithms within the PESA framework would configure by using `jobOptions` files. The PESA developer should not, though, presume that all offline packages will be available during configuration. It is advisable that the developer keeps track of the `jobOptions` files that an algorithm requires.

### 5.2.2 Athena/GAUDI Design and Coding Issues

For Athena algorithms and services, some of the Athena/GAUDI base classes are over-written to provide the creation, instantiation, and connection of algorithms and services to the different threads in an automatic and transparent way. Hence, algorithms should whenever possible use the methods provided by these base classes.

**L2CG.2.1 Algorithms**

**L2CG.2.1.1** Obtain the pointer to a required service with the method provided by the base class

```
template <class T>
StatusCode service(const std::string& name,
T*& svc, bool createIf=false)
```

and do not create services on the fly (`createIf = true`) since they may require data for initialization which are not available during algorithm execution.

**L2CG.2.1.2** Obtain pointers to typical framework services with the accessor functions provided by the base class, such as

```
IAuditorSvc* auditorSvc() const;
IChronoStatSvc* chronoSvc() const;
IChronoStatSvc* chronoStatService() const;
IDataProviderSvc* detSvc() const;
IDataProviderSvc* detDataService() const;
IConversionSvc* detCnvSvc() const;
IConversionSvc* detDataCnvService() const;
IDataProviderSvc* eventSvc() const;
IDataProviderSvc* eventDataService() const;
IConversionSvc* eventCnvSvc() const;
IConversionSvc* eventDataCnvService() const;
IHistogramSvc* histoSvc() const;
IHistogramSvc* histogramDataService() const;
IMessageSvc* msgSvc() const;
```
INTupleSvc* ntupleSvc() const;
INTupleSvc* ntupleService() const;
IRndmGenSvc* randSvc() const;
IToolSvc* toolSvc() const;

L2CG.2.1.3 To get access to the StoreGateSvc, do not use StoreGate::instance() since this is a singleton which would not work in the multithreaded L2PU environment. It also requires the existence of ActiveStore which is not supported in LVL2. Use the service method mentioned above to get a pointer to the StoreGateSvc.

L2CG.2.1.4 There should be no creation of configuration, geometry, or database objects which are used during algorithm processing triggered by the processing of the first event.

L2CG.2.1.5 If you create and use sub-algorithms, use the methods

```cpp
StatusCode createSubAlgorithm(const std::string& type, const std::string& name, Algorithm* & pSubAlg);
std::vector<Algorithm*>* subAlgorithms() const;
```

provided by the base class.

L2CG.2.2 Services

L2CG.2.2.1 Retrieve and create all necessary database and configuration information in the initialize() step of the service.

L2CG.2.2.2 For access to other services use the methods provided by the base class.

L2CG.2.3 Tools

L2CG.2.3.1 When tools are used, it is necessary to distinguish between tools which are modifying data during execution and tools which provide read-only data. In the first case, the tool should be used only as a ”private tool.” In the second case, the tool can be used as a ”shared tool” (see the Athena/GAUDI manuals):

```cpp
private tool retrieval:
StatusCode sc = toolSvc()->retrieveTool("My_Tool", m_myTool, this);

shared tool retrieval:
StatusCode sc = toolSvc()->retrieveTool("My_Tool", m_myTool);
```
L2CG.2.3.2 If you create a new shared tool, make all data access methods which can be used during execution \texttt{const}.

5.2.3 Testing

L2CG.3.1 Optimization of execution speed is vital for LVL2. The current time budget is roughly 10 ms/event average to: produce a LVL2 result, extract from it the LVL2 decision, \textit{and} send it back to the LVL2 supervisor. The testbed activities will help estimate the overhead for collecting data for algorithms. The L2CG.1 series of guidelines above will help make an algorithm as fast as it can be inherently. Further performance improvements rely upon optimizing the algorithm itself, not the implementation.

L2CG.3.2 Due to the high LVL2 event rate, even minor and rarely occuring bugs are very disruptive for LVL2. A very slow memory leak which in an Offline context may not be problematic is much more evident and significant when run at the LVL2 expected event rate of 100 kHz. Since each L2PU is multithreaded and not a set of separated processors, bugs may also hang-up the whole processor, not only the worker thread. Hence, again, it is imperative that LVL2 code should be as bug-free as possible and contain no memory leaks.

The HLT infrastructure team asks that algorithms to be be tested with at least 100,000 events which in fact represents only 1 s of running under realistic conditions. The events should be representative of a realistic physics scenario and match the ones that will be available for integration studies. This will help guarantee that code will not disrupt LVL2 execution or performance. Note that if only a limited set of data are available, it is acceptable to run over the same dataset multiple times to achieve this 100,000 event goal.

5.2.4 Parameters

This subsection provides a list of parameters HLT infrastructure team requires for each algorithm before integration starts. Items L2CG.4.1 to L2CG.4.3 should come in a file named \texttt{description.txt} in your package directory, under the \texttt{doc} sub-directory. Item L2CG.4.4 should come also in a simple text file, at the same sub-directory as before, with the name \texttt{dependencies.txt}. Item L2CG.4.5 should be your normal \texttt{jobOptions.txt}. Avoid following other documentation schemas because this complicates integration work.

L2CG.4.1 Algorithm name should be provided;

L2CG.4.2 A simple description of what the algorithm does, data that is retrieved, data that is produced, \textit{etc.} should be provided;
L2CG.4.3 The time to execute the algorithm on a modern CPU should be provided;

L2CG.4.4 The size of the .so file to determine the size of the algorithm library should be provided.

L2CG.4.5 A dependency map for algorithm library should be provided. This can be produced by running `ldd` on the library(ies) created by your package(s). Running `ldd` will present you with a list of libraries your algorithm library needs to load at execution time and their location. This will help you to understand the dependencies you bring in with such an algorithm. The dependency map should include the versions of the packages your algorithm depend on.

L2CG.4.6 A dependency map for algorithm configuration should be provided that shows which `jobOptions` are needed. This can be produced by providing a single `jobOptions` file that does not contain any `include` statements. You should comment each variable in this `jobOptions` file and explicitly state if the values you assign override the default values.

5.3 Event Filter Guidelines

In general, Event Filter coding guidelines are a sub-set of those for Level 2.

5.3.1 General Design and Coding Issues

EFCG.1.1 Access to Configuration Data. In the Event Filter, the possibility of updating calibration constants during a run will be open. However, this should not be a general practice, but reserved for special cases where it is needed to guarantee the performance of an algorithm during a run.

EFCG.1.1.1 Hence, for most cases, the L2CG.1.1 recommendation is valid: all initializations shall be done in the `initialize()` method, and certainly not at the first event inside the event loop.

EFCG.1.1.2 The mechanism for updating constants on-the-fly in Athena is not completely established yet nor the amount of network traffic or latency this will introduce in the event processing. One possibility is that it will be based on the Transient Detector Store(TDS), where constants would be refreshed automatically. This is then transparent to the algorithm provided that there are no internal constants that would need to be calculated from the set of constants available in the TDS.

EFCG.1.1.3 On the timescale of the HLT TDR, we will likely not investigate in detail how to accomodate the change of constants during a run, but it is good to keep
this question in mind, especially what reinitialization would be needed by an algorithm if some calibration data are changed.

**EFCG.1.2 Thread Safety.** In the current EF design, the Athena Processing Task (PT) is a standard Offline Athena process that runs a single event loop and instantiates all services, just like in the Offline case. Communication with the EF Dataflow (EFD) and Supervision will trough services instantiated in each process. There is no multithreading inside the Athena PT.

In this respect, there is in principle no restrictions in the use of singletons, global variables, etc. as is outlined in L2CG.1.2. However, the EF would like to recommend respecting these rules, as they are good programming habits would not preclude a multithreaded EF environment in the future.

**EFCG.1.3 Debug and Status Information** Compared to the situation for LVL2 described in L2CG.1.3, the situation is somewhat different in the EF.

- **EFCG.1.3.1** The Athena PT shall not produce any message through use of `cout` or `cerr`. The Athena Messaging Service (MS) should be used instead. It will be interfaced with the Online EFSupervision. Although the message debugging level or severity error messages can be controlled at the EFSupervision, it is better to filter them at the source (i.e., at the Athena level). The filtering level of each specific task will be defined at initialization (and in the future may change during a run).

- **EFCG.1.3.2** In the EF, monitoring information will be collected that will be communicated to EFSupervision via the MS (interfaced to the Online MS) or the Histograming Service (interfaced to Web based histogramming service). The exact amount of information is not yet defined, but this will be typically include the event type accepted by the trigger type and histograms indicating how events are accepted/rejected by each trigger. The amount of messages and histograms should be controlled by levels.

**EFCG.1.4 Data Copying, New Memory Allocation, and Configuration.** These are design issues for which no strong rule can be defined. For external access, the rule is a function of the amount of data, the frequency of operation, and the cost in terms of the latency. However, the event processing should minimize the average event processing time, memory, and network bandwidth, since few resources reduce the overall cost of the system.

**5.3.2 Athena/GAUDI Design and Coding Issues**

**EFCG.2.1 Algorithms.**
EFCG.2.1.1 The EF does not in principle forbid the dynamic creation of new services, but it will depend on the speed of accessing to external database, the amount of information to transfer, and the frequency.

EFCG.2.1.2 The EF also recommends obtaining pointers to typical framework services with the accessor functions provided by base classes (see L2CG.2.1.2).

EFCG.2.1.3 Although in principle access to the `StoreGateSvc` via `StoreGate::instance()` would work, it is recommended to use the service method as advocated in L2CG.2.1.3.

EFCG.2.1.4 There should be no creation of configuration, geometry, or database objects which are used during algorithm processing triggered by the processing of the first event.

EFCG.2.1.5 If you create and use sub-algorithms, use the methods provided by the base class as described in L2CG.2.1.5.

EFCG.2.2 Services. The EF has the same requirements on the use of services as LVL2 does as specified in L2CG.2.2.

Constness of methods, arguments, or returned value shall be specified as strictly as possible. This provides a simple program checking method at compilation time and makes it possible to use special optimizations wherever possible.

EFCG.2.3 Tools. The EF recommends the same guidelines as LVL2 as specified in L2CG.2.3.

5.3.3 Testing

The EF recommends the same guidelines as LVL2 as specified in the L2CG.3.X series of recommendations.

5.3.4 Parameters

The EF does not have a very strict prescription on how to organize configuration information that will be obtained from a database. In the minimal scenario, the database is simply the `jobOptions` file. It will be useful to have the information described in L2CG.4.5 (i.e., a dependency map for the algorithm library), since the Athena PT in the EF will be installed via the RPM Package Manager (RPM) that should be produced with all the necessary libraries.
6 System Performance Measurement Methods

Due to the extreme LHC interaction rate and the limited HLT computing and networking resources, algorithms must be fully optimized for minimal execution time. To do this, several tools for profiling the system performance have been utilized in ATLAS; they are discussed in the following four sub-sections. The choice of tool(s) depends on the information needed and required resolution. Determining the system performance of LVL2 components requires microsecond resolution, while EF components require millisecond resolution.

In general, when making and reporting system performance results, one should carefully note the system parameters of the computer you are running on. Under Linux, they are obtainable by issuing the following commands:

- for information on the parameters of the processor:
  more /proc/cpuinfo

- for information on the parameters of the memory:
  more /proc/meminfo

- for the version of the C++ compiler:
  gcc --version

- to determine if you are making an optimized or debug version of an Athena package:
  cmt show tags | grep Linux-gcc-

It is vital to profile system performance in a consistent and reproducible way using one of the methods described in the following subsections.

6.1 Tuning and Analysis Utilities (TAU)

TAU is a portable profiling package providing microsecond resolution [26]. It provides an analysis environment for parallel C++ multi-threaded applications and dynamically shared libraries. It is a free package available via Ref. [27].

There are two operational modes: User-level and Automatic. With the User-level mode, macros are inserted in the source code manually while in the automatic mode, one uses an interface, the Program Database Toolkit (PDT). A further visualization tool exists: the Event Trace Visualization tool (Vampir). TAU reports time spent exclusively and inclusively in each function, number of times each function was called, and the mean time per call for each function. It can be compiled with different options: profiling, data collection, tracing, and turn-off options.

As with all of these tools, there is a non-zero overhead. For TAU this overhead is between 8 and 40 $\mu$s for the first function invocation, and 1 to 2 $\mu$s for subsequent calls.
6.2 Network Application Logger (NetLogger)

NetLogger provides tools for time measurements of distributed systems with microsecond resolution [28]. It is a free package available via Ref. [29].

Monitoring exists at the network, host and application-level. A simple API generates timestamps with event information at selected places in the code: date, program name, host, unique identifier for the event being logged, additional label an event. There is a visualization Tool for correlating monitoring data (nlv). It provides a standardized message format in ULM (Universal Logger Message). It has been used by DataCollection software developers through an API.

The overhead associated with the use of NetLogger is on the order of 5 $\mu$s.

6.3 GAUDI Chrono and Auditor Services

Within the GAUDI [30] (and hence Athena) framework, the GAUDI Chrono Service exists for code for fine-grained monitoring inside algorithms with ms resolution. The Service ChronoStatSvc is created by default by the Application Manager. To access the service from inside an algorithm use the member function chronoSvc().

One disadvantage of this package is that the implementation of the Chrono Service uses two std::map containers and could generate a performance penalty for very frequent calls.

As an example of the use of the GAUDI Chrono service, one would place in the jobOptions file:

ChronoStatSvc.ChronoPrintOutTable=true;

and inside the code to be profiled:

IChronoStatSvc* svc = chronoSvc();
   // Start profiling under the tag name
   svc->chronoStart("SomeTag");
   // Here some user code exists
   svc->chronoStop("SomeTag");

The GAUDI Auditor Service consists of a set of auditors to provide monitoring various characteristics of the execution of Algorithms: NameAuditor, ChronoAuditor, MemoryAud. They are called immediately before and after each call to each Algorithm instance: initialize(), execute(), finalize(). This provides millisecond precision and is used by Offline algorithm developers.

As an example of the use of the GAUDI Auditor Service, one would place in the jobOptions file:
//Enable the Auditor Service
ApplicationMgr.DLLs += {GaudiAud};

// Enable the NameAuditor and ChronoAuditor
AuditorSvc.Auditors={"NameAuditor", "ChronoAuditor "};

// Enable initialize and finalize auditing for
// MyAlgo (execute by default)
myAlgo.AuditInitialize = true;
myAlgo.AuditFinalize = true;

6.4 TDAQ TimeStamp (TS) Library

The TS library provides nanosecond resolution and an interface to the NetLogger Library that has microsecond resolution.

There are three compile flags to switch: Table 8 shows a summary of their output, resolution, etc.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Output</th>
<th>Precision</th>
<th>Thread Safeness</th>
</tr>
</thead>
<tbody>
<tr>
<td>INST_TS_RAW</td>
<td>counter clock output</td>
<td>ns</td>
<td>not yet</td>
</tr>
<tr>
<td>INST_TS_NL</td>
<td>ULM output</td>
<td>ns</td>
<td>not yet</td>
</tr>
<tr>
<td>INST_NL†</td>
<td>ULM clock output</td>
<td>µs</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 8: A summary of compiler flags for the TimeStamp (TS) library. †This is the NetLogger library.

To be able to use the TS Library you must create a link to an external package that makes a link to the DC repository, and then include a use of this package in your requirements file. In this package you must also include the library you want to use by choosing the corresponding instrumentation flag.

Ref. [31] is a directory that includes the DC libraries needed for this library. For further information, consult Ref. [32].

There are two implementations with a common API within the same CMT package; one can switch between them via compile flag. It generates results based on the time registers of a Pentium CPU, 64-bit Machine Specific Registers (MSR), and it handles string and integer logging of timestamps. At the end of the run, it can convert Read Time Stamp Counter (RDTSC) values in absolute time. The output file is compliant to NetLogger format; hence one can use nlv tool. Currently it is not thread safe. It does allow online processing inside the application.
There is an API in Data Collection (DC) "instrumentation package" of the DC repository, available for different platforms (e.g., RedHat Linux 6.1 and 7.2) and gcc compilers (e.g., 2.95 and 2.96). An external link from the offline to the DC repository can be established to use the TS library; see Ref. [33].

The overhead associated with the use of the TS library is between 100 and 150 ns.

An example of instrumenting an Algorithm with the TS library is

```c++
// In the source code of the Algorithm:
#include "instrumentation/Instrumentation.h"

// In the exec() instance:
// At the beginning to initialize instrumentation for 10000 timestamps
INSTRUMENTINIT("MyAlgorithm","",1,10000,100);

// At critical places where an event should be logged
INSTRUMENT("START_EXEC",0);

INSTRUMENT("END_EXEC",0);
// At the end of the instance to flush memory and close log file
INSTRUMENTTERMINATE();
```

### 6.5 T2TimerSvc

T2TimerSvc is a utility to record timing information. User defined timing statistics can be generated using objects of the class `T2Timer` controlled by the user’s code. The timing mechanism is based on an Athena Service `T2TimerSvc` from which `T2Timer` objects can be requested. The `T2Timers` are controlled (started, stopped etc.) by the requesting algorithm, but the `T2TimerSvc` retains ownership of the `T2Timers`. This means that `T2TimerSvc` is able to provide additional services such as end-of-run printing and providing a list of all `T2Timers` so that, for example, the execution times can be entered into an NTuple.

Since the algorithms request individual `T2Timers`, the timers are started and stopped directly. There is no need for a potentially slow map access by string key, as is the case in the GAUDI Chrono Service.

Further information, including examples of use can be found via Ref. [34]. An example algorithm to fill an Ntuple may be found via Ref. [35].
7 Inventory of Algorithms

A brief description of Algorithms under active development for LVL2 and EF is provided in the subsections below. More details on these algorithms as well as their system and physics performance are anticipated to follow the release of this document as PESA Core Algorithm Group efforts continue.

Reiterating the points made in Section 2.1, unlike in the Offline environment, HLT algorithms must operate in a fundamentally different way. Not only must they allow themselves to be be guided and controlled by the HLT Steering but they also must operate in a seeded manner in order to validate and invalidate specific Trigger Elements. These seeds are provided either by previous levels of the Trigger system (e.g., a LVL1 EMTAUROI would seed LVL2 algorithms for validating e/γ hypotheses) or by previously executed Algorithms at the same Trigger level in the Sequence.

Furthermore, as discussed previously, Algorithms must be extremely robust and minimize their system and network resource requirements due to the short LVL2 and EF latencies.

7.1 Level 2 Algorithms

7.1.1 SiTree

SiTree is a track reconstruction package for LVL2 [36]. It has been extensively tested using previous incarnations of the HLT Software in the LVL2 testbed environment [37].

The basic element that the SiTree algorithm uses is a pair of points in different layers of the SCT and Pixel detectors. This is referred to as a link. Tracks are recognized as chains built from links. The similarity criteria for grouping links into full tracks consists of them having one common point and near equality of track parameters.

The algorithm generates a number of pre-defined roads each of which corresponds to a set of detector layer or disk combinations. Tracks are searched for only within these pre-defined roads. In an RoI guided mode, SiTree uses guidance information to reduce the number of possible links in a region; this includes η, φ, and $E_T$ measured by the Calorimeter.

A least squares fit and a parabola fit (imported from the CTRIG [38] environment) are applied to track candidates to determine the track parameters.

7.1.2 IDSCAN

IDSCAN is a track reconstruction package for LVL2 [39].

It takes as input SpacePoints found in the Pixel and SCT Detectors. A series of sub-algorithms (ZFinder, HitFilter, GroupCleaner, TrackFitter) then processes these and
outputs Tracks and the SpacePoints associated with them.

The ZFinder finds the $z$-position of the primary interaction vertex. The algorithm puts all hits into narrow $\phi$-bins and extrapolates pairs of hits in each bin back to the beam-line, storing the $z$ of intersection in a histogram. It takes as the $z$-position the histogram region with the most entries.

The HitFilter finds groups of hits compatible with Tracks from the $z$ position found by ZFinder. It puts all hits into a histogram binned in $\phi$ and $\eta$. It then finds clusters of hits within this histogram. It creates a group of hits if such a cluster has hits in more than a given number of layers.

The group of hits found by HitFilter is used by GroupCleaner which splits groups into Tracks and removes noise hits from groups. Each triplet of hits forms a potential track for which $p_T$, $\phi_0$, and $d_0$ are calculated. It forms groups from these triplets with similar parameters, applying certain quality cuts. It accepts a track candidate if a group contains enough hits.

Finally, the TrackFitter verifies track candidates and finds the track parameters by using a standard Kalman-fitter-type fitting algorithm adapted from SCTKalman [40]. It returns a list of SpacePoints on the Track, the Track parameters, and an error matrix.

### 7.1.3 PixTrig/SiTrack

SiTrack is a track reconstruction package for LVL2 which extends and upgrades PixTrig [41] (originally developed in the CTRIG [38] framework).

SiTrack takes Pixel and SCT SpacePoints as input and outputs fitted reconstructed Tracks, each storing pointers to the SpacePoints used to build it. SiTrack is implemented as a single main algorithm SiTrack which instances and executes a user defined list of sub-algorithms (chosen among STSpacePointSorting, STMuonVertex, STTrackSeeding, and STThreePointFit).

STSpacePointSorting collects pointers to SpacePoints coming from the Pixel and SCT detectors and sorts them by module address, storing the result in a Standard Template Library (STL) map. This processing step is performed in order to speed-up data access for the other reconstruction sub-algorithms.

STMuonVertex is a primary vertex identification algorithm mostly suitable for low luminosity events with a high $p_T$ muon signature. It is based on track reconstruction inside the LVL1 muon RoI: the most impulsive track is assumed to be the muon candidate and its $z$ impact parameter is taken as the primary vertex position along $z$.

STTrackSeeding, using the sorted SpacePoint map and a Monte Carlo Look-Up Table (MC-LUT) linking each B-layer module to the ones belonging to other logical layers, builds track seeds formed by two SpacePoints and fits them with a straight line; one or more logical layers can be linked to the B-layer, the latter option being particularly useful if
robustness to detector inefficiencies must be improved. If the primary vertex has already
been reconstructed by \texttt{STMuonVertex}, a fraction of fake track seeds can be rejected during
their formation, applying a cut on their \( z \) distance from the primary vertex. Otherwise, if
no vertex information is available, an histogram whose resolution depends on the number
of seeds found is filled with the \( z \) impact parameter of each seed; its maximum is then taken
as \( z \) position for the primary vertex. This vertexing algorithm, which can be operated in
both RoI and full scan modes, is best suitable for high luminosity events containing many
high \( p_T \) tracks (e.g., b-tagging). Independent cuts on \( r - \phi \) and \( z \) impact parameters are
eventually applied to the reconstructed seeds to further reduce the fake fraction.

\texttt{STThreePointFit} extends track seeds with a third \texttt{SpacePoint}; it uses a Monte Carlo
map associating to each seed a set of module roads\footnote{A road is a list of modules ordered according to the radius at which they are placed starting from the innermost one} the track could have hit passing
through the Pixel or SCT detectors. A subset of modules is extracted from each road
according to a user defined parameter relating to their "depth" inside it (e.g., the user can
decide to use modules at the beginning or in the middle of each road, etc.). \texttt{SpacePoints}
from the selected modules are then used to extend the seed and candidate tracks are fitted
with a circle; ambiguities (e.g., tracks sharing at least one \texttt{SpacePoint}) can be solved on
the basis of the track quality, leading to an independent set of tracks that can be used for
trigger selection or as a seed for further extrapolation.

7.1.4 TRT-LUT

TRT-LUT is a LVL2 tracking algorithm for track reconstruction in the TRT. It is described
in detail elsewhere \cite{42}.

The algorithm takes as input Hits in the TRT. The algorithmic processing consists of
Initial Track Finding, Local Maximum Finding, Track Splitting, and Track Fitting and
Final Selection. It outputs the Hits used and Tracks with their parameters.

During the Initial Track Finding, every hit in a three-dimensional image of the TRT
detector is allowed to belong to a number of possible predefined tracks characterized by
different parameters. All such tracks are stores in a Look-Up Table (LUT). Every hit
increases the probability that a track is a genuine candidate by one unit.

The next step consists of Local Maximum Finding. A two-dimensional histogram is
filled with bins in \( \phi \) and \( 1/p_T \). A histogram for a single track would consists of a "bow-tie"
shaped region of bins with entries at a peak in the center of the region. The bin at the
peak of the histogram will, in an ideal case, contain all the hits from the Track. The
roads corresponding to other filled bins share straws with the peak bin, and thus contain
sub-sets of the hits from the track. A histogram for a more complex event would consist
of a superposition of entries from individual tracks. Hence, bins containing a complete
set of points from each track can be identified as local maxima in the histogram.
The Track Splitting stage of the algorithm analyzes the pattern of hits associated to a track candidate. By rejecting fake candidates composed of hits from several low-$p_T$ tracks, the track splitting step results in an overall reduction by a factor of roughly 2 in the number of track candidates. For roads containing a good track candidate, it identifies and rejects any additional hits from one or more other tracks. The result of the overall Track Splitting step is a candidate that consists of a sub-set of the straws within a road. 

The final step of TRT-LUT, Track Fitting and Final Selection, performs a fit in the $r - \phi$ ($z - \phi$) plane for the barrel (end-caps) using a third order polynomial to improve the measurement of $\phi$ and $p_T$. Only the straw position is used (i.e., the drift time information is not used). The track is assumed to come from the nominal origin. After the fit, a reconstructed $p_T$ threshold of 0.5 GeV/c is applied.

### 7.1.5 TRT-Kalman

The TRT-Kalman is a Level-2 track reconstruction algorithm which utilizes information of TRT part of Inner Detector only [43]. The core of algorithm is a set of utilities from ORP xKalman++ (see Section 7.2.1) for reconstruction of tracks in the TRT detector. It is based on a Hough-transform (histogramming) method.

At the initialisation step of the algorithm, a set of trajectories in the $\phi - r(z)$ space is calculated for the barrel and endcap parts of TRT. The real value of magnetic field is taken into account at each straw position coordinates when calculating trajectories. The lower bound of $p_T$ used is 0.5 GeV/c. Then a histogram with 500 bins in $\phi$ and 70 bins in curvature is filled with TRT drift circle positions. Each hit will populate many cells in the histogram, but for each track in thr RoI there will be a bin where all hits on the track have an entry. Thus the track candidates can be identified from peaks in the histogram. Bins with the number of contributing cluster greater than 7 are considered as a track candidates. The track candidate also must satisfy some quality criteria such as the number of unique hits and the ratio of hits to number of straws crossed by trajectory.

TRT-Kalman incorporates following modified modules from xKalman:

- **XK_Tracker_TRT**: This reads TRT geometry from from ROOT files. It uses InDetDescr, InDetIdentifier to access necessary Detector Description information;
- **XK_Algorithm**: A strategy is added to perform TRT standalone reconstruction;
- **XK_Track**: A step has been added with fine-tuning of track parameters after the histogramming step and Least Squares fit;
- **XKaTrtMan, XKaTRTRec**: This contains xKalman++ internal steering algorithms;
- **XKaTRTClusters**: This component retrieves TRT_RDO_Container from StoreGate filled from a ByteStream file.
7.1.6 T2Calo

T2Calo [44, 45, 46, 47] is a clustering algorithm for electromagnetic (EM) showers, seeded by the LVL1 EM trigger RoI positions [48]. This algorithm can select isolated EM objects from jets using the cluster $E_T$ and certain shower-shape quantities.

The RIOs are calibrated calorimeter cells (LArCells and TileCells), imported from the offline reconstruction. Both LArCells and TileCells have CaloCell as common base class. The output (T2EMCluster) is a specific LVL2 class containing the cluster energy and position, and the shower-shape variables useful for the selection of EM showers.

The first step in T2Calo is to refine the LVL1 position from the cell with highest energy in the second sampling of the EM calorimeter. This position $(\eta_1, \phi_1)$ is later refined in the second sampling by calculating the energy weighted position $(\eta_c, \phi_c)$ in a window of $3 \times 7$ cells (in $\eta \times \phi$) centered in $(\eta_1, \phi_1)$. In Ref. [45], the steps to perform the jet rejection are the following:

- In sampling 2, $R^\text{shape}_\eta = E_{3\times7}/E_{7\times7}$ is calculated. The expression $E_{n\times m}$ stands for the energy deposited in a window of $n \times m$ around $(\eta_1, \phi_1)$. This shape variable takes into account that most of the energy of EM showers is deposited in the second sampling of the EM calorimeter.

- In sampling 1, $R^\text{strip}_\eta = (E_{1\text{st}} - E_{2\text{nd}})/(E_{1\text{st}} + E_{2\text{nd}})$ is obtained in a window of $\Delta \eta \times \Delta \phi = 0.125 \times 0.2$ around $(\eta_c, \phi_c)$. $E_{1\text{st}}$ and $E_{2\text{nd}}$ are the energies of the two highest local maxima found, obtained in a strip-by-strip basis. The two $\phi$-bins are summed and only the scan in $\eta$ is considered. A local maximum is defined as a single strip with energy greater than its two adjacent strips.

- The total transverse energy $E_T$ deposited in the EM calorimeter is calculated in a window of $3 \times 7$ cells around $(\eta_1, \phi_1)$.

- Finally, the energy that leaks into the hadron calorimeter $E_T^{\text{had}}$ is calculated in a window of size $\Delta \eta \times \Delta \phi = 0.2 \times 0.2$ around $(\eta_c, \phi_c)$.

7.1.7 muFast

The muFast algorithm is a standalone LVL2 tracking algorithm for the Muon Spectrometer. In the past, it existed in the Reference software from ATRIG, and this version is described in detail elsewhere [49].

The program is steered by the RoI given by the LVL1 Muon Trigger and uses both RPCs and MDTs measurements. At present this algorithm is limited to the barrel region and it is based on four sequential steps:

- **LVL1 emulation**: the muon pattern recognition in the MDT system is initiated by the RPC hits that induced the LVL1 trigger accept. Among these hits, only
those related to the pivot plane (i.e., the middle RPC station) are provided by the muon trigger processor; the ones related to the coincidence plane (i.e., the innermost and outermost RPC stations) have to be identified running a fast algorithm that simulates the basic logic of the LVL1 selection.

- **Pattern recognition** is performed using the RPC hits that induced the LVL1 trigger to define a road in the MDT chambers around the muon trajectory. MDT tubes lying within the road are selected and a contiguity algorithm is applied to remove background hits not associated with the muon trajectory;

- A **Straight-line track fit** is made to the selected tubes (one per each tube monolayer) within each MDT station. For this procedure the drift-time measurements is used to fully exploit the high measurement accuracy of the muon tracking system. The track sagitta is then evaluated.

- A **fast $p_T$ estimate** is made using LUTs. The LUT encodes the linear relationship between the measured sagitta and the $Q/p_T$, as a function of $\eta$ and $\phi$. The output of this algorithm is the measurement of the muon transverse momentum $p_T$ at the main vertex, $\eta$, and $\phi$.

### 7.1.8 muComb

The combination of the features of the track measured in the Muon Spectrometer and the Inner Detector (ID) at LVL2 provides a rejection of pion and Kaon decays to muons and of fake muons induced by the cavern background. Moreover the combination of the two measurements improves the momentum resolution of reconstructed muons over a large momentum range.

The matching of the Muon Spectrometer tracks and of the ID can be performed extrapolating the ID track to the muon system. The procedure needs to take into account the detector geometry, the material composition and the inhomogeneity of the magnetic field. An accurate extrapolation would require the use of detailed geometry and magnetic field databases, together with a fine tracking. All this would is expensive in terms of CPU time and therefore not acceptable for the LVL2 trigger.

To provide a fast tracking procedure, the effects of the geometry, the materials and of the magnetic field have been described by simple analytic functions of $\eta$ and $\phi$. The extrapolation of the ID tracks to the entrance of the Muon Spectrometer is performed using linear extrapolation in two independent projections: the transverse and the longitudinal views. Two coordinates are extrapolated: the $z$-coordinate and the azimuthal angle $\phi$. The linear extrapolation is corrected using average corrections. In the transverse projection the ID track extrapolation in $\phi$ is corrected as

\[ \Delta \phi = \frac{\alpha}{p_T - p_T^0}; \quad (1) \]
where $\alpha$ is related to the field integral and allows for the transverse energy loss in the material of the calorimeter, that is approximately independent of the track transverse momentum. Both $\alpha$ and $p_T^0$ have been determined by fitting $\Delta \phi$ of simulated muons as a function of $p_T$. It is found that $p_T^0 \approx 1.5$ (i.e., about half of the transverse energy loss of low energy muons), as naively expected. A similar approach has been followed in the case of the extrapolation of the $z$-coordinate in the longitudinal view.

The matching is done geometrically using cuts on the residuals in each of $z$ and $\phi$.

For matching tracks the combined transverse muon momentum is estimated through a weighted average of the independent $p_T$ measurements in the Muon Spectrometer and in the Inner Detector. For each combined track, a $\chi^2$ parameter is used to evaluate the quality of the $p_T$ matching. Thanks to the high quality of the muon $p_T$ measurements in both detectors, secondary muons from pion and Kaon decays give typically bad $\chi^2$ matching, and thus can be rejected.

### 7.1.9 WuppertalVertexing

A secondary vertex finder to tag b-jets at LVL2 would enhance the physics potential of ATLAS. Of course, the challenge of this task is to develop a fast and efficient algorithm. The WuppertalVertexing algorithm [50] discussed in this subsection describes the tracks at the perigee point, the point of closest approach to the $z$-axis. In the perigee scheme the track parameters are derived in the vicinity of the estimated vertex position. Therefore, the track parameters at the vertex and at the perigee point are nearly the same. The tracks are described by the following five parameters: a signed impact parameter $d_0$ in the plane orthogonal to the beam ($z$) direction with respect to the beam position, the $z$ coordinate of the perigee point, the angles $\theta$ and $\phi$ of the tracks at this point, and the curvature which is signed by the charge.

This algorithm expects to receive as input Tracks from another LVL2 algorithm. The first step of the algorithm is the calculation of the track parameters from the expected vertex position and the momentum vector. Using the difference of the calculated and measured track parameters and the error matrix, a $\chi^2$ value is calculated. This $\chi^2$ is then minimized with respect to the estimated vertex position which is in general the origin in the first iteration step. The advantage of the perigee parametrization is that the tracks are described in the vicinity of the vertex. Therefore the momentum vectors can be considered constant. This will lead to a rather simplified mathematics, for example due to smaller matrices. If the $\chi^2$ is minimized with respect to the estimated vertex position, an equation is obtained for the vertex position of all tracks in the subsample of tracks.

In the second stepm the assumed vertex position is improved by rejecting tracks that are apparently not originating from this vertex. The mathematics of the vertex fitter allows this rejection to be fast since only a few calculations have to be redone. The algorithm will reject the track with the highest $\chi^2$ as long as the total $\chi^2$ probability is to low. If the $\chi^2$ probability is sufficient, a further iteration is performed with the fitted
vertex position being the new starting point. If the maximum number of iteration steps is achieved or the total $\chi^2$ is small and does not change anymore, the process is finished and a vertex is found.

This vertex fitter has fast system performance. On a system with two Pentium 3 processors with each 800 MHz an average time of 1 ms was achieved for fitting the primary vertex.

The algorithm is still under development, and further improvements to it are anticipated. For example, at the moment it searches only for a primary vertex. One further step will be the integration of a secondary vertex to perform b-tagging. In addition, system performance improvements to the algorithm are in progress. In this respect, the option should be explored to create a subset of tracks for the secondary vertex search without the knowledge of the primary vertex. This would permit a search for both vertices in parallel.

### 7.2 Event Filter Algorithms

It is intended that Event Filter algorithms are Offline algorithms adopted to the HLT environment.

#### 7.2.1 xKalman++

xKalman++ is a package for global pattern recognition and Track fitting in the Inner Detector for charged tracks with transverse momentum above 0.5 GeV/c. A more detailed description of this algorithm is available elsewhere [51].

The algorithm starts the track reconstruction in the TRT using a histogramming method or in the Pixel and SCT detector layers using segment search.

The first reconstruction method outputs a set of possible track candidate trajectories defined as an initial helix with a set of parameters and a covariance matrix. As a second step the helix is then used to define a track road through the precision layers, where all the measured clusters are collected. xKalman++ attempts to find all possible helix trajectories within the initial road and with a number of sufficient clusters.

The primary track finding in the Pixels or SCT outputs a set of SpacePoints as an initial trajectory estimation. In the next step these set of space points serve as an input for the Kalman filter-smoother formalism that will add the information from the precision layers. Each reconstructed track is then extrapolated back into the TRT, where a narrow road can be defined around the extrapolation result. All TRT Clusters together with the drift time hits found within this road are then included for the final track-finding and track-fitting steps.

There are three seeding mechanism available in the offline environment: **XKaSeedsAll**, the reconstruction of the full event; **XKaSeedKINE** reconstruction of a region of interest
and soon available EM calorimeter seeding. In the HLT environment as an EF algorithm xKalman++ will be seeded by the LVL2 result.

After the pattern recognition and Track fitting steps xKalman++ stores the final Track candidates as SimpleTrack objects in a SimpleTrackCollection. The Track candidate contains the following information:

- Fit procedure used (m-fit or e-fit);
- Helix parameters and their covariance matrix at the end-points of the filter procedure in the precision layers (point on the trajectory closest to the vertex) and in the TRT (point on the trajectory closest to calorimeter);
- Total $\chi^2$ resulting from final fit procedure;
- List of all hits on track from all sub detectors;
- Total number of precision hits $N_p$.
- Total number of straw hits $N_s$, empty straws crossed $N_e$, and of drift-time hits $N_t$.

Furthermore, a track candidate is stored in the final output bank if it passes the following cuts:

- The number of precision hits is larger than 5 to 7;
- The ratio $N_s/(N_s + N_e)$ is larger than 0.7 to 0.8;
- The ratio $N_t/N_s$ is larger than 0.5 to 0.7;
- No previously accepted track has the same set of hits as the current one; this last cut removes full ghost tracks.

### 7.2.2 iPatRec

iPatRec [52] is a pattern recognition algorithm used in the Event Filter that searches for tracks initiated from SpacePoint combinations. Pixel and SCT SpacePoints are used to form track-candidates. Candidates are extrapolated to the TRT and drift-time hits added.

In the initialization phase, iPatRec creates a geometry database describing the properties of each detector module in the precision tracker plus the module’s relationship to a simplified material model. This model comprises material layers assumed to be either concentric cylinders in the barrel region or planes normal to the beam-axis in the end-cap regions. Additional layers represent the TRT detector, beam-pipe and inert support/service material. The track finding, following and fitting procedures make extensive
use of this database. Another initialization task is to parametrize the magnetic field to enable a fast propagation of track parameters between layers.

In the first step of event reconstruction, adjacent raw-data channels are clustered, and SpacePoints produced from these clusters. Each SpacePoint is assigned to one of 7 partitions according to distance from the intersection region. Within each partition the points are ordered according to their azimuthal coordinate.

The general procedure is to form track-candidates using SpacePoint combinations from three different partitions subject to criteria on maximum curvature and crude vertex region projectivity. Candidates then undergo a track-fit procedure to give track parameters with covariance at the point of closest approach to the beam-line given in terms of perigee parameters. The track follower algorithm propagates these parameters to form an intersect with error ellipse at each layer in turn. Clusters are associated to the track from the traversed detectors. Final cluster allocation decisions are taken after a further track-fit. During this fit, energy loss and Coulomb scattering are taken into account by allocating a scattering centre (with associated extra fit parameters) to each layer traversed. An active detector region traversed without associated cluster is classified as a hole and retained for material and quality information. Tracks with fit probability greater than 0.001 and a maximum of three holes are extrapolated to the TRT, where a histogramming technique is used to select the TRT hits to be added to the track. Tight cuts are made on the straw residual and on the ratio of found to expected straws, in order to limit high luminosity occupancy effects.

Tracks with cluster(s) in the two innermost Pixel layers plus TRT association are termed primary tracks. Otherwise a maximum of only one hole is allowed: truncated tracks start in the innermost layers but cannot be followed to the outermost layers or TRT; secondary tracks start further out and are required to have TRT association. Various partition combinations are taken to maintain track-finding efficiency for the three types of track even in the event of a higher than expected detector inefficiency. To avoid track duplication, only candidates with two unallocated SpacePoints are initiated, and tracks sharing more than 50% of their clusters are deemed ambiguous whence only the one with higher quality is retained.

To speed up the execution, a preliminary track-finding pass looks only for high quality primary tracks and finishes as soon as one is found. The vertex from this track is then used to subdivide the SpacePoint partitions into projective slices - greatly reducing the combinatorial load. Impact parameter criteria are adjusted according to the distance to the first cluster to ensure there is no bias against b-, c- or s-particle decays. The code iterates to allow for several initial vertices, very necessary at high luminosity, and reverts to a slower algorithm when no high quality tracks are found.

A special fit procedure is available to better handle electron bremsstrahlung. This is invoked from the subsequent combined reconstruction for tracks associated to an EM-calorimeter cluster.
7.2.3 LArClusterRec

LArClusterRec is the reconstruction package for electromagnetic clusters in the calorimeter. It is generally used in conjunction with TileRec and referred to collectively as CaloRec.

In the first step towers are created by summing the cells of the electromagnetic calorimeter and the pre-sampler in depth using a granularity of $\Delta\eta \times \Delta\phi = 0.025 \times 0.025$. The input of the tower building are the calibrated calorimeter cells which are produced by the package LArCellRec or TileRec.

In the next step a sliding window algorithm is used. In case a local maximum is found with a total energy in the window above a given transverse energy threshold, clusters are created which are subsequently stored in the cluster container. To reconstruct the cluster energy and position is calculated in a given window.\footnote{This window can be different from the one used for the sliding window algorithm.} The cluster energy is corrected for $\eta$ and $\phi$ modulations and leakage outside the cluster in a given window. In the region between the barrel and end-cap calorimeter the cluster energy is in addition corrected for energy losses using the energy deposit in the crack scintillators. The $\eta$ position in the first and second sampling is corrected for s-shapes, which is a geometrical effect. The $\phi$ position is corrected for an offset, which is also a geometry effect.

7.2.4 egammaRec

egammaRec is designed to calculate useful quantities to separate clusters in the electromagnetic calorimeter from jets. To do so, electromagnetic cluster information as well as tracking information is used.

In the electromagnetic calorimeter electrons are narrow objects, while jets tend to have a broader profile. Hence, shower shapes can be used to reject jets. This is handled by the EMShowerBuilder which calls different algorithms which calculate diverse quantities using the information in the first and second sampling of the electromagnetic calorimeter as well as the leakage into the first sampling of the hadronic calorimeter.

Cluster and track information is combined in the TrackMatchBuilder. For a given cluster the nearest track is searched for by correctly taking into account the bending of tracks in the magnetic field. In case a track is found in a certain distance from the cluster and the $E/p$ ratio fulfills $0.5 < E/p < 1.5$ the trackmatch is successful. In the final $e$/jet separation step, jets can be rejecting by applying harder $E/p$ cuts as well as a cut in $\Delta\eta$ and $\Delta\phi$ between the cluster and the track. The next final step after egammaRec has run is the final particle identification step. A first version of this package is available, but it is not yet tested.
7.2.5 Moore

Moore (Muon Object Oriented Reconstruction) is a track reconstruction package for the Muon Spectrometer. A detailed description of Moore is available elsewhere [53].

Moore takes as input collections of digits or clusters inside the Muon Spectrometer (CSC, MDT, RPC, TGC) and outputs fitted reconstructed tracks whose parameters are expressed at the entrance of the muon spectrometer.

The reconstruction is performed in several steps and each step is driven by an Algorithm module, MooMakeXXX. Each algorithm is independent (i.e., it retrieves objects created by the previous modules from StoreGate and it builds a transient object to be recorded in StoreGate where it is available for the subsequent algorithms). The only link between algorithms are the transient objects, in such a way that the algorithms depend on transient objects but transient objects do not depend on algorithms. The decoupling between data and algorithms and the natural step sequence of algorithm performing the reconstruction gives the opportunity to plug-in different reconstruction algorithms at run time.

As it is now, the overall reconstruction starts from the searches for φ regions of activity and builds PhiSegments (MooMakePhiSegments). For each φ-Segment, the associated MDTs are found and a crude RZSegment is built (this is essentially a collection of z hits) (MooMakeRZSegments).

Inside the MDTs the drift distance is calculated from the drift time, by applying various corrections: such as the TOF, the second coordinate, the propagation along the wire, the Lorenz effect. From the 4 tangential lines the best one is found. All the MDT segments of the outer station are combined with those of the Middle layer. The MDT hits of each combination are added to the phi-hits of the φ Segment, forming outer track candidates. All the successfully fitted candidates are kept for further processing (MooMakeRoads).

The successful outer track is subsequently used to associate inner station MDT hits. A final track is defined as a successfully fitted collection of trigger hits and MDT hits from at least two layers (MooMakeTracks). The parameters of the fitted track are referred to the first measured point and are therefore expressed at the entrance of the Muon Spectrometer.

When dealing with data already selected by the trigger the first two steps (MooMakePhiSegments) and (MooMakeRZSegments) can be substitute with ad hoc makers that seed the track search in the regions selected by the trigger.
Appendix: Definition of Terms

This section defines terms relevant to HLT Algorithms.

- **Cluster**: A concept which exists for the Pixel and SCT detectors.
  - *Pixel Detector*: A two-dimensional group of neighbouring readout channels in a DE.
  - *SCT*: A one-dimensional group of neighbouring readout channels in a DE.

- **Coordinate System**:
  - *Global Coordinate System*: A right-handed Cartesian coordinate system where the origin is centered around the nominal interaction point (i.e., the mean beam collision point) with the $z$-component parallel to the axial component of the magnetic field produced by the solenoid.
  - *Local Coordinate System*: A DE-dependent coordinate system related to the Global Coordinate System by an Euler transformation.

- **Detector Element (DE)**:
  - *Pixel Detector*: a module, equivalent to a single Silicon wafer [19]. There are 1744 Pixel Detector Elements.
  - *SCT*: One side of a module, equivalent to a bonded pair of wafers whose strips are oriented in a single direction (i.e., axial or stereo) [19]. There are 8176 SCT Detector Elements.
  - *TRT*: a planar set of straw tubes representing one row at a given radius of straws in a barrel module (i.e., a plane corresponding to the tangential direction in the barrel) and $1/32$ in $r\phi$ at a given $z$ of straws in an end-cap wheel [19]. There are 19008 TRT Detector Elements.
  - *Liquid Argon*: The concept of Detector Element does not exist in the Liquid Argon calorimeter, but it should, and might correspond to Level 1 Trigger Towers.
  - *Tile Calorimeter*:
    - *MDT*: a single MDT chamber, where there is at most a single MDT chamber per station. And typically, an MDT chamber has two multilayers [54].
    - *RPC*: the RPC components associated to exactly one barrel muon station. There may be 0, 1 or 2 RPC doublet sets per station and a doublet set may comprise 1, 2 or 4 RPC doublets [54].
- **TGC**: one TGC eta division, or chamber, in a TGC station. There are 24 forward stations in a ring and 48 endcap stations in a ring and there are four rings at each end of the ATLAS detector [54].

- **CSC**: a single CSC chamber, where there is at most a single CSC chamber per station. And typically, a CSC chamber has two multilayers [54].

- **IdentifiableContainer**: A template container class inheriting from the `DataLinkVector` class for any identifiable class (e.g., a Collection class) specified by the user. Instances are stored in a Hash Table. See [7, 55].

- **Raw Data**: Read-Out Buffer (ROB)-formatted data produced by the ATLAS detector or its simulation.

- **Raw Data Object (RDO)**: Uncalibrated Raw Data converted into an object representing a set of readout channels. Historically this has been referred to as a *Digit*. It is the representation of Raw Data which is put into the Transient Event Store and is potentially persistifiable.

- **Space Point (SP)**:
  - **Pixels**: A single Cluster in the Global Coordinate System.
  - **SCT**: A combination of one axial Cluster and one stereo Cluster.
Acknowledgements

References


See the code contained in offline/MuonSpectrometer/MuonDigitContainer/RpcDigit.h

A. Schaffer, in the DetectorDescription/Identifier package in the ATLAS CVS repository.


E. Frank and A. Gupta, /TileCalorimeter/TileEvent/TileCell.h.


See within the ATLAS CVS repository Trigger/TrigEvent/TrigInDetEvent for the TrigInDetTrack class.


For more information on Singleton and other Design Patterns, refer to E. Gamma, R. Helm, R Johnson, and J. Vlissides, Design Patterns, Addison-Wesley Professional Computing Series (1995).

There are numerous publications describing TAU available via: http://www.cs.uoregon.edu/research/paracomp/tau/papers.html.

http://www.cs.uoregon.edu/research/paracomp/tau/tautools/


[34] http://hepunx.rl.ac.uk/atlasuk/simulation/level2/doc/Prototypes/TrigSteerExample1/TimerSvc.html


[39] M. Müller, N. Konstantinidis et al. (UCL Group), IDSCAN, presentation given in PESA Core Algorithms Group meeting, 3 December 2002;
H. Drevermann and N. Konstantinidis, “Determination of the z position of primary interactions in ATLAS,” ATL-DAQ-2002-014;

[40] For more information on SCTKalman see
I. Gaines, T. Huehn, and S. Qian, in Proceedings of CHEP97 (Berlin);
I. Gaines and S. Qian, in Proceedings of CHEP98 (Chicago);
I. Gaines, S. Gonzalez and S. Qian, in Proceedings of CHEP2000 (Padova);
D. Candlin, R. Candlin and S. Qian, in Proceedings of CHEP01 (Beijing);


[54] See the ATLAS CVS repository under /MuonSpectrometer/MuonDetDescr.