DIPLOMARBEIT

„System Integration of the Global Trigger for the CMS Experiment at CERN“

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Diplomingenieurs der technischen Physik unter der Leitung

von

Univ.Doz. Dipl.-Ing. Dr.techn. Claudia-Elisabeth Wulz

Atominstitut der Österreichischen Universitäten
Institutsnummer E141

in Zusammenarbeit mit

DDipl. Ing. Ildefons Magrans de Abril

Institut für Hochenergiephysik der Österreichischen Akademie der Wissenschaften

eingereicht an der Technischen Universität Wien
Technisch-Naturwissenschaftliche Fakultät

von

Philipp Glaser
Matr. Nr.: 9926600
Lerchenfeldergürtel 16/4
1070 Wien

Wien, am 20. März 2007

Acknowledgements

First of all I would like to thank Dr. Claudia-Elisabeth Wulz, who roused my interest in High Energy Physics and offered me the great possibility to work two months at CERN in a project in autumn 2004. I enjoyed working in the scientific environment at CERN in the CMS Global Trigger group. It was her again who encouraged me to come to CERN for a longer period to do my thesis, starting in October 2005, a decision I will never regret. I want to express my gratitude to her with these words.

The next person that I would like to mention is Tobias. Without him I would not have been courageous enough to undertake the step of moving to live in Geneva and work at CERN. He was a dear friend and colleague during good and bad times. I always could count on his great knowledge and experience with computers in general and also on his talent in explaining complex problems with great patience.

The successful completion of this work is due to the guidance and support from my supervisor Ildefons Magrans. He was the one who brought organization into my work and taught me to work independently.

Thanks goes also to the whole CMS Global Trigger group for making me feel welcome in their midst. From the first moment on I felt recognized and my opinion was always taken seriously. Toni and Barbara never got tired of explaining me aspects of the complex Global Trigger system. Manfred and Ivan had always an open ear for my problems and where not only important partners in fruitful discussions but also good company during lunch and coffee breaks.

Finally I would express my gratitude to my family. Without my parents, grandparents and brothers I would not be the person I am.

Abstract

This document describes how existing control software for the Global Trigger system of the CMS experiment at CERN was adapted and integrated in the infrastructure of the CMS and Level-1 Trigger control software. Two frameworks building the support structure are outlined and libraries providing high level access to the Global Trigger hardware are described. The main focus of this document is the description of how fine grained control of the Global Trigger hardware is achieved and how a consistent, flexible and extensible way configuration for the system making use of access to a database was implemented. The layout of the database schema designed to comply with the requirements of the Global Trigger is part of the present thesis too. The document gives a selective and brief introduction for future developers continuing the work on this software. It could also serve as a pool of ideas for people working on control software for other Level-1 Trigger sub-systems.

Contents

List of Figures

Chapter 1

Introduction

1.1 The Large Hadron Collider (LHC) at CERN

The Large Hadron Collider (LHC) [1] experiment is currently under construction at CERN, the European Organization of Nuclear Research in Geneva. The machine will provide proton-proton collisions with a center of mass energy of 14 TeV and a luminosity of $10^{34} cm^{-2} s^{-1}$. Also heavy (Pb) ion collisions with a center of mass energy of more than 1000 TeV will be provided. The collider is contained in a 26.7 km circumference tunnel located underground at a depth ranging from 50 to 150 meters. The tunnel was formerly used for the Large Electron Positron collider (LEP).

1.1.1 Motivation for the LHC

All evidence indicates that new physics, and answers to some of the most profound questions of our time lie at energies around 1 TeV. The results from the LHC might shed light on:

The origin of particle masses: In the Standard Model (SM) the origin of particle masses is explained by the *"Higgs Mechanism"*. This mechanism predicts a scalar boson that has not been found yet. The mass of this particle, the so called *"Higgs particle"*, is a free parameter in the SM. Although the particle has not yet been found in any high energy physics experiment, it is possible to derive a lower limit of its mass of about 115 GeV/c^2. If this particle exists with a mass below 1 TeV/c^2 it will be detected at the LHC [2].

Supersymmetry: There are theories beyond the SM that propose that the SM is just a low-energy approximation of a more fundamental theory. One of these is *Supersymmetry* or SUSY. According to this theory, for every known particle there should be a corresponding supersymmetric partner. Some of the new particles are expected to have masses well below the TeV scale. The LHC will therefore be capable of exploring this energy range and give insight to the question if SUSY is the right extension to the SM.

New states of matter: The theory of Quantum Chromodynamics (QCD) predicts a state of matter called "Quark-Gluon plasma", which is a deconfined state of quarks and gluons, occurring at extremely high energies, and which was present fractions of a second after the "Big Bang". The observation of this state of matter may also be possible in the heavy ion collisions provided by the LHC.

Moreover the LHC will definitely provide further understanding of the physics of known particles. Due to the high luminosity of the collider, heavy SM particles like the b and t quarks will be produced in large quantities, and allow precision measurements of parameters that are still not precisely known.

1.1.2 Detectors at the LHC

There are four collision points spread over the LHC ring 1.1, which house the main LHC experiments. While CMS is described in the subsequent section the other three detectors will be outlined here.

A Large Ion Collider Experiment (ALICE): In this experiment physics of strongly interacting matter at extreme energies and densities will be studied [3]. It is expected to produce a new phase of matter, the state in that our universe was 20

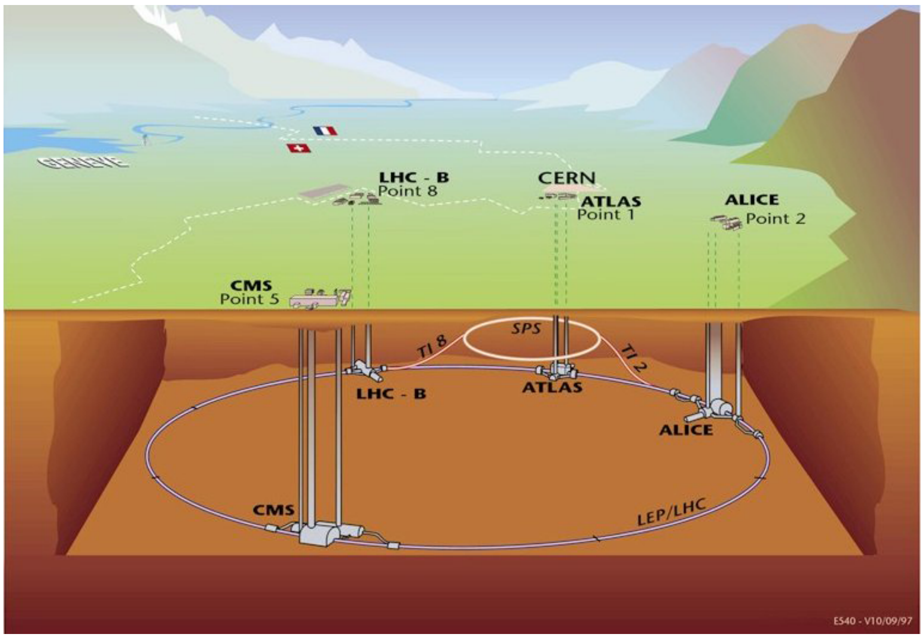


Figure 1.1: The LHC tunnel and its four main experiments.

to 30 μs after the big bang - the quark gluon plasma (QGP). Collisions of lead ions should provide the required energy for the production of this state of matter. The main detector of ALICE is a Time Projection Chamber (TPC) with a diameter of about 5 m that allows particle identification despite enormous particle multiplicities.

A Toroidal LHC Aparatus (ATLAS): Like CMS, ATLAS [4] is a general-purpose experiment at the LHC. Installed at interaction point 1, close to the CERN Meyrin site, it is the largest of the four main detectors with a length of 46 m and a diameter of 25 m. CMS and ATLAS are designed as complementary detectors, which both have the capability to detect a Higgs boson over a large mass range, using different decay channels.

The Large Hadron Collider beauty experiment (LHCb): As the name indicates, the LHCb [5] detector is dedicated to the study of B mesons and b-quark physics with the aim of precision measurements of CP violation and studies of rare B-decays. The detector will be able to exploit its full potential during the low luminosity period of the LHC. During high luminosity runs the beams have to be de-focused due to the large number of pile-up events.

1.2 The Compact Muon Solenoid (CMS) Experiment

1.2.1 Introduction

CMS [6] is a general-purpose experiment that is installed at interaction point 5 at the LHC. The concept of the detector was based on the requirements of having a very good muon system whilst keeping the detector dimensions compact. Only a strong magnetic field would guarantee good momentum resolution for high momentum muons. Studies showed that a magnetic flux density of 4 T could be generated by a superconducting solenoid. The experiment concept based on the above requirements was presented in 1990. It is basically unchanged since that time. The name given to the experiment reflects the original design ideas: Compact Muon Solenoid (CMS).

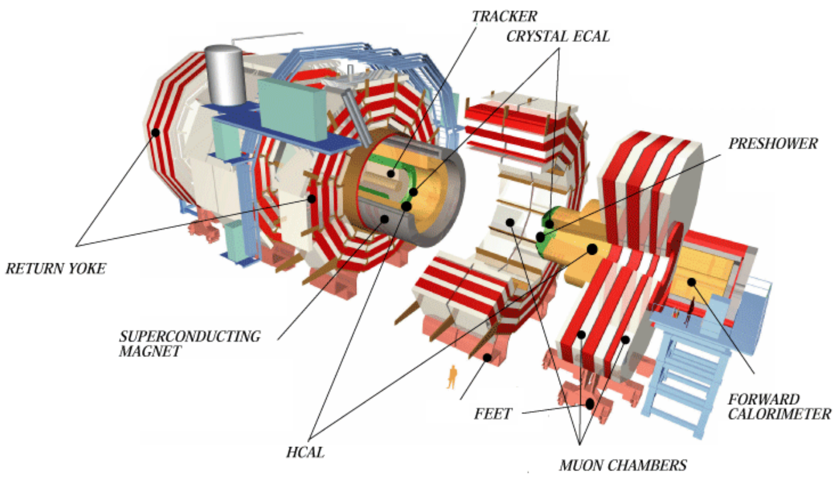


Figure 1.2: The layout of the CMS detector.

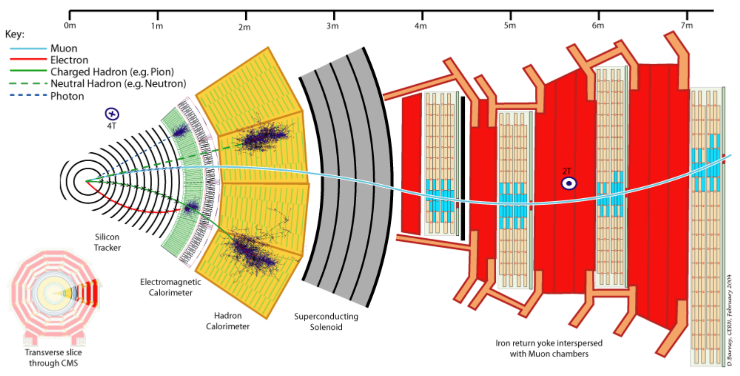

0m
1m
2m
3m
4m
5m
6m
7m
Key:
Muon
Electron
Charged Hadron (e.g. Pion)
Neutral Hadron (e.g. Neutron)
Photon
4T
2T
Silicon Tracker
Electromagnetic Calorimeter
Hadron Calorimeter
Superconducting Solenoid
Iron return yoke interspersed with Muon chambers
Transverse slice through CMS
D.Barney, CERN, February 2004

Figure 1.3: A slice of the CMS detector with trajectories of different types of particles.

Figure 1.2 shows a schematic drawing of the CMS detector and its components that will be described in detail in the subsequent section. Figure 1.3 shows a transverse slice of the detector. Trajectories of different kinds of particles and the traces they leave in the different components of the detector are also shown.

1.2.2 Tracker

The Tracker [7,8] is the innermost detector of the CMS experiment and consists of two different types of silicon detectors. Close to the beam pipe silicon pixel detectors are used to cope with the high track density. The subsequent layers of the tracker consist of silicon microstrip detectors.

Pixel Tracker

Figure 1.4 shows the layout of the Pixel Tracker. 3 layers at a distance of 4.3 cm, 7.2 cm and 11 cm from the beam pipe will be placed in the barrel region. During the initial period of LHC operation, the collider will run with lower luminosity, therefore only the 2 innermost layers will be installed. During high luminosity running the outer two layers will be used due to the enormous density of charged particles close to the beam pipe. Two disks of pixel detectors are located at each of the endcaps of the tracker to allow to reconstruct tracks of particles in the forward region.
The pixels of the Pixel Tracker are of rectangular shape (150 μm x 100 μm) with a sensitive thickness of around 250 μm. The spatial resolution that can be achieved with this pixel size, using interpolation algorithms, lies around 15 μm. With this resolution the task of reconstructing high momentum muons, electrons and hadrons with high accuracy is facilitated, and the precise determination of secondary vertices is possible.

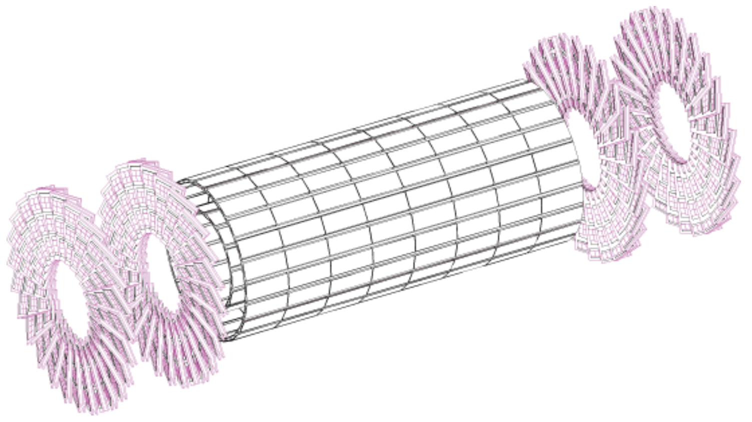


Figure 1.4: The layout of the CMS Pixel Tracker.

Silicon Microstrip Tracker

The rest of the Tracker volume with a total length of 5.40 m and a radius of 1.10 m is equipped with silicon microstrip detectors. 10 layers in the barrel region and 9 disks at each endcap result in a minimum of 12 measurement points per charged track over a wide range of pseudorapidity.

1.2.3 Calorimeters

Electromagnetic Calorimeter

The task of the Electromagnetic Calorimeter (ECAL) [9] is the measurement of energies and directions of electrons and photons. The reconstruction of Higgs particles decaying into two photons imposes the strictest performance requirement on the CMS ECAL.
To fit into the magnet the layout chosen for the ECAL is very compact utilizing dense lead tungstate crystals with a short radiation length as scintillation material. Over 80000 of those crystals with a total weight of 92.6 t are installed in the experiment taking up a volume of 11.18 m^3.

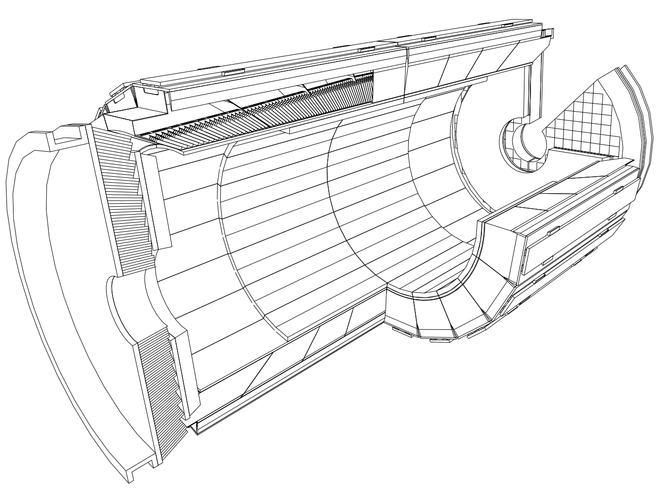


Figure 1.5: The Electromagnetic Calorimeter of the CMS experiment.

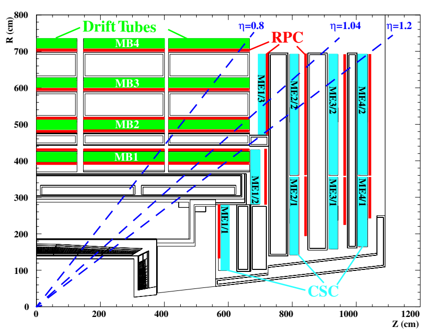
Hardronic Calorimeter

The Hadronic Calorimeter (HCAL) [10] measures the energy and direction of hadronic showers and together with the ECAL the missing and total transverse energies. Furthermore, information from HCAL helps to identify electrons, photons and muons. The HCAL consists of a barrel (HB) and encaps (HE) inside the magnet and an Outer Hadronic Calorimeter (HO) outside the coil. To achieve full hermeticity another part of the HCAL is located in the forward regions (HF) outside the muon system. HB, HE and HO are sampling calorimeters with copper as absorber and plastic scintillator tiles as active material. The HF uses quartz fibers as the active element embedded in a copper absorber matrix.

1.2.4 Muon System

Introduction

A redundant and precise muon system was one of the first requirements of CMS. The ability to trigger on and reconstruct muons, being an unmistakable signature for a large number of new physics processes CMS is designed to explore, is central to the concept of CMS.



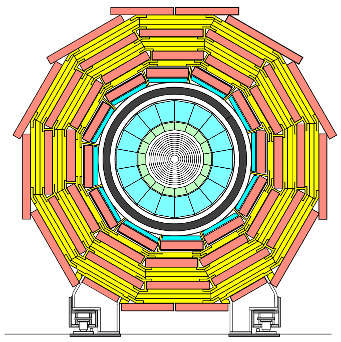


Figure 1.6: Longitudinal and Transverse views of the Muon System of CMS.

The muon system of CMS [12] is embedded in the iron return yoke of the magnet (1.2). It makes use of the bending of muons in the magnetic field for transverse momentum (p_T) measurements of muon tracks identified in association with the tracker. The large thickness of absorber material in the return yoke helps to filter out hadrons, so that muons are practically the only particles apart from neutrinos able to escape from the calorimeter system. The muon system consists of 4 stations of muon chambers in the barrel region and disks in the forward region 1.6, employing three different technologies of muon detectors that will be outlined shortly in the next sections.

Drift Tube Chambers

Drift Tubes (DTs) have been chosen in the barrel region because of the large area that has to be covered and the unproblematic radiation environment and almost vanishing magnetic flux [13]. The DTs consist of drift cells 1.7 filled with a gas mixture of 80 $\%$ Ar and 20 $\%$ $C0_2$, achieving a resolution of approximately 180 μm. Four staggered layers of DT cells are combined into a super layer. One DT chamber is made of three of such super layers. The two outer layers measure the coordinate in the bending plane (ϕ), the middle layer measures the coordinate perpendicular to the bending plane (z). Four layers of chambers are mounted in cocentric cylinders around the beam axis in five wheels of 2.5 m length, yielding a total of 250 chambers in the barrel.

Cathode Strip Chambers (CSCs)

CSCs have been chosen in the endcaps since they are capable of providing precise space and time information in the environment of a high magnetic field and high particle rate. CSCs are multiwire proportional chambers, in which one cathode plane is segmented into strips and anode wires run perpendicular to them. They are filled with a gas mixture of 30 $\%$ Ar, 50 $\%$ CO_2, and 20 $\%$ CF_4, providing a spatial resolution of 50 μm in ϕ direction. The closely spaced wires make the CSCs fast and therefore suitable for triggering. The CSC chambers are arranged in four disks perpendicular to the beam axis with iron disks of the magnet return yoke between them.

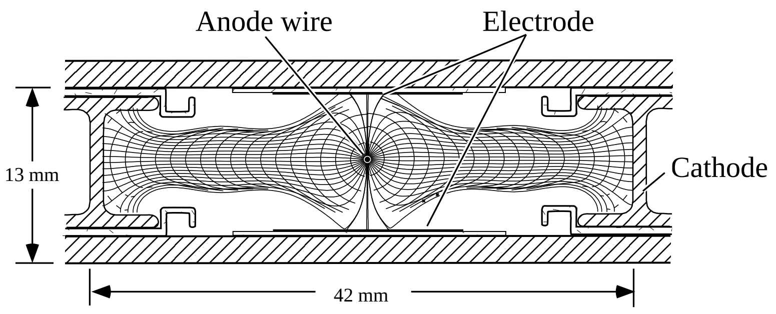


Figure 1.7: Schematic view of a Drift Tube Cell.

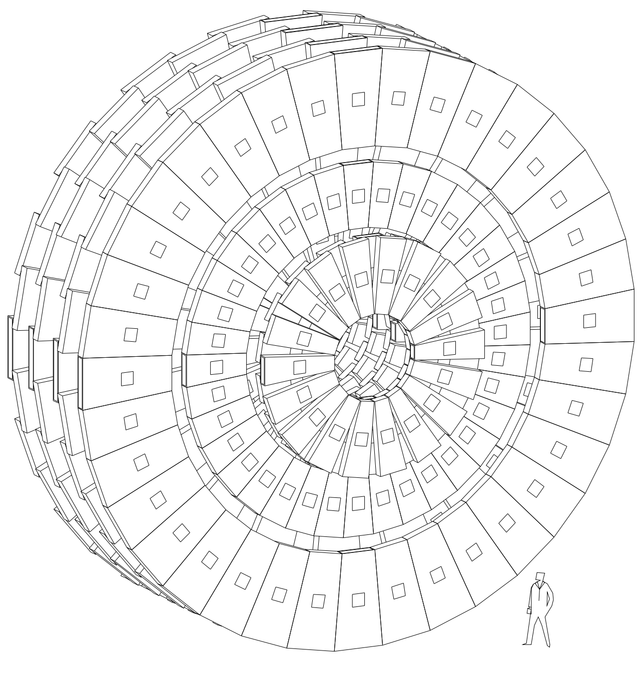


Figure 1.8: The CSC endcap system.

Resistive Plate Chambers (RPCs)

RPCs are used in both the barrel and endcap regions. Because of their fast response time, comparable to scintillators, they serve trigger purposes. By being sufficiently highly segmented the RPCs can also deliver spatial information, thus allowing to identify tracks and provide momentum information.

As shown in 1.6 RPCs are mounted on the DT chambers. In the inner two layers RPCs are mounted at each side of the DTs, in the outer layers only outside the DTs, giving a total of 6 layers. In the endcaps 4 layers are attached to the 4 CSC disks.

1.3 The Level-1 Trigger of the CMS Experiment

At the nominal LHC design luminosity the rate of interactions will be close to 10^9 per second, which is far more than the online computer farm could archive. Therefore the rate has to be reduced by a factor of 10^7 to 100 Hz. In CMS this reduction is achieved in two steps. The L1 Trigger, a custom-built electronic system, is designed to reduce the rate to 100 kHz using only coarsely segmented trigger data. In case of a Level-1 Accept (L1A), referring to an event that is considered ”interesting” by the L1 Trigger [14], high resolution data stored in pipeline memories are read out and passed to the High Level Trigger (HLT) [15], an online computer farm built of commercial processors, where further selection is done in software.

The L1 Trigger decision has to be taken after a fixed latency of 3.2 μs. This limitation is determined by the size of the analog pipeline memories used by certain subsystems. Due to the physical size of the CMS detector and the underground caverns signal propagation time represents already a significant fraction of the overall latency. About one microsecond of the latency is left to the whole L1 trigger system to generate a L1 Accept or reject the event. As a consequence only simple calculations can be processed in the L1 stage of triggering, still, unlike other L1 trigger systems that rely just on counting objects with energy or momenta above defined thresholds, the L1 Trigger of CMS is capable of applying sophisticated topological trigger algorithms.

Figure 1.9 illustrates the treelike hierarchy of the Level-1 Trigger. First so called trigger primitives are generated by the calorimeter systems that are the Electromagnetic Calorimeter (Ecal) [9], the Hadronic Calorimeter (Hcal) [10] and the two Hadronic Forward Calorimeters (HF) located at each end of the CMS detector and by the muon systems [12] that are the Drift Tubes (DT), the Cathode Strip Chambers (CSC) and the Resistive Plate Chambers (RPC).

1.3.1 Calorimeter System

In case of the calorimeter system 1.9 the trigger primitives refer to energy deposits in the trigger towers of the electromagnetic and the hadronic calorimeter. The Regional Calorimeter Trigger (RCT) then tries to find candidates for electrons or photons, jets, isolated hadrons and calculates energy sums in different detector regions. These objects are forwarded to the Global Calorimeter Trigger (GCT) where the electrons or photons, the $\tau' s$ and jets are sorted according to energy and quality. The best four objects of each category are sent to the Global Trigger (GT). Moreover, the total and missing transverse energies are calculated as well as twelve numbers representing jet multiplicities for twelve different transverse energy thresholds [24] and also sent to the Global Trigger.

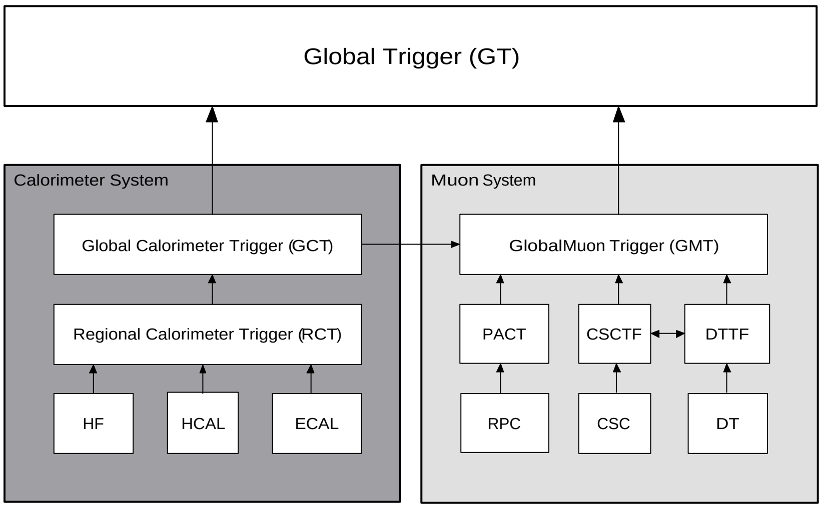


Figure 1.9: Hierarchy of the Level-1 Trigger. Acronyms are explained in the text.

1.3.2 Muon System

The muon system 1.9 of CMS consists of dedicated trigger chambers (RPCs) in addition to the tracking chambers (DTs and CSCs). Each of the components has its own trigger logic. The resistive plate trigger chambers are connected to a Pattern Comparator Trigger (PACT) which combines hits to tracks that are sent to the Global Muon Trigger (GMT). The Cathode Strip Chambers (CSCs) form Local Charged Tracks (LCTs) [14]. The best three LCTs per chamber are transmitted to the CSC Regional Muon Trigger [12]. The barrel DT chambers consist of three superlayers, each composed of four staggered layers of drift cells. The inner and outer superlayers have wires parallel to the beam direction to measure the azimuthal coordinate ϕ in the magnetic bending plane. The central superlayer has wires orthogonal to the beam line and measures the track position along the beam, in units of pseudorapidity η. With this information track segments are formed by the Bunch and Track Identifier (BTI) for each superlayer. For each chamber a Track Correlator (TRACO) [16] attempts to combine segments from the inner and the outer superlayer and forwards them together with segments found in the central superlayer to the Trigger Server. Only the best two track segments per chamber are forwarded to the DT Regional Muon Trigger.

The Regional Muon Trigger or Track Finder is made of two separate parts running in parallel. The Drift Tube Track Finder (DTTF) [17–20] and the Cathode Strip Chamber Track Finder (CSCTF) [21] reconstruct muons out of track segments found in the corresponding muon system and sort them according to quality and transverse momentum. In the region where the CSCs and the DTs overlap the two Regional Muon systems exchange information. The best four muons of each Track Finder system are sent to the Global Muon Trigger together with the best four barrel and best four forward muons from the Pattern Comparator Trigger of the Resistive Plate Chambers. The GMT converts track parameters from all subsystems to the same η, ϕ and p_T scales, validates the muon sign and attempts to correlate the tracks. Furthermore, the GMT receives isolation information and an indication whether the energy deposit is compatible with that of a Minimum Ionizing Particle (MIP) from the Global Calorimeter Trigger.

The final ensemble of muons is sorted based on the muons' quality, correlation and p_T. The top four muons are sent to the Global Trigger.

1.3.3 The Global Trigger System

The Global Trigger [24, 25](GT) is the final stage of the Level-1 Trigger. For physics data, decisions taken by the GT are based on trigger objects delivered by the GCT and the GMT. These are:

- Four muons
- Four isolated electron/photon objects
- Four non-isolated electron/photon objects
- Four central jet objects
- Four forward jet objects
- Four τ-flagged jet objects
- Missing transverse energy
- Total transverse energy
- Twelve numbers of jets above different thresholds

The logic of the Global Trigger can be programmed to apply selection criteria, so called trigger algorithms, on these objects. An algorithm is defined as a logic combination of the trigger objects together with a set of energy or momentum thresholds, windows in η and /or ϕ and topological conditions. The Global Trigger Logic module (GTL) can calculate up to 128 of such algorithms in parallel. Another module, the Final Decision Logic (FDL), then decides to accept an event for further processing by the HLT and eventual storage by the Data Acquisition (DAQ) or to reject it immediately.

The GT is a complex electronic system consisting of several VME modules mounted in a VME9U crate together with the GMT and the central Trigger Control System (TCS). Figure 1.10 shows the simplified layout of the GT crate.

Pipeline Synchronizing Buffers (PSB): The PSB modules receive and synchronize data coming from the GCT and send them over the back-plane to the dedicated modules. There are three PSBs receiving MIP and isolation information for the GMT and three receiving calorimeter objects sent to the GTL. One PSB board receives so called Technical Trigger (TT) signals that are directly sent to the Final Decision Logic (FDL) without further processing.

Timing Module (TIM): The TIM receives control and timing signals like the 40 Mhz LHC-clock and the Bunch Crossing 0 (BC0) signals and propagates it to all other GT modules over the back-plane (1.10).

Global Trigger Logic (GTL): The GTL combines data from the GMT and GCT and applies algorithms to the incoming trigger objects. 128 algorithm bits are sent via short flat cables to the Final Decision Logic.

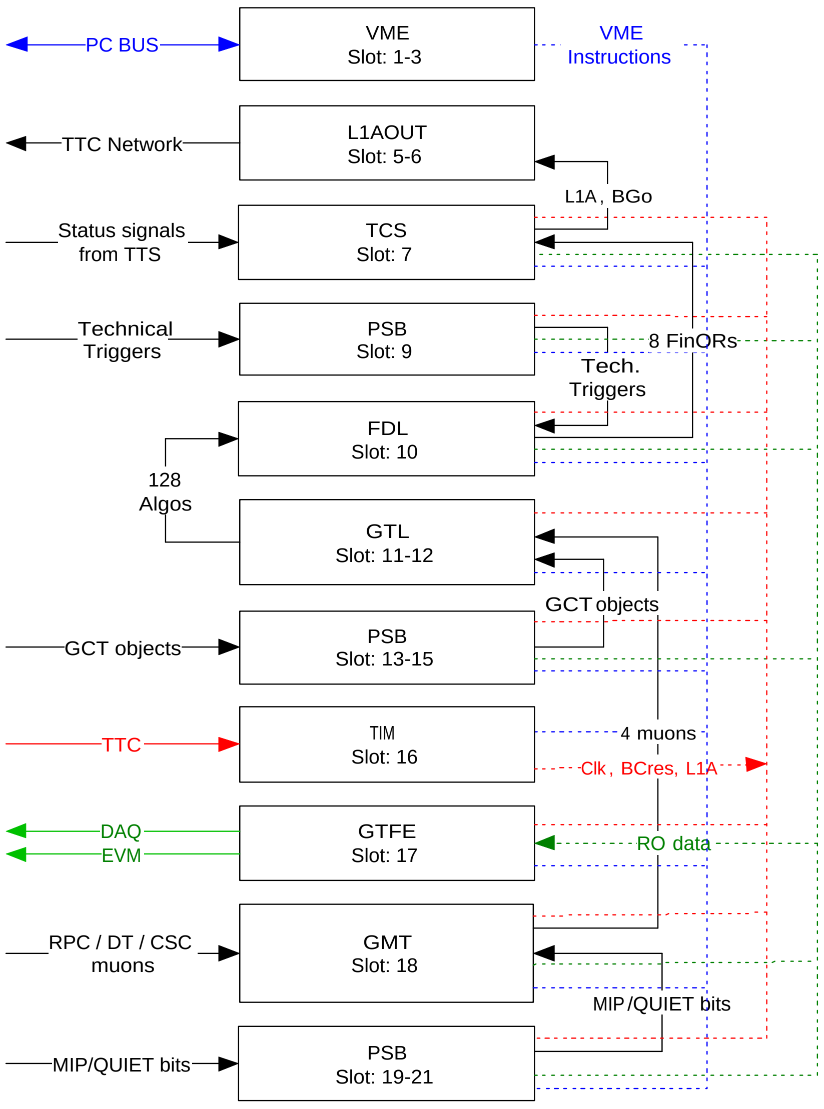


Figure 1.10: Simplified layout of the Global Trigger Crate

Final Decision Logic (FDL): The FDL receives 128 algorithm bits from the GTL and 64 TT bits from a dedicated PSB board, optionally downscales the rate of those and combines them according to a predefined trigger mask by a final OR function to generate the Level1 Accept (L1A) signal that starts the CMS readout. The 8 final OR (finOR) and 8 Technical Trigger (TT) signals are sent to the TCS.

Trigger Control System (TCS): The TCS ensures that the trigger subsystems are ready to receive L1A signals, depending on the status of the readout electronics or the data acquisition. The L1A signals are transmitted over the L1AOut modules to the TTC network.

Global Trigger Front End (GTFE): In case of a L1A the GTFE module collects the trigger data from all GT modules and sends them to the Data Acquisition system like any other subsystem. Figure 1.10 shows all modules that are read out.

VME Bus: Slots 1 to 3 house the standard CMS VME crate controller hardware. All GT modules apart from the L1AOut boards have a VME interface.

1.4 Control Mechanism

The CMS detector is a system composed of a great number of physically distributed devices that have to cooperate and interact with each other. Therefore sophisticated control systems are needed to hide the enormous complexity of the whole system to the people that shall operate the detector. In this chapter an overview of the architecture of the software infrastructure is given 1.4.1. Furthermore the mechanism how the Global Trigger hardware is controlled via the Trigger Supervisor (TS) framework is outlined 1.4.3.

1.4.1 Overall System Architecture

Figure 1.11 gives an idea of how the overall online software architecture of the CMS experiment was layed out. The architecture consists of four main parts:

- Data acquisition components manage the readout after a L1A and concatenate the data into a single data structure, the ”physics event”.
- Run Control and Monitor System (RCMS) provides a set of userfriendly interfaces that allow the user to operate the experiment.
- Detector Control System (DCS) is responsible for maintaining and monitoring the operational state of the experiment.
- Cross-platform DAQ framework: XDAQ is a distributed processing environment through which RCMS, DCS and DAQ components interoperate with each other.

The Global Trigger (GT) hardware will indirectly receive its orders from the RCMS that will be described in the next sub-section 1.4.2.

1.4.2 Run Control and Monitoring System (RCMS)

The Run Control and Monitoring System (RCMS) is defined as the collection of hardware and software components responsible for controlling and monitoring the CMS experiment during data taking [15].

The RCMS views the experiment as a collection of partitions. The Data Acquisition system may be divided into up to 8 partitions, which are read out independently to test and calibrate parts of the readout and trigger electronics in parallel. A partition is a configurable group of resources. Up to 32 subdetector groups (TTC-partitions) can be added to one DAQ-partition. Multiple DAQ-partitions can be active concurrently, sharing resources if necessary, allowing several sections of the experiment to run independently [23]. The running of a DAQ-partition is defined as a ”session”. Each session is associated with a Session Manager (SMR) that coordinates user access to the session and propagates commands from the users to the lower levels. For each of the involved

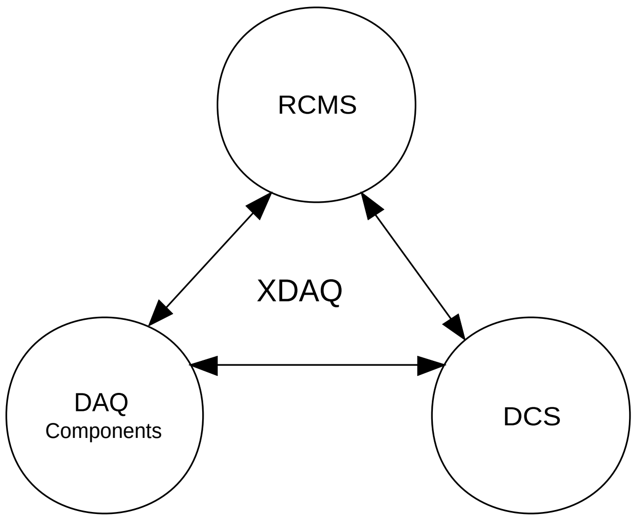


Figure 1.11: Architecture of the Online Software

subsystems there is a dedicated Function Manager (FM). Figure 1.12 gives an idea how this multisession approach of the RCMS works. For each session the according FM is instantiated dynamically. The Function managers define the interface to the subsystem application. In case of the Trigger Control this is the Trigger Supervisor (TS) [30] that is outlined in the next sub-section and described in detail in 3.2.

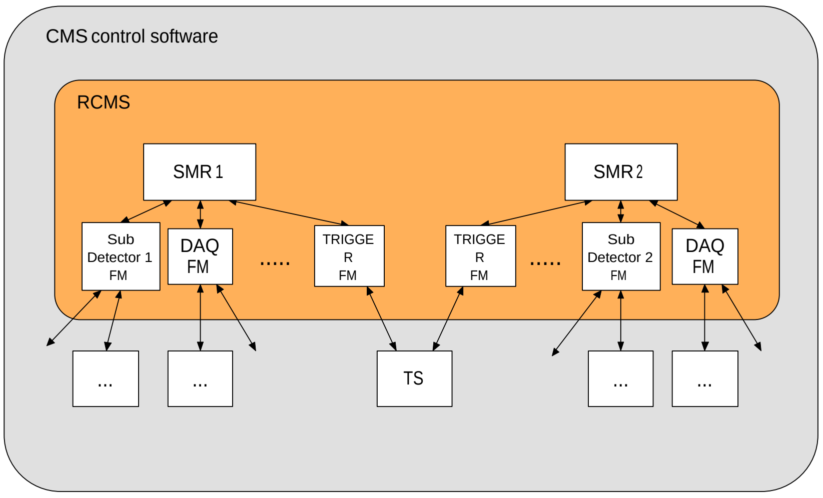


Figure 1.12: RCMS and Trigger Control in multiple partition running.

1.4.3 Trigger Supervisor (TS)

The Trigger Supervisor (TS) software system was designed to provide a framework to set up, test and monitor the trigger components and to manage their interplay and information exchange with the RCMS [30]. Figure 1.13 illustrates the architecture of the Trigger Supervisor (TS). The system consists of one central node, also denoted as ”Central Cell” and an arbitrary number of subnodes, so called ”Subsystem Cells”. On of these Subsystem Cells is the Global Trigger Cell (GT Cell) 1.4.4 that will access the hardware of the GT system.

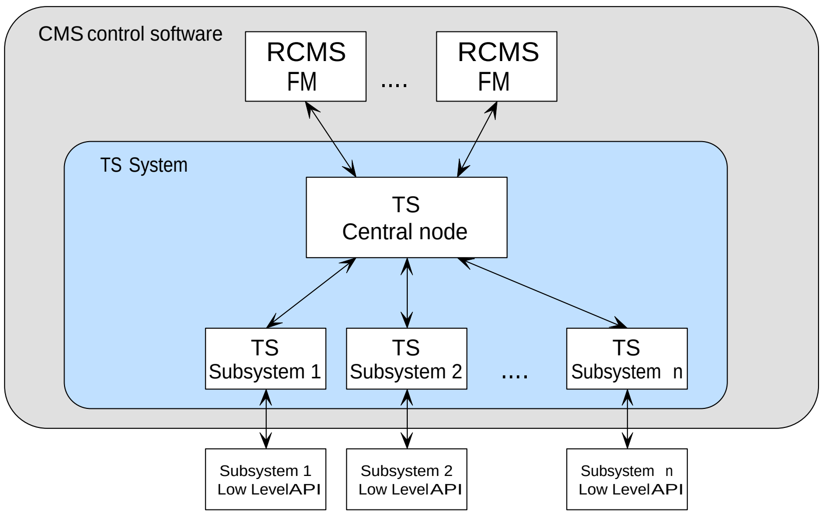


Figure 1.13: Trigger Supervisor Architecture.

The TS central cell can serve several Run Control (RC) sessions in parallel and propagates information to the Subsystem Cells. The Subsystem Cells are software skeletons that have to be implemented in order to be able to control hardware. Both the central and the subsystem nodes are XDAQ applications that use SOAP (Simple Object Access Protocol) [32] messaging for exchange of information. Section 3.2 takes a closer look at the functionality of the Trigger Supervisor Framework [41].

1.4.4 Global Trigger (GT) Cell

The GT Cell is the part of the TS system that acts as an interface to control and configure the GT hardware. To configure the GT hardware the GT Cell makes use of the database access infrastructure of the TS framework to read and write registers and load memories and firmware. Due to the well defined SOAP interface to the TS subsystem nodes there are several ways to send commands to the GT Cell. The TS Central Cell can either run in standalone mode configuring and controlling the GT Cell or propagate RCMS orders to the GT Cell. A generic TS Graphic User Interface (GUI) based on AjaXell [44], a C++ library implemented for the TS, permits access to the GT Cell over a web browser. For more fine grained control GT control panels also based on AjaXell are under development. The GT Cell uses C++ libraries 4.1 that access the hardware through VME utilizing the Hardware Access Library (HAL) [45]. There are also ”Expert GUIs” that use these libraries. Whereas the GT Cell currently only offers

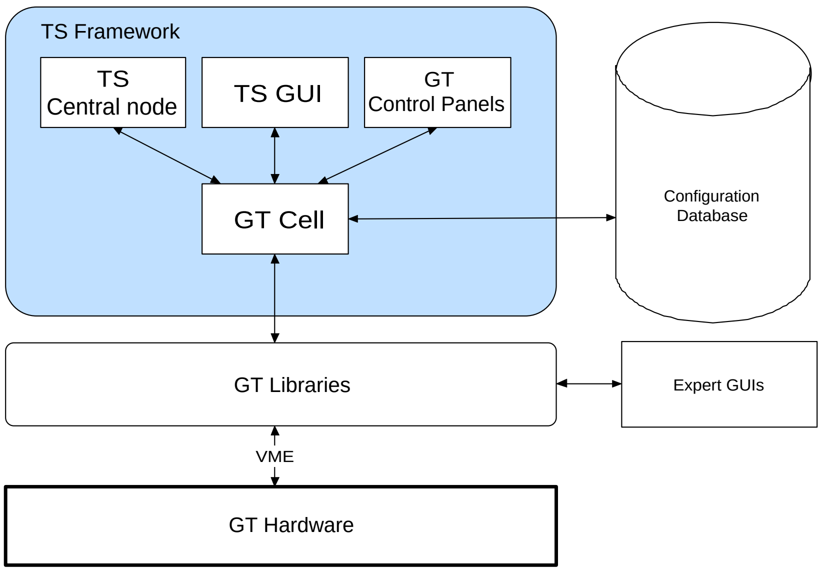


Figure 1.14: Possible scenarios of interaction with the GT hardware.

a very basic interface to the hardware, those Expert GUIs allow the most fine grained control of the GT hardware.

Chapter 2

Global Trigger Hardware

2.1 Timing Module (TIM)

The LHC clock corresponding to the Bunch Crossing Frequency of 40 MHz and the ”Bunch Crossing 0” (BC0) signal are used to synchronize the various parts of the detector electronics to each other and to the LHC-orbit structure. Along with the Level-1 Accept (L1A) signal, which is sent in case of a positive Level-1 Trigger decision, those signals are distributed by the Timing, Trigger and Control (TTC) system [14]. The Global Trigger (GT) crate receives those signals over a TTCrx chip mounted on the Timing Module (TIM) that broadcasts them over the back-plane to all other GT and GMT modules. For each module in the crate the corresponding delay register can be set in a way to synchronize all modules in the crate to each other. By default the TIM module sends signals to all boards in the crate. VME registers allow to inhibit signals to boards not installed in the crate.

The TIM module also accepts external clock, L1A and BC0 signals over LEMO connectors on the front panel. For debugging in standalone mode the 40 MHz clock can be provided by an onboard oscillator and a memory can be loaded to send simulated TTC messages and L1As periodically.

2.2 Pipeline Synchronizing Buffers (PSB)

The PSBs act as input modules for trigger objects sent to the GT or GMT. One PSB module can receive 8 channels of serial trigger data from the Global Calorimeter Trigger. One PSB channel corresponds to two trigger objects (2 isolated electron/photon objects, 2 central jet-objects, ...) from the GCT (1.3.3). Data from each channel are converted by receiver chips to 80-MHz parallel data multiplexing 2 GCT trigger objects in time. A synchronization chip synchronizes the channels to each other and to the LHC clock. A programmable delay for each channel is intended to compensate for any time skew between cables and link chips [26]. Finally data are sent to the logic module (GTL) over the backplane.

Each PSB module has 8 input channels corresponding to two trigger objects. Two channels carry data of one quadruplet of trigger objects from the GCT. One PSB can therefore receive up to 4 quadruplets. Two PSB modules are reserved for the 7 quadruplets the GCT sends to the GT leaving two PSB channels free. The third PSB dedicated to the GTL can receive trigger bits from other CMS subdetectors. Another PSB sends so-called Technical Triggers (TTs) directly to the Final Decision Logic (FDL) module where they can be used in the ”final OR” (FinOR) circuits like algorithms of the GTL.

Serial GCT data are transmitted to the PSBs over 4 ”Infiniband” connectors, each carrying data of one quadruplet. For other trigger bits and TTs, 16 RJ45 connectors can receive up to 64 Low Voltage Differential Signals (LVDS) at 40 MHz frequency.

Each of the eight input channels has a dedicated SIM/SPY memory that can either be used to spy on input data or to simulate input data transmitted to the backplane. Moreover channels 0 to 3 can also act as transmitters sending simulation data to another PSB module or to channels 4 to 7 on the same board. Another memory can provide data for comparison with input data.

2.3 Global Trigger Logic (GTL)

The Global Trigger Logic module (GTL) [24] is the heart of the Global Trigger (GT) system. Synchronized data from the Global Calorimeter Trigger (GCT) and Global Muon Trigger (GMT) are combined and conditions and algorithms are calculated on the two condition chips (COND1, COND2) from Altera company. The results (128 algorithm bits) are sent via short flat cables as parallel data to the Final Decision Logic (FDL).

Three receiver chips REC1, REC2 and REC3 receive 80-MHz trigger data over the back-plane and distribute them to the two condition chips. The GMT sends the 4 best muon objects that are combined into one quadruplet. The GCT sends 5 quadruplets: isolated electron/photon objects, non-isolated electron/photon objects, central jet objects, forward jet objects, tau-flagged jet objects. Information on total and missing transverse energy and 12 numbers of jets above different thresholds are combined into groups as well and received by the receiver chips.

As a first step, conditions for each group of objects are calculated. Energy or momentum thresholds and windows of the space coordinates η and ϕ are applied. For muons also the Minimum Ionizing Particle (MIP) and isolation information as well as the quality is checked. For the quadruplets containing total transverse energy and numbers of jets, thresholds are applied. For the missing transverse energy vector a ϕ-window can be chosen. Also conditions requiring two particles of the same or of different types having a specified distance in η and ϕ can be calculated. In the next step the previously applied conditions are combined by a simple AND-OR logic to form a trigger algorithm. As mentioned before, up to 128 algorithms can be calculated in parallel on the two conditions chips.

One set of conditions and algorithms is called a ”trigger menu”. Should other trigger algorithms be required for physics data taking or for a calibration task, a new trigger menu in the form of a firmware file for the condition chips is loaded using JTAG over VME. Setting up a new trigger menu basically involves two steps. First the conditions and algorithms are defined with a graphical setup software [28]. The output of this program is an XML [35] file that is processed by ”GTS”, a C++ program that generates the according corresponding VHDL [34] code. The Quartus synthesizer (Altera company's software for firmware processing from Altera company) translates the VHDL code and generates a new firmware file for the condition chips. The GTS program also generates a second XML file that defines the base addresses of thresholds for the conditions used in the trigger menu. Thresholds can be adjusted at any time by simple VME access, without creating a new trigger menu. Figure 2.1 shows the data flow diagram for the creation of a new trigger menu.

The GTL contains several memories on its FPGAs to allow a variety of self tests and interconnection tests. The receiver chips (REC1, REC2, REC3) contain memories to send simulated trigger data to the condition chips, according to the trigger objects they are intended to receive. Those memories can run also in spy mode monitoring data coming from the GMT or the GCT. On the condition chips there are memories that can either be used to spy out the results of the trigger logic or to insert simulated algorithm bits for the FDL.

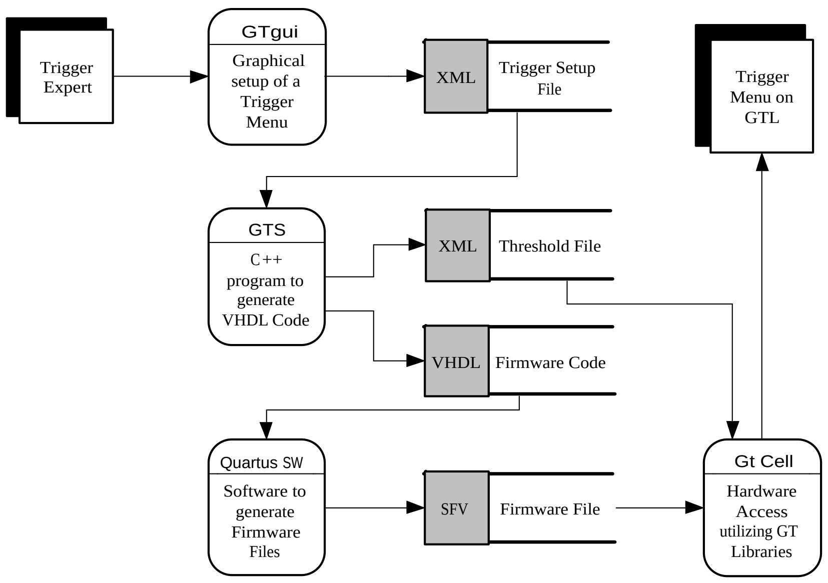

Trigger Expert
GTgui
Graphical setup of a Trigger Menu
XML
Trigger Setup File
Trigger Menu on GTL
GTS
C++ program to generate VHDL Code
XML
Threshold File
VHDL
Firmware Code
Quartus SW
Software to generate Firmware Files
SFV
Firmware File
Gt Cell
Hardware Access utilizing GT Libraries

Figure 2.1: Processes involved in the creation of a new Trigger Menu.

2.4 Final Decision Logic (FDL)

The Final Decision Logic (FDL) receives 128 algorithm bits from the Global Trigger Logic (GTL) and 64 Technical Trigger (TT) bits from a dedicated Pipeline Synchronizing Buffer (PSB) module. Algorithms and TTs together are denoted as ”slices”. Each Algorithm and or TT bit increments a rate counter for the corresponding slice. The periodicity of how often rate counters are reset to 0 can be chosen individually for each slice. If the rate is too high a pre-scaler reducing the rate by a certain factor can be applied. As mentioned in 1.4.2 the Data Acquisition (DAQ) of CMS can be divided into 8 partitions. Therefore the FDL combines the algorithm and TT bits into 8 Final-OR (FinOR) signals that are sent to the Trigger Control System (TCS). A veto mechanism permits to define each TT slice as veto-bit for one or more DAQ-partitions. In that case the Fin-OR signal is suppressed for the selected partitions. The FDL board contains also a SIM/SPY memory to either send simulated Algorithm and TT bits to the FinOR circuits or monitor data from the GTL and TT-PSB module. Another memory is permanently recording the 8 Fin-OR signals.

2.5 Trigger Control System (TCS)

The main tasks of the central Trigger Controller module (TCS) are to ensure that the subsystems are ready to receive Level-1 Accepts from the Global Trigger [24]. This information is partly provided by the synchronous Trigger Throttling System (sTTS) that sends warning signals to the TCS if buffers become nearly full, partly by Front-end Emulators that have been designed for the Tracker and Preshower detectors to avoid buffer overflows in their readout electronics. Furthermore the delivery of L1As has to comply with a set of trigger rules, like such as one that there have to be two untriggered crossings between two consecutive L1As [33]. The TCS is also responsible for the generation of synchronization and fast commands (2.1) as well as calibration and test trigger sequences to be distributed to the sub-detectors by the TTC network in addition to the L1A signals. All signals are sent over the back-plane to the Level-1 Accept Output (L1AOUT) module that acts as interface between Global Trigger and TTC network. Also the dead time of the experiment is monitored by dead time counters on the TCS.

Table 2.1: BGO control signals generated by the TCS.

NAME	DESCRIPTION
BC0	Resets the Bunch Crossing (BC) counters to begin a new LHC orbit.
Test Enable	Starts a calibration procedure.
Private Gap	Inhibits data taking to allowing a subsystem to perform private tasks.
Private Orbit	Inhibits data taking for one orbit.
Resync	Clears buffers and pipelines.
Hard Reset	Resets the Hardware.
Reset Event Counter	Resets the Event Counter to zero. Sent during at the end of a resynchronization procedure
Reset Orbit Counter	Resets the Orbit Counter to zero. Sent at the begin of a new run
Start	Starts data taking with the next orbit.
Stop	Stops data taking with the next orbit.

The logic of the TCS reflects the optional segmentation of the CMS readout system into 8 DAQ-partitions (1.4.2). The TCS chip contains 8 DAQ-partition Controllers (PTCs) running independently from each other with the sole limitation that only one DAQ-partition at a time is allowed to send triggers. at a given Bunch Crossing (BC). Therefore the TCS splits up the active time of a DAQ-partition according to the setting of the `TIMESLOT` registers. Each PTC provides the following functions:

- A Finite State Machine (FSM) runs control procedures according to the state of the connected TTC partitions. A detailed description of the states and responses of the FSM to signals from the TTS can be found in the TCS manual [27].
- A BGO Generator generates and sends BGO signals (2.1) to the TTC partitions of a DAQ-partition according to the FSM of the according PTC. The time behavior of the signal generation is derived from the register values SETTLE_TIME, ACTIVE_TIME and RECOVER_TIME.
- A BC-Table implemented as two 16 bit wide and 4 K long memories permits to define at which Bunch Crossing (BC) additional BGO control commands and other calibration signals will be sent. The format of files that can be loaded into the BC-Table memories can be loaded with is described in the Appendix (A.3.3).
- A Calibration logic to run calibration cycles. First a BGO control signal starts calibration procedures in the connected subsystems. After a certain time a L1A is issued to read out the calibration data.
- A Trigger Merger combines all trigger sources (Final-OR, Random Trigger, Calibration Triggers) into a L1A signal.
- A periodic Signal Generator generates signals according to register values that can be loaded over VME.

The assignment of TTC-partitions to the 8 DAQ-partitions is also done in the TCS chip through the `ASSIGN_PART` registers. One TTC-partition either belongs to exactly one DAQ-partition or is disabled so that it does not receive , not receiving any L1A or any other control signals.

The TCS sends L1As and BGO control signals to the L1AOUT module, where they are converted to Low Voltage Differential Signals (LVDS) and transmitted to 32 TTCci boards, which that manage further distribution them to the sub-detector crates.

2.6 Global Trigger Front End (GTFE)

A Level-1 Accept (L1A) sent to the TTC network by the TCS is also sent via the crate backplane to will return to the Global Trigger crate via the TIM and starts the readout of the Global Trigger (alternatively, it could also be sent to the TIM module via the TTC system). The FDL, TCS, GMT and PSB boards contain ring buffers to keep trigger data until the arrival of a read-out signal. When a L1A signal arrives on each a board a Readout Processor (ROP) moves the data from the ring buffer into a derandomizing buffer. Afterwards the ROP collects data from all derandomizing buffers, appends format words and sends it them as a record to the GTFE. On the GTFE merges data from all boards is merged into a GT/GMT record and sends them to the Data Acquisition system (DAQ). In parallel, data from the FDL and TCS are sent merged into an event record and sent to the Event Manager (EVM). A detailed description of the readout formats can be found on the Global Trigger homepage ([27]).he GTFE merges data from all boards is merged into a GT/GMT record and sends them to the Data Acquisition system (DAQ). In parallel, data from the FDL and TCS are sent merged into an event record and sent to the Event Manager (EVM). A detailed description of the readout formats can be found on the Global Trigger homepage ([27]).

Chapter 3

Framework

3.1 Cross-Platform DAQ Framework (XDAQ)

XDAQ is a framework designed specifically for the development of distributed data acquisition systems. It provides platform independent services, tools for local and remote inter-process communication, configuration and control, as well as technology independent data storage. To achieve these goals, the framework builds upon industrial standards, open protocols and libraries [15]. As mentioned in 1.4.3 all Trigger Supervisor nodes are XDAQ applications that make use of several services, that will be described in the following sections.

3.1.1 XDAQ Executable configuration and Partitions

At the startup a XDAQ process can be configured passing the path of configuration file as a command line argument. The configuration file contains the configuration schema of the XDAQ process, represented in XML [35]. A configuration is hierarchically structured into 3 levels:

Partition: Every configuration contains exactly one partition that is a collection of XDAQ processes hosting applications.

Context: Every context defines one XDAQ process uniquely identified by its URL that is composed of host name and port. A partition may contain an arbitrary number of contexts. The <module> tag inside the <context> tag specifies the location of shared libraries that have to be loaded in order to make applications available.

Application: The <application> tag uniquely identifies a XDAQ application. Each context can be composed of an arbitrary number of XDAQ applications. Applications can define properties that can be read during runtime using the <properties> tag.

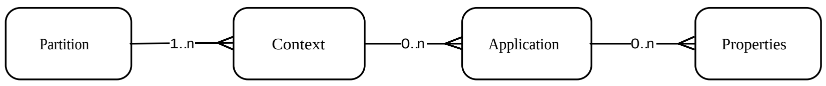


Figure 3.1: Multiplicity relations of a XDAQ-Configuration

The XDAQ configuration file in figure 3.2 describes a standalone scenario of the TS Global Trigger Cell with database access. The GT Cell is running on the first host configured with several properties that will be described later in this document (3.2.1). Two other XDAQ applications inside the same context provide control of the GT Cell (CellGui) and manage the access in standalone mode (CellAccess). The mentioned applications are compiled into two libraries. Their locations are given in the <module> tags. The second process runs on a different host with one single application loaded that acts as an interface to the Configurations Database (ConfDb). This application will be treated later in this chapter 3.1.3.

3.1.2 Simple Object Access Protocol (SOAP) communication

SOAP [32] is a means to exchange structured data in the form of XML-based [35] messages among computers over HTTP. XDAQ uses SOAP for a concept called Remote

```
<?xml version='1.0'?>
<!-- The following line contains a reference to an XSL file      -->
<!-- that allows the rendering of this XML file into an HTML file -->
<!-- <?xml-stylesheet type="text/xsl" href="xdaqConfig.xsl"?> -->

<xc:Partition
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:xc="http://xdaq.web.cern.ch/xdaq/xsd/2004/XMLConfiguration-30">

<xc:Context url="http://gtswpc3.cern.ch:3666">

        <xc:Application class="Cell" id="13" instance="3" network="local">
                <properties xmlns="urn:xdaq-application:Cell" xsi:type="soapenc:Struct">
                        <name xsi:type="xsd:string">GT</name>
                        <xhannelListUrl xsi:type="xsd:string">
                                file://${XDAQ_ROOT}/trigger/gt/ts/cell/xml/xhannelList/cell_xhannel_list_GtMTCC.xml
                        </xhannelListUrl>
                </properties>
        </xc:Application>

        <xc:Application class="CellGui" id="11" instance="3" network="local">
        </xc:Application>

        <xc:Application class="CellAccess" id="15" instance="0" network="local">
        </xc:Application>

        <xc:Module>file://${XDAQ_ROOT}/trigger/gt/ts/cell/lib/linux/x86/libCell.so</xc:Module>
        <xc:Module>${XDAQ_ROOT}/trigger/ts/access/cell/lib/linux/x86/libCellAccess.so</xc:Module>

</xc:Context>

<xc:Context url="http://cmsdaquser0.cern.ch:3667">

        <xc:Application class="TStore" id="120" instance="0" network="local">
                <properties xmlns="urn:xdaq-application:TStore" xsi:type="soapenc:Struct">
                        <views xsi:type="soapenc:Array" soapenc:arrayType="xsd:ur-type[1]">
                                <item xsi:type="xsd:string" soapenc:position="[0]">
                                        ${XDAQ_ROOT}/trigger/gt/db/tcs_registers_test_view.xml
                                </item>
                        </views>
                </properties>
        </xc:Application>

        <xc:Module>${XDAQ_ROOT}/daq/tstore/lib/linux/x86/libtstore.so</xc:Module>
        <xc:Module>${XDAQ_ROOT}/daq/tstore/tstore/lib/linux/x86/libTStore.so</xc:Module>

</xc:Context>

</xc:Partition>
```

Figure 3.2: Example for a XDAQ configuration file for a GT Cell with database access.

Procedure Calls (RPC). This means that the SOAP message contains an XML tag that is associated with a function call, a so called ”callback”, at the receiver side. That way all XDAQ applications belonging to one XDAQ-partition 3.1.1 can interchange information with each other and execute procedures on remote XDAQ nodes. It is even possible to send SOAP messages from outside a XDAQ-partition using the curl [39] library for instance.

3.1.3 Access to the Configurations Database (TStore)

Access to a database is a basic requirement for the configuration of the trigger components. The XDAQ framework provides access to the Configuration Database (ConfDB) through an application called TStore.

Data exchange utilizing TStore is done through SOAP messages. A SOAP message specifying the name of a TStore-view that determines which data should be read or written, is sent to the TStore application. The SOAP reply contains relevant tables as attachments. TStore-views are part of an XML-configuration file for the TStore application. A configuration of TStore can contain an arbitrary number of TStore-views. A detailed explanation how to configure the TStore application is given in ref. [38].

3.1.4 Web Interface

XDAQ offers an infrastructure that allows to make XDAQ applications ”web exposed” applications. If a browser sends a certain request to the XDAQ executable, a callback inside a XDAQ application will dynamically create HTML [36] code making use of the C++ library cgicc [40]. The html code is displayed by the web browser and that way the user can interact with a XDAQ-application. In the TS framework this interface was extended using Asynchronous XML and JavaScript Technology (AJAX) 3.2.1.

3.2 Trigger Supervisor

An overview on how the Trigger Supervisor central node interacts with the Run Control and Monitoring System (RCMS) as an interface to the Trigger Control was given in 1.4.3. This chapter will go more into detail explaining aspects of the TS framework used in the implementation of the Global Trigger (GT) Cell as well as important features of the TS system related to the Trigger Control, due to the fact that the hardware is partly located in the GT crate 2.5.

3.2.1 Trigger Supervisor Framework

This section will focus on the TS as a software framework. First the functionality and interaction of the software components will be described. Afterwards the customization procedure of the TS framework to fit subsystem requirements is described. The detailed description of how the customization of the TS framework was done in case of the GT system is given in 5.

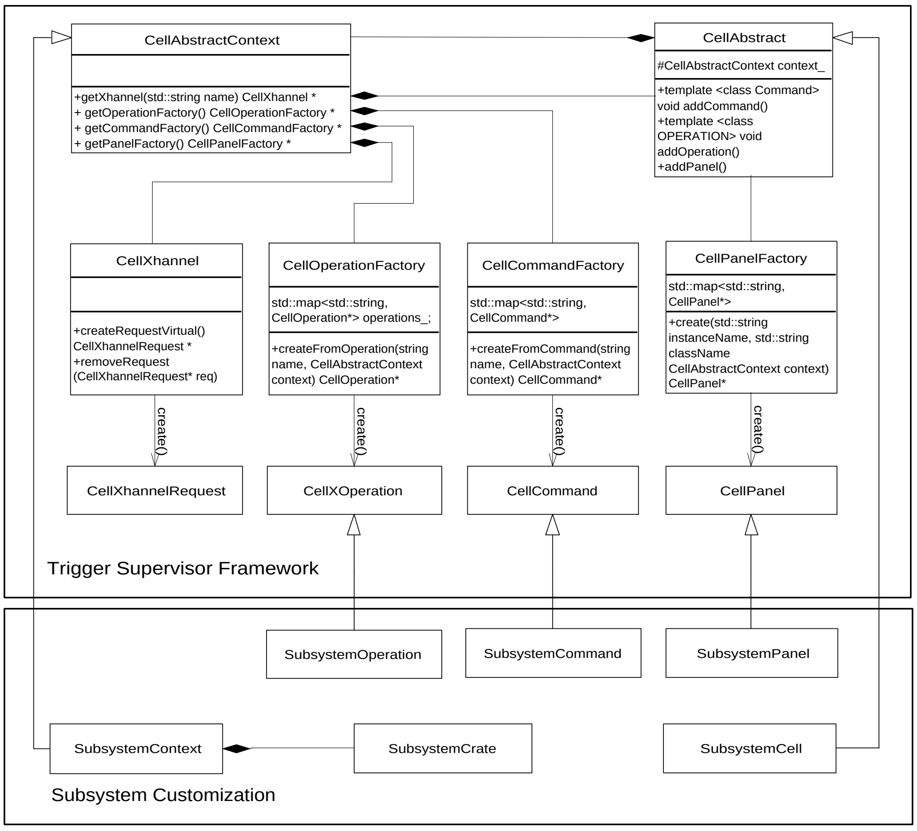

CellAbstractContext
+getXhannel(std::string name) CellXhannel *
+ getOperationFactory() CellOperationFactory *
+ getCommandFactory() CellCommandFactory *
+ getPanelFactory() CellPanelFactory *
CellAbstract
#CellAbstractContext context_
+template <class Command> void addCommand()
+template <class OPERATION> void addOperation()
+addPanel()
CellXhannel
+createRequestVirtual() CellXhannelRequest *
+removeRequest (CellXhannelRequest* req)
CellOperationFactory
std::map<std::string, CellOperation*> operations_;
+createFromOperation(string name, CellAbstractContext context) CellOperation*
CellCommandFactory
std::map<std::string, CellCommand*>
+createFromCommand(string name, CellAbstractContext context) CellCommand*
CellPanelFactory
std::map<std::string, CellPanel*>
+create(std::string instanceName, std::string className CellAbstractContext context) CellPanel*
create()
create()
create()
create()
CellXhannelRequest
CellXOperation
CellCommand
CellPanel
Trigger Supervisor Framework
SubsystemOperation
SubsystemCommand
SubsystemPanel
SubsystemContext
SubsystemCrate
SubsystemCell
Subsystem Customization

Figure 3.3: Components of the TS framework and Subsystem Customization classes in an UML diagram.

Framework Components

Figure 3.3 shows a UML diagram of the most important TS framework components as well as a possible scenario of derived or customized subsystem classes. In the following lines the abstract classes and the functionality they offer, the dynamic creation of functional classes like Commands, Operations and Control Panels and the Xhannel Architecture that acts as a means of communication, will be described.

Cell Application: The functionality of the Cell Application can be divided into three domains. XDAQ services are made available for the Cell through inheritance from the `xdaq::Application` class. A Cell Application can therefore be added to a XDAQ partition, making it browseable through the XDAQ Web Interface 3.1.4 and allowing Remote Procedure Calls (PRCs) through callbacks. In the `CellAbstract` class TS framework functionality is implemented. Template functions allow to add implemented Commands and Operations, Xhannels are created dynamically according to a XhannelList file and generic callbacks for SOAP messages according to calls of Commands or Operations are implemented. Each subsystem `Cell` class inherits from the `CellAbstract` class and adds custom Commands, Operations and Control Panels in the implementation.

Context: The functionality of the context of the TS framework can again be grouped into XDAQ related functions, that are accessed over a pointer member to the `xdaq::ApplicationContext`, that manages URLs and URNs, instances and ids of modules loaded into the XDAQ executable 3.1.1. These resources are made available to other TS components through a getter method in the CellAbstractContext. Furthermore the `CellAbstractContext` class implements TS framework functionality like access to factory classes that dynamically create Commands, Operations or Control Panels and Xhannels to access external resources. If any other resources should be made available for a Subsystem Cell, the access should be made through a descendant of the CellAbstractContext. The GT makes the C++ API to access the hardware available for the GT Cell that way 3.3.

Xhannel Architecture: To get access to external resources or other TS nodes from a TS Cell the Xhannel architecture was implemented in the TS framework. Xhannels define transparent interfaces that decouple the development of external services and Subsystem Cells [42]. Xhannels are created during startup by the `CellContext` from an XML file that is defined through a property of the Subsystem Cell in the configuration schema file of the XDAQ-Executable 3.1.1. An example of this xhannelListFile can be found in the Appendix A.3.1. Currently Xhannels and the corresponding Requests to other TS Cells, to the TStore application and to the monitoring infrastructure are implemented 3.4.

Command Interface: The TS Command Interface is a SOAP interface that allows to make Remote Procedure Calls (PRCs) inside a Trigger Supervisor node. A SOAP message sent to a TS node following a certain syntax 3.5 will result in the execution of a so called ”callback function” that attempts to instantiate the command according to the SOAP message and execute it and return a SOAP message that informs the sender about status of the execution. To comply with the requirements for a distributed control system the TS framework implements

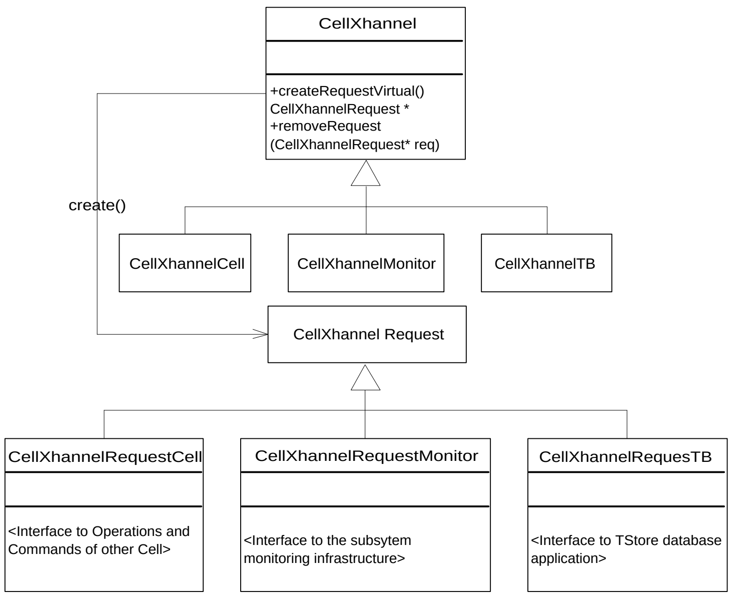
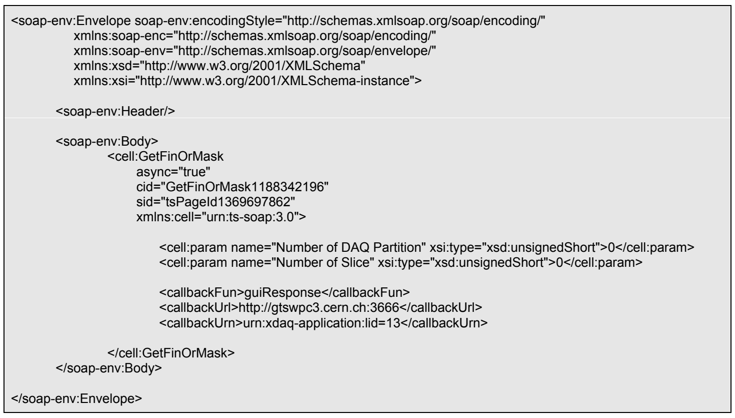

CellXhannel
+createRequestVirtual()
CellXhannelRequest *
+removeRequest
(CellXhannelRequest* req)
create()
CellXhannelCell
CellXhannelMonitor
CellXhannelTB
CellXhannel Request
CellXhannelRequestCell
<Interface to Operations and Commands of other Cell>
CellXhannelRequestMonitor
<Interface to the subsytem monitoring infrastructure>
CellXhannelRequesTB
<Interface to TStore database application>

Figure 3.4: Currently implemented descendants of CellXhannel and CellXhannelRequest.

an asynchronous protocol for commands. In case of synchronous commands the sender of the command waits until the execution of the command is complete and gets back the response from the command. During this time also the sender of the command is blocked. For commands that are expected to take longer until their execution is complete the asynchronous protocol should be used. The receiver of an asynchronous command replies immediately acknowledging the reception. Once the command is executed the real reply message is sent back to the sender.

Figure 3.5 shows an example XML file for a SOAP command. The information required to execute the command is packed into the SOAP body inside the <soap-env:Body> tag. The name of the command - GetFinOrMask - is required to create the command. The attributes *cid* and *sid* are identifiers for the command and of the TS Session that is running it. The *async* attribute determines the mode of execution. Parameters of the command are passed as child tags. The last three tags, *callbackFun*, *callbackUrl*, and *callbackUrn* are identifiers where to send the reply of the command and are required only if the command runs asynchronously.

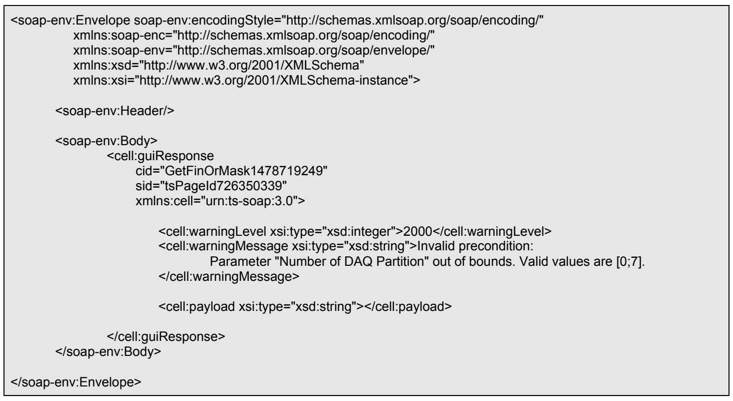
The reply SOAP message that is returned to the sender after the execution of the command contains all relevant information about the status of the command. In case of a successful execution the message contains its return value in the <payload> tag. If some error occurred the tags <warningLevel> and <warningMessage> are filled with information about the error. There are 3 default warning levels - INFO, WARNING, ERROR [42]. In case of INFO or WARNING it may happen that the reply contains a payload, too. Figure 3.6 shows an example of a SOAP reply of a command that was executed without



```
<soap-env:Envelope soap-env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            xmlns:soap-enc="http://schemas.xmlsoap.org/soap/encoding/"
            xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

        <soap-env:Header/>

        <soap-env:Body>
                <cell:GetFinOrMask
                     async="true"
                     cid="GetFinOrMask1188342196"
                     sid="tsPageId1369697862"
                     xmlns:cell="urn:ts-soap:3.0">

                         <cell:param name="Number of DAQ Partition" xsi:type="xsd:unsignedShort">0</cell:param>
                         <cell:param name="Number of Slice" xsi:type="xsd:unsignedShort">0</cell:param>

                         <callbackFun>guiResponse</callbackFun>
                         <callbackUrl>http://gtswpc3.cern.ch:3666</callbackUrl>
                         <callbackUrn>urn:xdaq-application:lid=13</callbackUrn>

                </cell:GetFinOrMask>
        </soap-env:Body>

</soap-env:Envelope>
```

Figure 3.5: Example of a Command SOAP message.

success.



```
<soap-env:Envelope soap-env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            xmlns:soap-enc="http://schemas.xmlsoap.org/soap/encoding/"
            xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

        <soap-env:Header/>

        <soap-env:Body>
                <cell:guiResponse
                     cid="GetFinOrMask1478719249"
                     sid="tsPageId726350339"
                     xmlns:cell="urn:ts-soap:3.0">

                         <cell:warningLevel xsi:type="xsd:integer">2000</cell:warningLevel>
                         <cell:warningMessage xsi:type="xsd:string">Invalid precondition:
                                   Parameter "Number of DAQ Partition" out of bounds. Valid values are [0;7].
                         </cell:warningMessage>

                         <cell:payload xsi:type="xsd:string"></cell:payload>

                </cell:guiResponse>
        </soap-env:Body>

</soap-env:Envelope>
```

Figure 3.6: Example of a Command Reply SOAP message containing an error.

Operation Interface: For complex control sequences the command interface may not be appropriate. Therefore the TS framework offers Operations, which is a protocol that can be represented by a Finite State Machine. The states, possible transitions and code that should be executed during transitions, as well as errors, warnings and other notifications and a set of parameters for the Operation can be chosen according to requirements. Three skeletons of ”standard operations” that should be implemented for each subsystem already exist in the CVS-repository. Those are the Configuration, Selftest and Interconnection Test Operations that will be controlled for the whole trigger system from the TS Central Cell. 3.2.2.

Control Panels: A server-side C++ library implemented for the TS framework called AjaXell [44] allows Subsystems to create Graphical User Interfaces (GUIs), also called ”Control Panels”, for their Subsystem Cells. AjaXell is based on the Asynchronous Java Script and XML (Ajax) technology and has been tested with the Apache Tomcat and the XDAQ servers. Unlike simple HTML user interfaces the AjaXell library offers a bigger variety of widgets to create desktop like GUIs and accelerates the interaction with the back end by exchanging smaller amounts of data. The Control Panels interact with the Command interface just like an external sender of SOAP messages.

Framework Customization

The development of the TS requires the distribution of its implementation among the subsystem groups. Depending on how the interface to manipulate the hardware is defined, there are different ways of customizing the TS framework [42]:

1. If there exists a web service based approach to control the hardware like a SOAP interface, the Subsystem Cell should interact with the hardware using the Xhannel infrastructure of the TS framework.
2. If the software used to control the hardware is based on XDAQ as well, the according libraries and the Subsystem Cell library can be loaded into the same XDAQ executable. In this case public methods would act as a communication channel between the Subsystem Cell and the hardware.
3. If there exists a C++ API to control the hardware of a subsystem, the corresponding libraries have to be used in the code of Commands and Operations for the Subsystem Cell. The Cell Application has to run locally in the corresponding subsystem server. This scenario describes the situation of the GT and will therefore be described in depth in 5.

3.2.2 Trigger Supervisor System

As mentioned 1.4.3, the TS was designed to setup and test the trigger components, hiding the complexity of the involved sub components to the RCMS. How these tasks are achieved will be described in this section, focusing on the Global Trigger as a subsystem of the Trigger Supervisor.

Setup of the Trigger Hardware - Configuration Operation

Earlier in this document it was described how the TS system interacts with the RCMS. Up to eight Session Managers (SMs) according to the same number of running DAQ-partitions, communicate over SOAP with the TS Central node. The Central node coordinates the access to resources that are shared among DAQ-partitions and ensures that different sessions do not interfere with each other. This goal is achieved with the following configuration procedure, explained as a Finite State Machine diagram in 3.7:

The first RCMS Function Manager (FM) that connects to the TS Central node initiates a Configuration Operation and executes the first transition ”Configure” with a key as a parameter. The key corresponds to a full configuration of the trigger system that is common for all DAQ-partitions. A ConfDb table of the TS associates this key with keys for all subsystems and passes them as parameters to the according subsystem Configuration Operations. In this way the whole trigger system is put into a well defined working state. The next FMs attempting to connect to the TS Central have to provide the same Configuration key as a parameter to be able to move to the Configured state. They are not allowed to reconfigure the trigger system, as this would lead to inconsistencies with the first FM. FM2 to FM8 can execute the ”Partition” transition with a partition key as parameter that is associated with ”partition dependent” data that

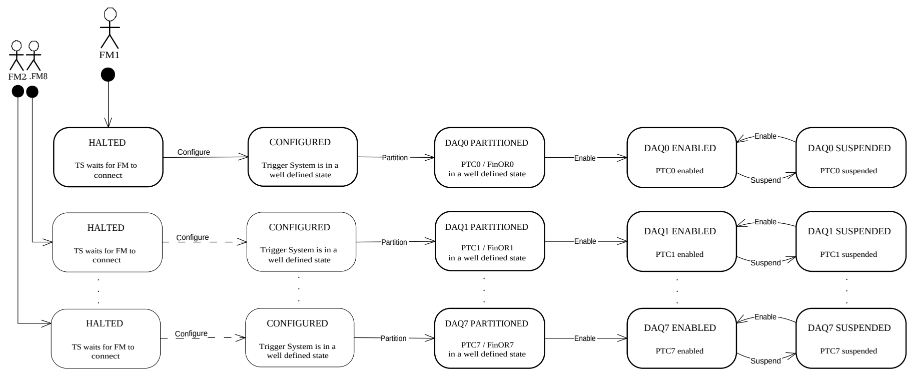


Figure 3.7: The Finite State Machine of the Trigger Configuration Operation.

affect just the DAQ-partition the corresponding FM is controlling. Partition dependent data are:

- A TTC vector that assigns TTC partitions to a certain DAQ-partition.
- A FinOR vector that defines which Algorithms and Technical Triggers should be used to trigger a DAQ-partition.
- A BC Table that defines which Bunch Crossings should be used for triggering and which fast and synchronisation signals should be sent to TTC-partitions belonging to one DAQ-partition.

Among all subsystems the GT Cell plays a special role in the Configuration of the Trigger System. The *Configure* transition in the FSM of the TS triggers a *Configure* transition of the FSM of the GT Configuration Operation. The *Partition* transition however will make use of the GT Command Interface 5.3. The intelligence that assures that FMs cannot interfere with each other is implemented in the TS Central Cell.

Testing inside the TS framework

All TS nodes can run in standalone mode, which gives trigger subsystems the possibility to implement testing routines for their hardware. Simple Commands 3.2.1 could be used to first write a register and read it back, which ensures that VME access of a module is working correctly. The skeleton of a Selftest Operation could be implemented to perform more complex testing procedures. Subsystems can even implement new custom operations if a certain testing routine should require them.

Interconnection Tests (ITs) are a way of testing interoperability of trigger components. Interconnection Tests involve at least two subsystem nodes and the TS Central node that coordinates the test. A simple scenario of an IT would be to send simulation data from one subsystem to another where the data is read out and compared with the input. As the TS framework does not restrict the number of involved subsystems ITs could be much more complex than that.

Chapter 4

Global Trigger Software

4.1 Global Trigger Libraries

The Global Trigger Software Libraries are C++ classes based on the Hardware Access Library (HAL) [45]. HAL is a library developed to provide high level access to VME hardware modules. Built on top of the HAL the GT Libraries are the next step in hiding complexity of the hardware to software experts. Shared libraries for each type of GT board provide high level methods to control the module. Loading memories from ASCII files, loading new firmware into PROMs using the JTAG Access Library [46] or writing a sequence of registers defined in an ASCII file are features of the GT Libraries that used by the Configuration Operation of the GT Cell.

Standalone applications for every GT module are based on the GT Libraries and have been used for debugging the GT hardware for a long time, which makes the GT Libraries a well tested and reliable base for other software. Figure 4.1 shows a diagram that illustrates the hierarchy of libraries the GT Cell library relies on.

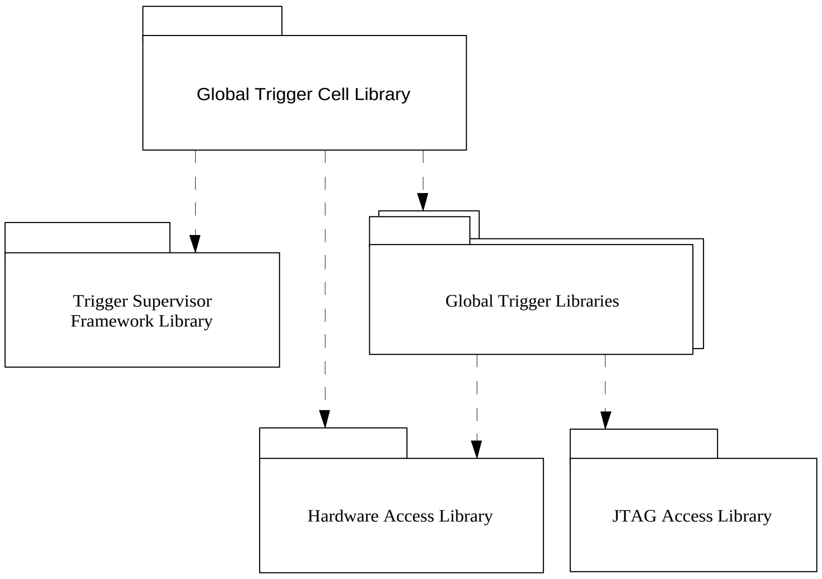


Figure 4.1: The most important libraries the GT Cell depends on.

4.2 Global Trigger Control Panels

The Global Trigger system as the highest instance of the L1-Trigger plays a crucial role in physics data taking. Setting up the trigger menu is done in the Global Trigger Logic (GTL), applying prescales to the trigger algorithms in the Final Decision Logic (FDL) and the Central Trigger Control (TCS) manages the distribution of Level1 Accepts. Unlike other subsystems the setup of these components has to be done more frequently and even during a data taking period parameters might change. Therefore it has been decided that a user friendly intuitive user interface is needed for the GT Cell that is the only entry point to the GT hardware inside the TS framework. A prototype of these GT

Control Panels based on AjaXell has been developed. The Control Panels communicate with the GT Cell via the implemented SOAP interface.

4.3 Global Trigger Cell

The Global Trigger (GT) Cell as a Trigger Supervisor (TS) component was described in 1.4.4, this chapter will outline possible usage scenarios of the GT Cell, whereas the next chapter (5) will describe the software components of the GT Cell in depth.

4.3.1 GT Cell as part of TS system

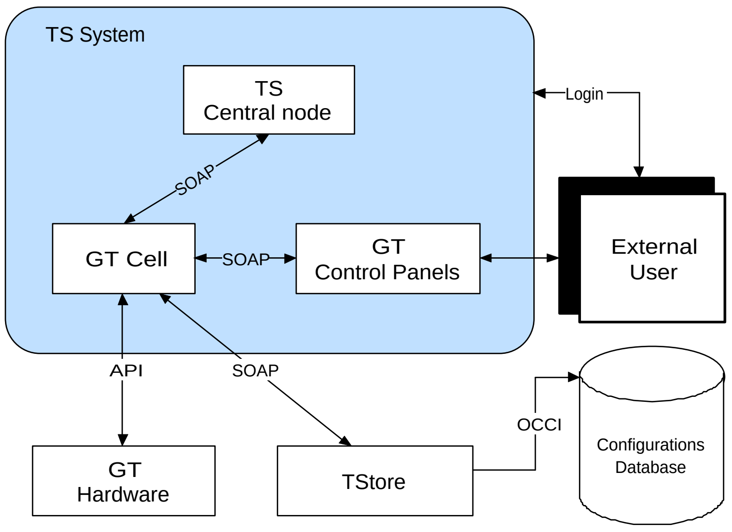


Figure 4.2: The GT Cell as part of the TS system.

Being part of the TS system, accessed through the SOAP interface, is the default running mode for the GT Cell. Database driven configuration operations and interconnection tests involving other subsystems are controlled directly by the TS Central node. Access to the GT Control Control panels used to control shared trigger resources like the FDL and TCS via the command interface of the GT Cell will managed by a login-mechanism in the TS. This way different tasks are split upon involved software components in a Model View Controller (MVC) like architecture that separates data storage (Model) from the presentation of the data to the user (View) and the underlying intelligence that determines the behavior of the application (Controller):

Model: The GT Cell stores and retrieves data from the database or writes and reads data from the hardware, which can be seen as another way of storing data.

View: The GT Control Panels are the View presenting data provided by the Model to the user.

Controller: The TS acts as the Controller in this Context implementing the intelligence that manages the manipulation of the data.

Figure 4.2 shows a schematic sketch of components involved in the process of controlling the GT hardware and their way of interaction.

4.3.2 GT Cell in standalone mode

Like every TS node the GT Cell can run in standalone mode, which is essential to test newly implemented Commands and Operations with the hardware. For filling the GT Configurations Database (ConfDb) it will also be necessary to use the GT Cell in combination with the GT Control Panels, as the GT standalone applications do not have access to the database. It is possible that the standalone mode of the GT Cell will also be used for hardware tests, once the Cell is evolved enough. Figure 4.3 graphically shows the interaction of the involved components. A way of filling the database with configurations for the GT system will be proposed later in this document (5.4.1).

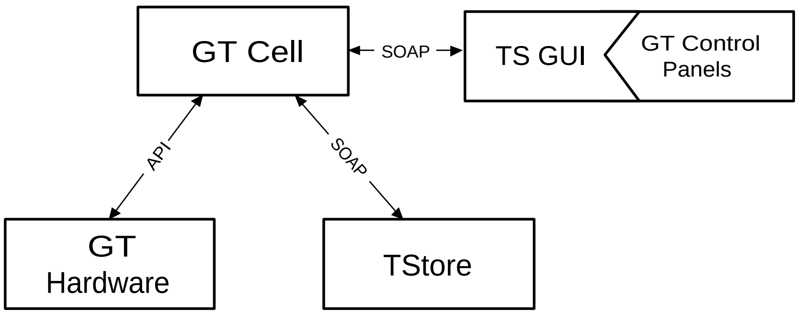


Figure 4.3: The GT Cell in standalone mode and required components.

Chapter 5

Software Integration

5.1 Global Trigger Cell Objects

5.1.1 CellContext

The `CellContext` class inherits from the class `CellAbstractContext` that implements the management of the interplay of different TS framework components like different Subsystem Cells, TS Central node, Xhannel Architecture, etc. The CellContext is created in the constructor of the `Cell` object like all vital components of the TS framework. The Global Trigger CellContext extends the functionality by providing access to the CellGTCrate object that will be described in the next section.

5.1.2 CellGTCrate

The `CellGTCrate` class has been implemented to manage the dynamical creation of board objects defined in the Global Trigger Libraries (4.1). The fact that the hardware plugged into the Global Trigger crate changes very frequently, according to the requirements of a test setup, lead to the decision that the software objects would have to be instantiated dynamically. This is done by utilizing the crate scan that is implemented in the Hardware Access Library (HAL) for VME64x crates. For each slot of the crate it is checked for a VME64x module. In case of success the number of the slot and the serial number of the device are put into an associative map that is kept for the lifetime of the CellGTCrate object. This map is the base for the next step in the initialization of the crate. For each entry it is ensured that the module identified by its serial number is a Global Trigger module by checking if the serial number appears in the HAL *ModuleMapper.dat* file. All serial numbers of GT modules are listed in this file with an identifier of their type. This type is used by the crate object to determine which software objects to initialize, which is done as the final initialization step. Objects according to the type of board are initialized and pointers to them stored in an associative map with their serial number. Providing access to this pointers the hardware can now be manipulated through the CellGtCrate object. The code snippet below show how this works in practice.

```
gtCrate_ = new CellGTCrate(*logger_);
try {
        string mmf = "ModuleMapperFile.dat";
        string atf = "AddressTableMap.dat";
        string busAdapter = "CAEN";
        gtCrate_->initializeCrate(mmf, atf, busAdapter);
}
catch( CellException& e) {
        string msg;
  msg = "Errormessage";
        XCEPT_RETHROW(CellException, msg, e);
}

TcsBoard* board;
try {
        board = gtCrate_->getBoardFromSerialNumber("TCS");
}
catch ( CellException& e) {
        string msg;
        msg = "Errormessage"';
        XCEPT_RETHROW(CellException, msg, e);
}

uint32_t chipId;
try {
        chipId = board->getTCSChipId();
}
catch ( HardwareAccessException& e) {
        string msg;
        msg = "Errormessage";
        XCEPT_RETHROW(CellException, msg, e);
}
```

The CellGTCrate object is created and initialized in the constructor of the Cell-Context. The parameters of the initialization are hard coded as they should not change. The type of busadapter that should be used for hardware access is defined through a compiler flag. This way it is ensured that the user can only work with the busadapters that are present as libraries on the machine the GT Cell has been compiled on.

5.1.3 GtCommand

The `GtCommand` class is one of the base classes of all Commands for the GT. Whereas the base class `CellCommand` makes a Command part of the TS framework, the `GtCommand` class implements some methods that are frequently used in GT Commands. Mostly these are checks if number parameters are in a certain range or string parameters are part of a certain vector of possible strings. Any code that could be used in more GT Commands should go into this class.

5.2 GTBase - An Extension to the GT Libraries

Procedures like the initialization of the GT crate or the configuration operation have to work dynamically. In this context this means, that no matter which boards are in the crate or intended to be configured, certain routines have to be executed. This requires a common base class for all software objects referring to a GT board. The `GtBoard` class has been implemented therefore, providing a set of methods to load memories, firmware or a set of registers in a generic way. Errors that occur in the `GtBoard` class like caught exceptions from the HAL are rethrown as `GtExceptions` by the time being. If a more complete set of exceptions should be required those should inherit from the `GtException` class.

Another part of the GTBase library is the `GtHttpDownloader` class that facilitates easy download of files over HTTP [47]. This functionality is used frequently for downloading firmware and memory files that are located in the GT afs-repository that will be described in the appendix (A.2).

5.3 Global Trigger Command Interface

As described in 4.3.1 the Global Trigger Command Interface will be utilized by the Global Trigger Control Panels and by the TS Central Cell to access the hardware. According to the needs of these Control Panels the interface has been implemented, focusing on the resources of the Global Trigger that have to be accessed frequently and by non trigger experts. Those are the Central Trigger Control (TCS), the Final Decision Logic (FDL) and partly the Global Trigger Logic (GTL). Furthermore there are some

Commands that are called from inside the Configuration Operation. In this chapter an overview of the functionality of the Command Interface is given.

5.3.1 Command Customization

To make new commands available for the GT Cell the following steps have to be undertaken:

1. The GT libraries are checked if they offer methods appropriate for the requested Command. If this is not the case the responsible has to implement new methods.
2. A set of parameters for the command is chosen with the person that would like to make use of the Command.
3. The skeleton for a CellCommand descendant is downloaded and adapted. The methods `init()` (5.1), `code()` (5.3) and `preCondition()` (5.2) are implemented as described in the TS User Manual [42]
4. The source file of the new Command is added to the Makefile and compiled into the GT Cell library.
5. The Command is added through a template function in the Cell.cc (5.4), to make it available during runtime.
6. The Command can now be created and executed by the TS generic AJAX GUI.

The code snippets below illustrate the above described procedure.

Listing 5.1: Defining parameters in the init() method.

```
void ExampleCommand::init()
{
  paramList_["Item"] = new xdata::String();
  paramList_["Value"] = new xdata::UnsignedInteger();
  paramList_["Serial Nr."] = new xdata::String();
}
```

Listing 5.2: Implementing the preCondition() method.

```
bool ExampleCommand::preCondition()
{
CellContext* gtCellContext = dynamic_cast<CellContext*>(context_);

if( !gtCellContext->getGtCrate() )
{
getWarning()->setMessage ("Crate is not initialized.");
getWarning()->setLevel(CellWarning::ERROR);

return false;
}

string serNr = paramList_["Serial Nr."]->toString()
else if(!gtCellContext->getGtCrate()->checkBoardInitialized(serNr))
{
getWarning()->setMessage ("Requested Board not initialized." );
getWarning()->setLevel(CellWarning::ERROR);

return false;
}

else return true;
}
```

Listing 5.3: Implementing the code() method.

```
bool ExampleCommand::code()
{
CellContext* gtContext = dynamic_cast<CellContext*>(context_);
CellGTCrate* crate = gtCellContext->getGtCrate();

xdata::String* pSerNr;
xdata::String* pItem;
xdata::UnsignedInteger* pVal;

pSerNr = dynamic_cast<xdata::String*>(paramList_["Serial Nr."]);
pItem = dynamic_cast<xdata::String*>(paramList_["Item"]);
pVal= dynamic_cast<xdata::UnsignedInteger*>(paramList_["Value"]);

string sItem = pItem->toString();
string sSerNr = pSerNr->toString();
unsigned int iVal = *(pVal);

try
{
GtBoard* pGtBoard = crate->getBoardFromSerialNumber(sSerNr);

gtBoard->write(sItem, iVal );
}
catch ( CellException& e)
{
string msg = e.what();

getWarning()->setMessage ( getWarning()->getMessage() + msg);
getWarning()->setLevel(CellWarning::ERROR);
}

ostringstream result;

result
<< "Board Serial: " << sSernNr << endl;
<< "Item: " << pItem << endl;
<< "Value: " << iVal << endl;

this->payload_ = new xdata::String(result.str());
}
}
```

Listing 5.4: Adding a new Command in the Cell.cc.

```
void Cell::init()
{
  addCommand<ExampleCommand>();
}
```

5.3.2 General Command Issues

Naming Conventions and Return Values

Due to the large number of Commands required to allow fine grained control of the GT hardware through the SOAP interface of the GT Cell the following guidelines have been established to keep the implementation transparent:

- The class-names of GT Commands start with the prefix *"Gt"*.
- If is implemented for a certain GT module another prefix according to the type of the board is added (*"Fdl"*, *"Tcs"*, *"Gtfe",...*).
- Commands that are used to read out data from a module ("getters") have a return value of the appropriate datatype. (xdata::UnsignedShort, xdata::UnsignedLong, xdata::String, xdata::Boolean, ...)
- Commands that are used to write data to a module ("setters") have a return value of xdata::String containing relevant information concerning the write access.

Commom Command Errors

Common for all GT Commands in the `preCondition()` method is a check that ensures that the `CellGTCrate` object is initialized. Commands that are related to a certain board check if the board object has been initialized 5.5. Boards that are plugged into the crate should normally be initialized automatically at the startup of the Cell application. The Command GtCommonGetCrateStatus can be used to check manually, which board objects have been instantiated by the `CellGTCrate` class.

Listing 5.5: Checks done before the execution of every GT Command.

```
bool ExampleCommand::preCondition()
{
  CellContext* gtContext = dynamic\_cast<CellContext*>(context\_);

  if(!gtCellContext->getGtCrate()) {
    getWarning()->setMessage ("Crate is not initialized.
    Use command InitCrate for initialization
    and execute this command again.");
    getWarning()->setLevel(CellWarning::ERROR);

    return false;
  }

  else if (!gtCellContext->getGtCrate()->getTCS()) {
    getWarning()->setMessage ( "TCS Board is not initialized.
    Check if it is in the crate.");
    getWarning()->setLevel(CellWarning::ERROR);

    return false;
  }
        else return true;
}
```

All Other checks in GT Commands are related to parameters given to the Command and inform the user about the range of valid values for the parameter. In case of the GT Control Panels addressing the SOAP interface of the GT Cell it may be assured by a GUI widget that no invalid values are passed as parameters. But as the SOAP interface is exposed to any other controller as well, parameters have to be checked for consistency as well on this level.

5.3.3 FDL Commands

The FDL is one of the GT modules that is required to be accessible during the configuration of the Trigger System 3.2.2. Being able to manipulate the Final-Or and Veto circuits of the FDL is essential to partitionate the trigger. After the configuration of the trigger it is necessary to observe the L1A-rates for each FDL-slice and optionally apply a prescaler to a certain algorithm or technical trigger.

Table 5.1: Description of Parameters used in FDL Commands

NAME	TYPE	VALID VALUES
Number of Slice	xdata::UnsignedShort	The Number of FDL slices depends on the firmware. Currently there are 192 slices foreseen on the FDL. Valid values for the parameter are therefore [0;191].
Number of DAQ-Partition	xdata::UnsignedShort	The Number of DAQ-partitions is 8. Therefore valid values are between [0;7].
Prescale Factor	xdata::UnsignedLong	Value of the prescaler for a slice that is determined by a 16 bit register. Range of valid values is [0;65535].
Update Step Size	xdata::UnsignedLong	Value of the update step size is determined by a 16 bit register. Range of valid values is [0;65535].
Bit for Refresh Rate	xdata::UnsignedShort	Each of 8 bits refer to a different multiplicity that is defined in the firmware of the FDL. Valid values are between [0;7].

SetFinOrMask:

Description:	Each Slice can be added to the Final-OR of one or more DAQ-partitions (2.4). This Command adds or removes a specific Slice to or from a DAQ-partition's Final-Or circuit according to the *"Enable for FinOR"* parameter.
Parameters:	Number of Slice Number of DAQ-Partition Enable for FinOR
Return Value:	Slice number: *"Number of Slice"* *"enabled/disabled"* for FinOR in DAQ partition number: *"Number of DAQ-Partition"*

GetFinOrMask:

Description:	Reads out whether a Slice is currently part of the Final-OR of a certain DAQ-partition
Parameters:	Number of Slice Number of DAQ-Partition
Return Value:	`xdata::Boolean`

SetVetoMask:

Description:	Each Slice can suppress a L1A for one or more DAQ-partitions 2.4. This Command enables or disables that Veto-mechanism for a given Slice and DAQ-partition.
Parameters:	Number of Slice Number of DAQ-Partition Enable for Veto
Return Value:	Slice number: *"Number of Slice"* *"enabled/disabled"* as Veto for DAQ partition number: *"Number of DAQ-Partition"*

GetVetoMask:

Description:	The getter to the above described Command. Reads if a certain Slice is a currently defined as Veto for a certain DAQ-partition.
Parameters:	Number of Slice Number of DAQ-Partition
Return Value:	`xdata::Boolean`

SetPrescaleFactor:

Description:	To control L1A rates that are too high, a prescale factor for each Slice can be applied. This factor can be set individually for each Slice with this Command. Setting the factor to 0 or 1 does no prescaling.
Parameters:	Number of Slice Prescale Factor
Return Value:	Prescale Factor of Slice Number: *"Number of Slice"* set to: *"Prescale Factor"*

GetPrescaleFactor:

Description:	Reads out the Prescale Factor for a certain FDL Slice.
Parameters:	Number of Slice
Return Value:	xdata::UnsignedLong

ReadRateCounter:

Description:	Reads out the rate counter for a certain slice.
Parameters:	Number of Slice
Return Value:	xdata::UnsignedLong

SetUpdateStepSize:

Description:	Sets the common step-size for the reset period of all rate counters.
Parameters:	Update Step Size
Return Value:	Update Step Size set to: *"Update Step Size"*

SetUpdatePeriod:

Description:	Sets the update-period of the rate-counter for a certain Slice, based on the common update step-size.The update-period is chosen by setting a bit of registers. Each bit corresponds to a factor the common update-period is multiplied with ([27]). An array in the code of the command maps bitnumbers to multiplicities.
Parameters:	Number of Slice Bit for Refresh Rate
Return Value:	Update Period of Slice Number: *"Number of Slice"* set to: *"multiplicity"*

GetNumberOfAlgos:

Description:	Depending on the version of the firmware of the FDL chip, the number of algos may differ. This Command gives back the number of algorithms currently implemented.
Parameters:	
Return Value:	`xdata::UnsignedShort`

GetNumberOfAlgos:

Description:	Depending on the version of the firmware of the FDL chip, the number of Technical Triggers (TTs) may differ. This Command gives back the number of TTs currently implemented.
Parameters:	
Return Value:	`xdata::UnsignedShort`

5.3.4 TCS Commands

The Central Trigger Control (TCS) controls the distribution of L1As 2.5 and therefore plays a crucial role with respect to Data Acquisition (DAQ) and Readout of the trigger components of CMS. The Global Trigger Control Panels 4.2 will provide very fine grained control over the TCS through the TCS Command Interface of the GT Cell. Assigning TTC-partitions to DAQ-partitions, assigning Time Slots, controlling the Random Trigger Generator, generation of fast and synchronisation signals and loading predefined Bunch Crossing Tables separately for each DAQ partition are tasks the Command Interface has to cope with. Commands of the TCS can be grouped into Commands affecting more than one Partition Controller (PTC) and PTC dependent commands. The first group of Commands therefore contains the prefix ”Master” whereas Commands of the second group start with ”Ptc”. The second group of Commands has the number of the PTC as a common parameter.

Table 5.2: Description of Parameters used in TCS Commands

NAME	TYPE	VALID VALUES
DAQ Partition	xdata::UnsignedShort	The Number of DAQ-partitions is 8. Therefore valid values are between [0;7].
Number of PTC	xdata::UnsignedShort	For each DAQ-partition there is a PTC implemented on the TCS chip. Therefore valid values are between [0;7].
Detector Partition	xdata::UnsignedShort	This parameter refers to one of 32 TTC partitions. Valid values are between [0;31].
Time Slot	xdata::UnsignedShort	The Time Slot for a PTC is calculated from a 8 bit value. Valid values are between [0;255].
Random Trigger Frequency	xdata::UnsignedLong	The Random Frequency is calculated from a 16 bit register value. Valid values are between [0;65535].

MasterSetAssignPart:

Description:	This Command assigns a TTC-partition (Detector-Partition) to a DAQ-partition. In case the TTC-partition is already part of a DAQ-partition it will be assigned to the new partition anyway.
Parameters:	Detector Partition DAQ Partition
Return Value:	Detector Partition: *”Detector Partition”* Assigned to DAQ-Partition: *”DAQ Partition”*

MasterGetAssignPart:

Description:	Returns the number of the DAQ-partition a certain TTC-partition is part of.
Parameters:	Detector Partition
Return Value:	`xdata::UnsignedShort`

MasterSetAssignPartEn:

Description:	This Command enables or disables a TTC-partition. Before a TTC-partition can be assigned to a DAQ-partition it has to be enabled.
Parameters:	Detector Partition Enable Partition
Return Value:	Detector Partition *enabled/disabled*

MasterGetAssignPartEn:

Description:	Reads out if a certain TTC-partition is enabled or not.
Parameters:	Detector Partition
Return Value:	`xdata::Boolean`

MasterStartTimeSlotGen:

Description:	Depending on the registers that define the time slots for every DAQ-partition the time slot generator switches between the DAQ-partitions in round robin mode. This command starts the time slot generator.
Parameters:	
Return Value:	Time Slot Generator Started.

PtcSetTimeSlot:

Description:	Sets the time slot for a certain Partition Controller (PTC)
Parameters:	Number of PTC Time Slot
Return Value:	Time Slot for PTC *"Number of PTC"* set to *"Time Slot"*

PtcGetTimeSlot:

Description:	Returns the current Time Slot assignment for a certain PTC.
Parameters:	Number of PTC
Return Value:	`xdata::UnsignedShort`

PtcStartRndTrigger:

Description:	Starts the random trigger generator for a specified PTC.
Parameters:	Number of PTC
Return Value:	Random trigger generator started for DAQ partition controller *"Number of PTC"*

PtcStopRndTrigger:

Description:	Stops the random trigger generator for a specified PTC.
Parameters:	Number of PTC
Return Value:	Random trigger generator stopped for DAQ partition controller *"Number of PTC"*

PtcRndFrequency:

Description:	Sets the frequency of generated Triggers by the Random Trigger generator for a specified PTC.
Parameters:	Number of PTC Random Trigger Frequency
Return Value:	Random Frequency of Partition Group: *"Number of PTC"* set to: `"Random Trigger Frequency"`

PtcGetRndFrequency:

Description:	Reads out the frequency of the Random Trigger generator for a PTC.
Parameters:	Number of PTC
Return Value:	`xdata::UnsignedLong`

PtcStartRun:

Description:	Starts a run for a Partition Controller (PTC), by first resetting and starting the PTC and then sending a start run command pulse.
Parameters:	Number of PTC
Return Value:	Run started for PTC: *"Number of PTC"*

PtcStopRun:

Description:	Stops a run for a PTC.
Parameters:	Number of PTC
Return Value:	Run stopped for PTC: *"Number of PTC"*

PtcCalibCycle

Description:	Starts a calibration cycle for the specified PTC.
Parameters:	Number of PTC
Return Value:	Calibration cycle for DAQ-partition: *"Number of PTC"* started.

PtcResync:

Description:	Manually starts a rescynchronisation procedure for the specified PTC.
Parameters:	Number of PTC
Return Value:	Resynchronization procedure for DAQ-Partition `"Number of PTC"` initialized.

PtcTracedEvent:

Description:	Manually sends a traced event for a specified PTC.
Parameters:	Number of PTC
Return Value:	Traced event initiated for DAQ-Partition `"Number of PTC".`

PtcHwReset:

Description:	Manually sends a hardware reset to the PTC.
Parameters:	Number of PTC
Return Value:	Hardware for DAQ-Partition: *"Number of PTC"* has been reset.

PtcResetPtc:

Description:	Resets the state machine of the PTC.
Parameters:	Number of PTC
Return Value:	PTC: *"Number of PTC"* reset.

5.3.5 Other Commands

Several Commands not specifically implemented for a certain type of GT module but rather used during the initialization, for debugging or for filling the database with register data. Those will be described in this section.

Table 5.3: Description of Parameters used in non-board-specific Commands

NAME	TYPE	VALID VALUES
Item	xdata::String	Refers to a register item, defined in the HAL AddressTable file for a module. If the specified item is not found the HAL will throw an Exception that is caught in the Command.
Offset	xdata::UnsignedInteger	The Offset to the register address specified by an Item parameter according to the HAL AddressTable file. In case the Offset gets too large, a HAL Exception caught by the Command will indicate that.
Board Serial Number	xdata::String	Only Serial Numbers of GT modules that are initialized will be accepted. The ”GetCrateStatus” Command returns a list of boards in the crate.
Bus Adapter	xdata::String	The GT Cell only accepts busadapters of type ”DUMMY” and ”CAEN”.
Module Mapper File	xdata::String	The full path to the HAL ModuleMapper file has to be specified. If the file is not found a HAL exception caught by the Command will inform the user about that.
AddressTableMap File	xdata::String	The full path to the HAL AddressTableMap file has to be specified. If the file is not found a HAL exception caught by the Command will inform the user about that.

GtCommonRead:

Description:	To read out register values from any GT module in the crate, which is useful for debugging, this Command was written. When correctly using the Offset parameter also lines of memories can be read out.
Parameters:	*Item* *Offset* *Board Serial Number*
Return Value:	`xdata::UnsignedLong`

GtCommonWrite:

Description:	Generic write access for all GT modules is provided by this Command.
Parameters:	*Item* *Value* *Offset* *Board Serial Number*
Return Value:	Register Value for Item: ”*Item*” set to: ”*Value*” (offset=”*Offset*”) for board with serial number: ”*Board Serial Number*”

GtInitCrate:

Description:	The initialization of the GT crate (5.1.2) is done during startup of the Cell application. If the creation of the crate object did not work correctly or if another type of bus adapter or different HAL files should be used this Command is used. Only if the ”*Reinitialize Crate*” parameter is set to true a new `CellGTCrate` object is instantiated and registered to the `CellContextClass`.
Parameters:	*Module Mapper File* *AddressTableMap File* *Bus Adapter* *Reinitialize Crate*
Return Value:	The GT crate has been initialized with ”*Bus Adapter*” busadapter. Board with serial nr.: ”*Board1 Serial Number*” in Slot Nr. ”*Board1 Slot Number*” Board with serial nr.: ”*Board2 Serial Number*” in Slot Nr. ”*Board1 Slot Number*” . .

GtGetCrateStatus:

Description:	The crate object dynamically creates associative maps during its initialization where information about modules in the crate is put. This information can be read out using this Command.
Parameters:	
Return Value:	The GT crate has been initialized with *"Bus Adapter"* busadapter. Board with serial nr.: *"Board1 Serial Number"* in Slot Nr. *"Board1 Slot Number"* Board with serial nr.: *"Board2 Serial Number"* in Slot Nr. *"Board1 Slot Number"* . .

GtInsertBoardRegistersIntoDB:

Description:	This Command reads out all registers for a specified GT module that are Configuration Database (ConfDb) (5.4.3) registers and inserts a row of values with a Primary Key and optionally a description into the according database table.
Parameters:	Board Serial Number Primary Key Description
Return Value:	Register Values have been read from the hardware and inserted into table *"Name of Register Table"* with Primary Key: *"Primary Key"*

5.4 Global Trigger Configuration Database

5.4.1 Introduction

Access to the Configuration Database (ConfDB) is a very important and useful feature during the testing period and is essential during the physics data taking runs of the LHC to be able to trace back how the hardware had been configured for a certain set of data.

As the name implies the ConfDB is meant to store data to configure the GT hardware. Configuring the hardware in this contents means to set registers, check and possibly load firmware to the FPGAs, load memories and anything else that is required to put the hardware in a well defined working state. There will be many different configurations for the Global Trigger (GT) hardware depending on what the purpose the global trigger should be configured for. A simple selftesting procedure 5.5, a more complex interconnection test involving several other systems like the GMT or the GCT or a physics run. To set up the GT hardware for such purposes the GT Standalone Applications will be used offering a more fine grained control of the hardware. After the setup is complete and the hardware in a working state, the GT Cell will be used to store the information in the ConfDB that is needed to achieve the same state at any other time.

5.4.2 General Database Concepts

One very common concept for design of database schemas is the Relational Model. In this model the the data is stored in multiple tables that are linked by keys. A key in this contents is collection of one or more columns in one table whose values match corresponding columns in other tables. This concept helps to make the database schema easier to understand, more flexible and easier to maintain.

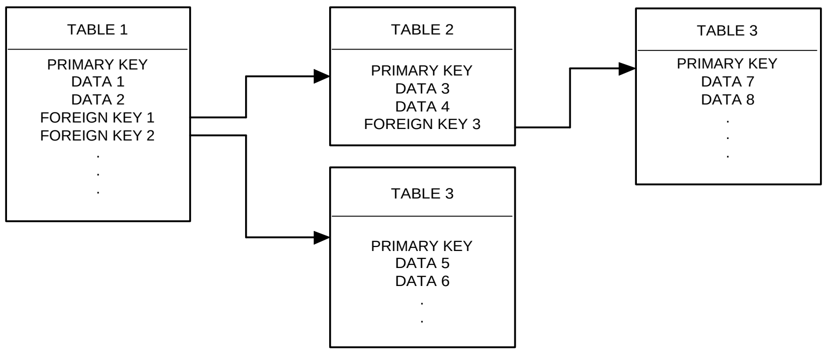


Figure 5.1: Simple example of a Relational Database Model schema

Fig 5.1 shows conceptually the layout of a relational database schema. Every table has one unique identifier that is called Primary Key (PK). Additionally each table can hold some data and one or more Foreign Keys (FK) that are referencing other tables where this value is a Primary Key. This procedure can repeat as often as necessary to group the data into intuitive and understandable structure. How this was done in case of the Global Trigger Configuration Database is described in the next section.

5.4.3 Global Trigger Database Implementation

When looking at the design process of the GT ConfDB the following aspects should be mentioned. Neither the developers of the database schema nor people working with the database in the future are or will necessarily be database experts, but rather physicists. Moreover people working with the database will change frequently over the working period of the experiment. Therefore the above described relational design model 5.4.2 was chosen. This way the schema is kept simple and intuitive as it somehow reflects the hardware. Another strength of the concept is the easy extensibility that has already proved itself in the process of development.

Figure 5.2 shows how the relational model was applied in case of the schema for the GT ConfDB. The top level table GT_CONFIG contains links to the BOARD-_CONFIG tables. For each board in the crate that should be configured there can be a foreign key (FK) that is associated with a primary key (PK) in the corresponding BOARD_CONFIG table. To avoid storage of redundant data all PSB and GTL keys point to just one table. The BOARD_CONFIG tables contain links to resources of the board that should be configured during a configuration. Those resources are the firmware that can be configured using JTAG over VME, memories that are loaded from files, HAL sequencer files and register values. Loading new firmware will play

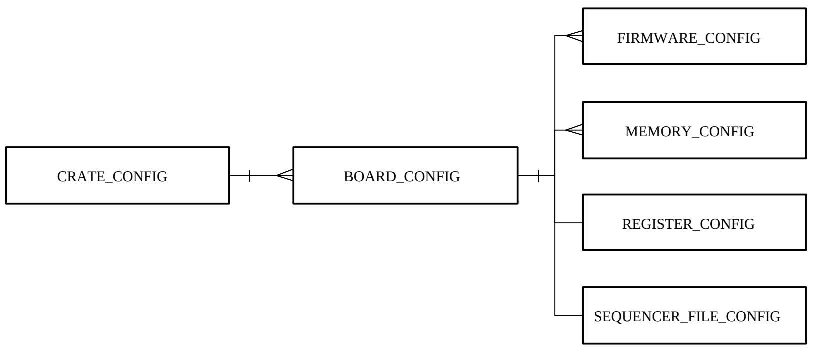


Figure 5.2: The conceptual design of the GT ConfDB.

a less important role for most of the boards apart from the GTL where the new trigger menu is loaded as firmware for the FPGAs frequently. Loading memories is especially important for the TCS, where BC tables are implemented as memories for the Partition Controllers (PTCs) 2.5. For Interconnection Tests (ITs) it might be useful to load SIM/SPY memories on the PSBs or the GTL from the ConfDb as well. The main usage for HAL sequencer files will be, to send a sequence command pulses that initiate certain processes on a GT module. This is necessary because registers that are stored for each board in the ConfDB have to be read- and writable. As described 5.4.1 there has to be the possibility to read out a register configuration from the hardware and write it to the database.

Figure 5.3 shows the full contents of the GT_CONFIG table and the names of the tables it is linked to. For each board a subset of tables like in 5.4 exists. Depending on the number of memories and JTAG chains to be loaded per board the tables differ. Also the BOARD_REGISTERS table that contains the names of registers that should be loaded is different for every board. Only the rightmost four tables of each board contain real configuration data that is values of registers in case of the register tables or links to files on the AFS-repository in case of firmware, memory or sequencer files. In case of the TCS there is no firmware branch because it is not possible to load new firmware over VME for this module.

5.5 Global Trigger Operations

5.5.1 Global Trigger Configuration

The GT Configuration Operation and the GT ConfDb schema were designed to flexibly cope with a variety of different configuration options for the trigger hardware. The Configuration Operation relies on the consistency of the hardware with the chosen ConfDB-key. Figure 5.5 illustrates how the GT Configuration Operation attempts to configure the hardware depending on the state of the hardware and the ConfDB data according to a key. If pieces of the hardware should not be configured during a certain configuration, the FKs referencing the configuration data should be left empty.

1. A row of the GT_CONFIG table is retrieved from the database with the key that is given as a parameter to the Operation. If a certain board should not be configured

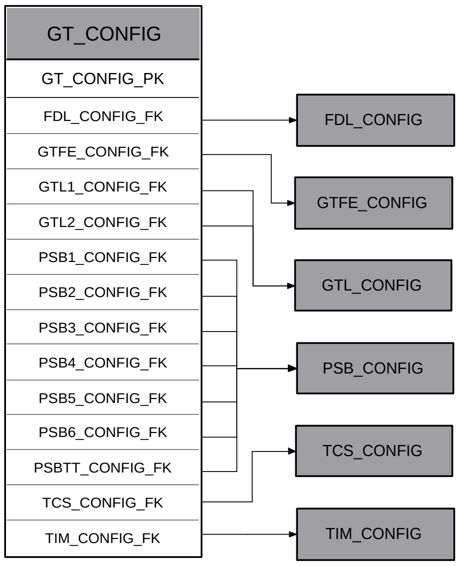


Figure 5.3: The GT_CONFIG table references the different board tables.

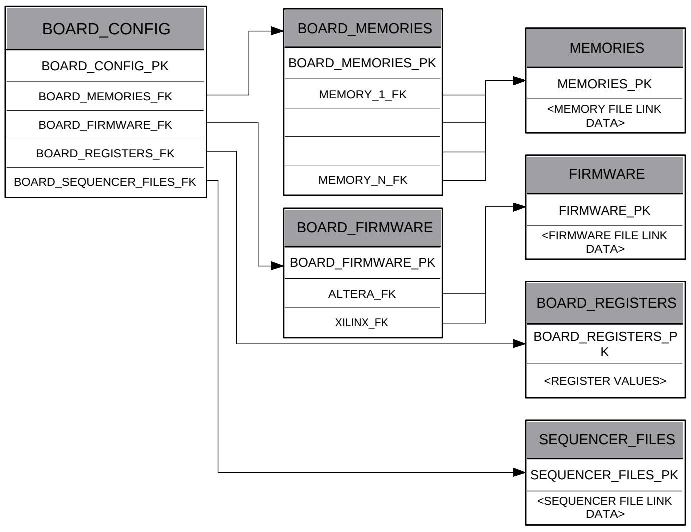


Figure 5.4: Each BOARD_CONFIG table references a set of sub tables.

at all, the according FK entry in the GT_CONFIG table has to be left empty. If there is no row with this PK the Configuration Operations is stopped with an ERROR.

2. A loop over all boards that are defined in the `CellGTCrate` class is done. Boards not found in the crate are logged.

3. For all boards that are initialized in the board it the BOARD_FIRMWARE table is retrieved. New firmware is attempted to be loaded for JTAG chains referenced

in this table. New firmware is only loaded when the version numbers of the current firmware does not match the firmware version of the configuration.

4. The same loop is done over all possible memories for a board, that are found in the BOARD_MEMORIES table. Empty links are omitted just like above.
5. The register table for each board is retrieved. If this table is empty because of a missing link, a warning message is issued, because loading registers is essential to put the hardware into a well defined state.
6. Finally a sequencer file is attempted to be downloaded for every board. This sequencer file can be used to write any register for a board, whereas the registers table of a board can only contain read and writable registers of a board.
7. The above procedure is executed for every board in the crate, before the operation ends successfully.

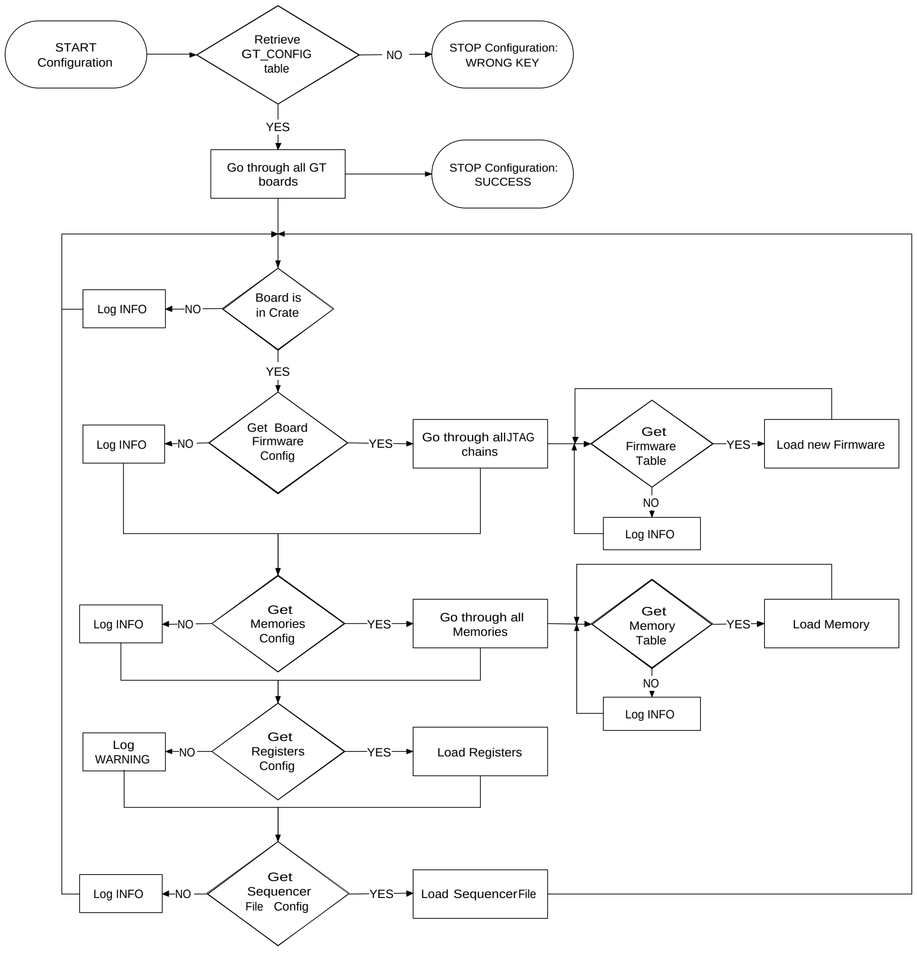


Figure 5.5: Flowchart of the GT Configuration Operation.

The Configuration Operation of the GT Cell involves many components and might therefore need some more explanation for people working with it in the future. This following sections will therefore explain details of the implementation and provide some pieces of code for better understanding.

Database Access using Xhannels and Tstore

To make use of TStore 3.1.3 to retrieve data from the ConfDB one or more views have to be defined in the configuration file of the XDAQ-executable TStore belongs to. For the GT Configuration Operation one view A.2 with two queries was defined. The first view is used to retrieve a row from one table uniquely identified by the primary key column. In step 1 of the GT Configuration Operation a row of the GT_CONFIG table is read from the database A.4.

- The Xhannel connects to the specified view, passing the password for the GT ConfDb.
- A request is created by the Xhannel and a map of parameters matching those in the used Tstore query A.2 is created.
- The name of the query that should be used is specified and given to the request like the parameters are.
- The request is sent to the TStore application through the Xhannel.
- The reply of the query contains a xdata::Table with a row of FKs.

Steps 3,4,5 and 6 issue the second type of query "getRowFromChildTableForBoard" to determine whether firmware, memories, registers or a HAL sequencer file should be loaded or not. The firmware and memory branch of the configuration of a GT board differ from the register and sequencer branch, as there may be more than one firmware or memory file to be loaded per board. Therefore a loop is done over all columns of the retrieved board firmware or memories table and another simple query to retrieve the information from where to download the according file is done. The code snippet in the appendix A.4.2 shows the code of the firmware branch in the Configuration Operation.

- The Xhannel connects to the specified view, passing the password of the GT ConfDb.
- A request is created by the Xhannel. The parameter map is created due to specified configuration branch, board serial number, and configuration key for the board.
- A query of type "getRowFromChildTableForBoard" is sent. In case of the firmware branch the reply is the BOARD_FIRMWARE table.
- A loop over all entries of the table is done. If the entry is a FK and not empty, another simple query is sent to retrieve information to download the file.
- A pointer to the board that should be configured is used to call the method `loadJtagChainFromDb` that downloads the firmware file according to the information in the table and attempts to load the firmware file calling methods of the GT libraries of the specific board.

Downloading files over HTTP

Due to performance issues neither binary files nor ASCII files should be stored in the database. In case of the GT and GMT it was therefore decided to make files required for the configuration of the system available over HTTP. Trigger experts can upload files over a CERN AFS account and make entries in the database linking to them. It is planned to keep a local copy of the whole AFS repository to be able to work if AFS is not available. The class checks the local files of the AFS repository. If it does not find the requested file there it downloads the file to a temporary file and returns the filename that can now be used by other methods A.4.3.

Appendix A

Appendix

A.1 Global Trigger ConfDB Tables

The following naming conventions for tables and column names have been established:

- All table names as well as names of PK and FK columns contain only upper case letters. Names consisting of more words use underscores as word separating character.
- PK have the same name like the table with the suffix ”PK”.
- FK have the same name like the table they are referencing with the suffix ”FK”.

A.1.1 Main Table

Table Definition of the GT Main Table:

	NAME	TYPE	REFERENCED TABLE
TABLE	GT_CONFIG		
PRIMARY KEY	GT_CONFIG_PK	VARCHAR2(256)	
FOREIGN KEY	FDL_CONFIG_FK	VARCHAR2(256)	FDL_CONFIG
	GTL1_CONFIG_FK	VARCHAR2(256)	GTL_CONFIG
	GTL2_CONFIG_FK	VARCHAR2(256)	GTL_CONFIG
	TCS_CONFIG_FK	VARCHAR2(256)	TCS_CONFIG
	TIM_CONFIG_FK	VARCHAR2(256)	TIM_CONFIG
	PSB1_CONFIG_FK	VARCHAR2(256)	PSB_CONFIG
	PSB2_CONFIG_FK	VARCHAR2(256)	PSB_CONFIG
	PSB3_CONFIG_FK	VARCHAR2(256)	PSB_CONFIG
	PSB4_CONFIG_FK	VARCHAR2(256)	PSB_CONFIG
	PSB5_CONFIG_FK	VARCHAR2(256)	PSB_CONFIG
	PSB6_CONFIG_FK	VARCHAR2(256)	PSB_CONFIG
	PSBTT_CONFIG_FK	VARCHAR2(256)	PSB_CONFIG

A.1.2 Board Tables

Table Definition of the FDL Board Table:

	NAME	TYPE	REFERENCED TABLE
TABLE	FDL_CONFIG		
PRIMARY KEY	FDL_CONFIG_PK	VARCHAR2(256)	
FOREIGN KEY	FDL_FIRMWARE_FK	VARCHAR2(256)	FDL_FIRMWARE
	FDL_MEMORIES_FK	VARCHAR2(256)	FDL_MEMORIES
	FDL_REGISTERS_FK	VARCHAR2(256)	FDL_REGISTERS
	FDL_SEQUENCER_FILES_FK	VARCHAR2(256)	SEQUENCER_FILES

Table Definition of the GTFE Board Table:

	NAME	TYPE	REFERENCED TABLE
TABLE	GTFE_CONFIG		
PRIMARY KEY	GTFE_CONFIG_PK	VARCHAR2(256)	
FOREIGN KEY	GTFE_FIRMWARE_FK	VARCHAR2(256)	GTFE_FIRMWARE
	GTFE_MEMORIES_FK	VARCHAR2(256)	GTFE_MEMORIES
	GTFE_REGISTERS_FK	VARCHAR2(256)	GTFE_REGISTERS
	GTFE_SEQUENCER_FILES_FK	VARCHAR2(256)	SEQUENCER_FILES

Table Definition of the GTL Board Table:

	NAME	TYPE	REFERENCED TABLE
TABLE	GTL_CONFIG		
PRIMARY KEY	GTL_CONFIG_PK	VARCHAR2(256)	
FOREIGN KEY	GTL_FIRMWARE_FK	VARCHAR2(256)	GTL_FIRMWARE
	GTL_MEMORIES_FK	VARCHAR2(256)	GTL_MEMORIES
	GTL_REGISTERS_FK	VARCHAR2(256)	GTL_REGISTERS
	GTL_SEQUENCER_FILES_FK	VARCHAR2(256)	SEQUENCER_FILES

Table Definition of the TIM Board Table:

	NAME	TYPE	REFERENCED TABLE
TABLE	TIM_CONFIG		
PRIMARY KEY	TIM_CONFIG_PK	VARCHAR2(256)	
FOREIGN KEY	TIM_FIRMWARE_FK	VARCHAR2(256)	TIM_FIRMWARE
	TIM_MEMORIES_FK	VARCHAR2(256)	TIM_MEMORIES
	TIM_REGISTERS_FK	VARCHAR2(256)	TIM_REGISTERS
	TIM_SEQUENCER_FILES_FK	VARCHAR2(256)	SEQUENCER_FILES

Table Definition of the TCS Board Table:

	NAME	TYPE	REFERENCED TABLE
TABLE	TCS_CONFIG		
PRIMARY KEY	TCS_CONFIG_PK	VARCHAR2(256)	
FOREIGN KEY	TCS_MEMORIES_FK	VARCHAR2(256)	TCS_MEMORIES
	TCS_REGISTERS_FK	VARCHAR2(256)	TCS_REGISTERS
	TCS_SEQUENCER_FILES_FK	VARCHAR2(256)	SEQUENCER_FILES

A.1.3 Board Firmware Tables

Table Definition of the FDL Firmware Table:

	NAME	TYPE	REFERENCED TABLE
TABLE	FDL_FIRMWARE		
PRIMARY KEY	FDL_FIRMWARE_PK	VARCHAR2(256)	
FOREIGN KEY	ALTERA_FK	VARCHAR2(256)	FIRMWARE
	XILINX_FK	VARCHAR2(256)	FIRMWARE

Table Definition of the GTFE Firmware Table:

	NAME	TYPE	REFERENCED TABLE
TABLE	GTFE_FIRMWARE		
PRIMARY KEY	GTFE_FIRMWARE_PK	VARCHAR2(256)	
FOREIGN KEY	ALTERA_FK	VARCHAR2(256)	FIRMWARE
	XILINX_FK	VARCHAR2(256)	FIRMWARE

Table Definition of the GTL Firmware Table:

	NAME	TYPE	REFERENCED TABLE
TABLE	GTL_FIRMWARE		
PRIMARY KEY	GTL_FIRMWARE_PK	VARCHAR2(256)	
FOREIGN KEY	ALTERA_FK	VARCHAR2(256)	FIRMWARE
	XILINX_FK	VARCHAR2(256)	FIRMWARE

Table Definition of the PSB Firmware Table:

	NAME	TYPE	REFERENCED TABLE
TABLE	PSB_FIRMWARE		
PRIMARY KEY	PSB_FIRMWARE_PK	VARCHAR2(256)	
FOREIGN KEY	ALTERA_FK	VARCHAR2(256)	FIRMWARE
	XILINX_FK	VARCHAR2(256)	FIRMWARE

Table Definition of the TIM Firmware Table:

	NAME	TYPE	REFERENCED TABLE
TABLE	TIM_FIRMWARE		
PRIMARY KEY	TIM_FIRMWARE_PK	VARCHAR2(256)	
FOREIGN KEY	ALTERA_FK	VARCHAR2(256)	FIRMWARE
	XILINX_FK	VARCHAR2(256)	FIRMWARE

A.1.4 Board Memories Tables

There is a memory table for each type of GT module, containing one Primary Key column of the type VARCHAR(256) and a Foreign Key (FK) column for every memory on the board referencing the MEMORIES Table. The type of FK columns is VARCHAR(256) too. For reasons of readability for Board Memories Tables containing many FKs, only a list of the FK column names is given.

Table Definition of the GTFE Memories Table:

	NAME	TYPE	REFERENCED TABLE
TABLE	GTFE_MEMORIES		
PRIMARY KEY	GTFE_MEMORIES_PK	VARCHAR2(256)	
FOREIGN KEY	SIM_SPY_MEMORY_DAQ_FK	VARCHAR2(256)	MEMORIES

Table Definition of the TIM Memories Table:

	NAME	TYPE	REFERENCED TABLE
TABLE	TIM_MEMORIES		
PRIMARY KEY	TIM_MEMORIES_PK	VARCHAR2(256)	
FOREIGN KEY	BC_TABLE_FK	VARCHAR2(256)	MEMORIES

FDL Memories Table Foreign Key Columns:

TTRIG_MEM_b31_b0_FK	TTRIG_MEM_b63_b32_FK	ALGO_MEM_b31_b0_FK
ALGO_MEM_b63_b32_FK	ALGO_MEM_b95_b64_FK	ALGO_MEM_b127_b96_FK
ALGO_MEM_b159_b128_FK	ALGO_MEM_b191_b160_FK	

PSB Memories Table Foreign Key Columns:

SIM_SPY_MEM0_FK	SIM_SPY_MEM1_FK	SIM_SPY_MEM2_FK
SIM_SPY_MEM3_FK	SIM_SPY_MEM4_FK	SIM_SPY_MEM5_FK
SIM_SPY_MEM6_FK	SIM_SPY_MEM7_FK	REFERENCE_MEM_FK
SIM_SPY_MEM8_FK		

GTL Memories Table Foreign Key Columns:

SIM_SPY_MU12_L_FK	SIM_SPY_MU12_H_FK	SIM_SPY_MU34_L_FK
SIM_SPY_MU34_H_FK	SIM_SPY_CA1_13_FK	SIM_SPY_CA1_24_FK
SIM_SPY_CA2_13_FK	SIM_SPY_CA2_24_FK	SIM_SPY_CA3_13_FK
SIM_SPY_CA3_24_FK	SIM_SPY_CA4_13_FK	SIM_SPY_CA4_24_FK
SIM_SPY_CA5_13_FK	SIM_SPY_CA5_24_FK	SIM_SPY_CA6_13_FK
SIM_SPY_CA6_24_FK	SIM_SPY_CA7_13_FK	SIM_SPY_CA7_24_FK
SIM_SPY_CA8_13_FK	SIM_SPY_CA8_24_FK	SIM_SPY_CA9_13_FK
SIM_SPY_CA9_24_FK	SIM_SPY_CA10_13_FK	SIM_SPY_CA10_24_FK
DPRAM_b15_b0_COND1_FK	DPRAM_b31_b16_COND1_FK	DPRAM_b47_b32_COND1_FK
DPRAM_b63_b48_COND1_FK	DPRAM_b79_b64_COND1_FK	DPRAM_b95_b80_COND1_FK
DPRAM_b15_b0_COND2_FK	DPRAM_b31_b16_COND2_FK	DPRAM_b47_b32_COND2_FK
DPRAM_b63_b48_COND2_FK	DPRAM_b79_b64_COND2_FK	DPRAM_b95_b80_COND2_FK

TCS Memories Table Foreign Key Columns:

DAQ0_BC_TABLE0_FK	DAQ0_BC_TABLE1_FK	DAQ1_BC_TABLE0_FK
DAQ1_BC_TABLE1_FK	DAQ2_BC_TABLE0_FK	DAQ2_BC_TABLE1_FK
DAQ3_BC_TABLE0_FK	DAQ3_BC_TABLE1_FK	DAQ4_BC_TABLE0_FK
DAQ4_BC_TABLE1_FK	DAQ5_BC_TABLE0_FK	DAQ5_BC_TABLE1_FK
DAQ6_BC_TABLE0_FK	DAQ6_BC_TABLE1_FK	DAQ7_BC_TABLE0_FK
DAQ7_BC_TABLE1_FK		

A.1.5 Board Register Tables

Board Register Tables contain follow the same naming conventions as the other tables. The table names are composed of a board-type prefix followed by "REGISTERS" (FDL_REGISTERS, GTFE_REGISTERS, ...). Data types used for the columns are VARCHAR2(256) for key columns and NUMBER(5) for 16 bit registers and NUMBER(10) for 32 bit registers currently only used on the FDL board. The column names of columns containing register data matches the item names of registers in the HAL AddressTable file. Below register column names for every board are listed.

FDL Register Column Names:

SLICE_REG	UPDATE_PERIODE_REG	UPDATE_STEP_SIZE_REG
GENERAL_REG	IDLE_CODE_REG	BOARD_ID_LATENCY_DEL_REG
ORBIT_LENGTH	NO_ALGO_PRESCALER_REG	NO_ALGO_SETUP_REG

GTFE Register Column Names:

CMD_REG TTC_CTRL	EN_CHLINK BCRES_DEL_REG	DAQ_CMD_REG
MAX_BC_NUMBER INITIAL_CRC	CMS_HEADER_19_4	BOARD_ID
SETUP_VERS_15_0	SETUP_VERS_31_16	SPY_FULL_LIMIT
TEST_MASK1	TEST_MASK2	TEST_MASK3
SEL_PHASE	IGNORE_ERR_FOR_TCS	EVM_BCRES_DEL_REG
EVM_CMD_REG	EVM_MAX_BC_NUMBER	EVM_INITIAL_CRC
EVM_CMS_HEADER_19_4	EVM_BOARD_ID	EVM_SETUP_VERS_15_0
EVM_SETUP_VERS_31_16	EVM_SPY_FULL_LIMIT	EVM_TEST_MASK1
EVM_TEST_MASK2	EVM_TEST_MASK3	EVM_IGNORE_ERR_FOR_TCS
BST_UPDATE_DELAY		

GTL Register Column Names:

SEL_REG_COND1	SEL_REG_COND2	BCRES_DEL_REG_REC1
MAX_BC_NUM_REC1	CMD_REG_REC1	BCRES_DEL_REG_REC2
MAX_BC_NUM_REC2	CMD_REG_REC2	BCRES_DEL_REG_REC3
MAX_BC_NUM_REC3	CMD_REG_REC3	BX_NR_REG_LOW_COND1
BX_NR_REG_HIGH_COND1	BCNTRES_DEL_REG_COND1	MODE_REG_COND1
BX_NR_REG_LOW_COND2	BX_NR_REG_HIGH_COND2	BCNTRES_DEL_REG_COND2
MODE_REG_COND2	CMD_ENPROG	CMD_NPROG
CMD_INIT	CMD_REG	

PSB Register Column Names:

CMD_ENPROG	CMD_NPROG	CMD_INIT
CMD SERLINK0	SERLINK1	SERLINK2
EN_TTIN	CHAN_REG0	CHAN_REG1
CHAN_REG2	CHAN_REG3	CHAN_REG4
CHAN_REG5	CHAN_REG6	CHAN_REG7
CHAN_DELAY0	CHAN_DELAY1	CHAN_DELAY2
CHAN_DELAY3	CHAN_DELAY4	CHAN_DELAY5
CHAN_DELAY6	CHAN_DELAY7	LVDS_DELAY0
LVDS_DELAY1	LVDS_DELAY2	LVDS_DELAY3
LVDS_DELAY4	LVDS_DELAY5	LVDS_DELAY6
LVDS_DELAY7	LVDS_DELAY8	LVDS_DELAY9
LVDS_DELAY10	LVDS_DELAY11	LVDS_DELAY12
LVDS_DELAY13	LVDS_DELAY14	LVDS_DELAY15
BOARD_ID	BCRES_DELAY	LATENCY_DELAY
ROP_SETUP	MAX_BC_NUMBER	SEL_PHASE3100
SEL_PHASE6332	IDLE_ID_LOW	IDLE_ID_HIGH
COMP_DLY0	COMP_DLY1	COMP_DLY2
COMP_DLY3	COMP_DLY4	COMP_DLY5
COMP_DLY6	COMP_DLY7	REF_REG

TCS Register Column Names:

DAQ0_CMD_REG	DAQ0_RANDOM_FREQ	DAQ0_RANDOM_START_VALUE
DAQ0_TRIG_TYPE_A	DAQ0_TRIG_TYPE_B	DAQ0_BGO_PERIOD_L
DAQ0_BGO_PERIOD_S	DAQ0_ACTIVE_TIME	DAQ2_RANDOM_START_VALUE
DAQ1_RANDOM_FREQ	DAQ1_TRIG_TYPE_A	DAQ1_RANDOM_START_VALUE
DAQ1_TRIG_TYPE_B	DAQ1_BGO_PERIOD_L	DAQ1_BGO_PERIOD_S
DAQ1_ACTIVE_TIME	DAQ2_CMD_REG	DAQ2_RANDOM_FREQ
DAQ1_CMD_REG	DAQ2_TRIG_TYPE_A	DAQ2_TRIG_TYPE_B
DAQ2_BGO_PERIOD_L	DAQ2_BGO_PERIOD_S	DAQ2_ACTIVE_TIME
DAQ3_CMD_REG	DAQ3_RANDOM_FREQ	DAQ3_RANDOM_START_VALUE
DAQ3_TRIG_TYPE_A	DAQ3_TRIG_TYPE_B	DAQ3_BGO_PERIOD_L
DAQ3_BGO_PERIOD_S	DAQ3_ACTIVE_TIME	DAQ5_RANDOM_START_VALUE
DAQ4_RANDOM_FREQ	DAQ4_TRIG_TYPE_A	DAQ4_RANDOM_START_VALUE
DAQ4_TRIG_TYPE_B	DAQ4_BGO_PERIOD_L	DAQ4_BGO_PERIOD_S
DAQ4_ACTIVE_TIME	DAQ5_CMD_REG	DAQ5_RANDOM_FREQ
DAQ4_CMD_REG	DAQ5_TRIG_TYPE_A	DAQ5_TRIG_TYPE_B
DAQ5_BGO_PERIOD_L	DAQ5_BGO_PERIOD_S	DAQ5_ACTIVE_TIME
DAQ6_CMD_REG	DAQ6_RANDOM_FREQ	DAQ6_RANDOM_START_VALUE
DAQ6_TRIG_TYPE_A	DAQ6_TRIG_TYPE_B	DAQ6_BGO_PERIOD_L
DAQ6_BGO_PERIOD_S	DAQ6_ACTIVE_TIME	DAQ7_CMD_REG
DAQ7_RANDOM_FREQ	DAQ7_TRIG_TYPE_A	DAQ7_RANDOM_START_VALUE
DAQ7_TRIG_TYPE_B	DAQ7_BGO_PERIOD_L	DAQ7_BGO_PERIOD_S
DAQ7_ACTIVE_TIME	RULE4_DLY	RULE4_LIM
RULE4_LOW_LIM	RULE3_DLYH	RULE3_DLYL
RULE3_LIM	RULE2_DLY	RULE1_2
TIMESLOT0	TIMESLOT1	TIMESLOT2
TIMESLOT3	TIMESLOT4	TIMESLOT5
TIMESLOT6	TIMESLOT7	ASSIGN_PART3_0
ASSIGN_PART7_4	ASSIGN_PART11_8	ASSIGN_PART15_12
ASSIGN_PART19_16	ASSIGN_PART23_20	ASSIGN_PART27_24
ASSIGN_PART31_28	ASSIGN_LUM_GT_ENIO	ORBIT_LENGTH_1
GAP_LIMITH	GAP_LIMITL	SETTLE_TIME
RECOVER_TIME	MON_CNTR_PERIOD	GT_STATUS
TCSM_CMD_REG	SIM_GT_STATUS	SIM_PARTS_H
SIM_PARTS_L	V_STATUS2320	V_STATUS1916
V_STATUS1512	V_STATUS1108	V_STATUS0704
V_STATUS0300	V_DAQ_STATUS0704	V_DAQ_STATUS0300

A.1.6 Firmware Table

The Firmware table contains a link to a firmware file in the AFS repository. In order to allow the Configuration Operation to check if the loaded firmware is older than the requested version information about the date, version and id are provided as well. The number of the JTAG chain is needed to identify the JTAG chain on the module that should be loaded. Finally the column OFFLINE_INFO_LINK may be used to reference a file that contains information corresponding to the a firmware version (e.g. a certain trigger menu) that can be needed for off line analysis.

Table Definition of the Firmware Table:

	NAME	TYPE
TABLE	FIRMWARE	
PRIMARY KEY	FIRMWARE_PK	VARCHAR2(256)
DATA COLUMNS	BASE_URL	VARCHAR2(512)
	RELATIVE_URL	VARCHAR2(512)
	FW_ID	NUMBER(10)
	FW_REVISION	NUMBER(10)
	FW_DATE	DATE
	JTAG_CHAIN	NUMBER(1)
	OFFLINE_INFO_LINK	VARCHAR2(512)

A.1.7 Memories Table

The Memories table contains a link to a file in the AFS repository. Optionally a description can be added to each referenced file.

Table Definition of the Memories Table:

	NAME	TYPE
TABLE	MEMORIES	
PRIMARY KEY	MEMORIES_PK	VARCHAR2(256)
DATA COLUMNS	BASE_URL	VARCHAR2(512)
	RELATIVE_URL	VARCHAR2(512)
	DESCRIPTION	VARCHAR2(512)

A.1.8 Sequencer File Table

HAL Sequencer files are referenced from this table.

Table Definition of the Sequencer File Table:

	NAME	TYPE
TABLE	SEQUENCER_FILES	
PRIMARY KEY	SEQUENCER_FILES_PK	VARCHAR2(256)
DATA COLUMNS	BASE_URL	VARCHAR2(512)
	RELATIVE_URL	VARCHAR2(512)
	DESCRIPTION	VARCHAR2(512)

A.2 Global Trigger AFS repository

The Global Trigger group has agreed to store files related to the GT hardware like firmware files, trigger menu files or memory files should be stored in a place, where they would be accessible from Vienna as well as from CERN. Therefore an AFS repository accessible over HTTP was created to store those files. Figure A.1 shows the directory structure of the repository.

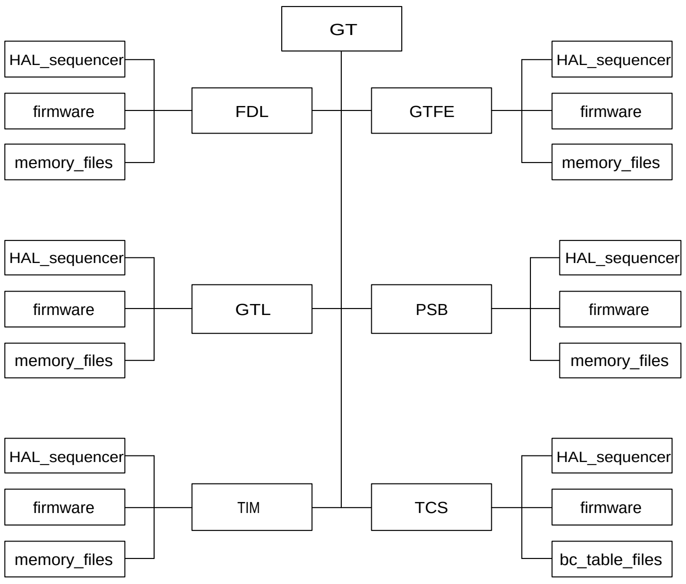


Figure A.1: The directory structure of the GT AFS repository.

A.3 File Formats

A.3.1 XhannelListFile

Listing A.1: An Example of a Xhannel List File defining a Xhannel to TStore with the name ”DatabaseXhannel”.

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

<cell_xhannel_list
xmlns=
"http://triggersupervisor.cern.ch/cell_xhannel_list"
xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation=
"http://triggersupervisor.cern.ch/
        cell_xhannel_list/cell_xhannel_list.xsd">

<xhannel>
        <name>DatabaseXhannel</name>
        <type>tstore</type>
        <target instance="0"/>
</xhannel>

</cell_xhannel_list>
```

A.3.2 GT Tstore Views

Listing A.2: Queries needed for the GT Configuration Operation.

```
<tstore:configuration
        xmlns:tstore="urn:xdaq-tstore:1.0"
        xmlns:sql="urn:tstore-view-SQL" >

<tstore:view
        id="urn:tstore-view-SQL:gtTstoreView"
        name="urn:tstore-view-SQL:gtTstoreView">

<tstore:connection dbname="omds" username="globaltrigger"/>

<sql:query name="getRowFromTable">

        <sql:parameter name="TableNam">
                <![CDATA[]]>
        </sql:parameter>

        <sql:parameter name="TablePkNam">
                <![CDATA[]]>
        </sql:parameter>

        <sql:parameter name="Pk" bind="yes">
                <![CDATA[]]>
        </sql:parameter>
        <![CDATA[select * from $TableNam where $TablePkNam = $Pk]]>
</sql:query>

<sql:query name="getRowFromChildTableForBoard">
        <sql:parameter name="BoardTableNam">
                <![CDATA[]]>
        </sql:parameter>

        <sql:parameter name="BoardTablePkNam">
                <![CDATA[]]>
        </sql:parameter>

        <sql:parameter name="BoardTablePk" bind="yes">
                <![CDATA[]]>
        </sql:parameter>

        <sql:parameter name="ChildTableNam">
                <![CDATA[]]>
        </sql:parameter>

        <sql:parameter name="ChildTableFkNam">
                <![CDATA[]]>
        </sql:parameter>

        <sql:parameter name="ChildTablePkNam">
                <![CDATA[]]>
        </sql:parameter>
        <![CDATA[
        select * from $ChildTableNam where $ChildTablePkNam=(
        select $ChildTableFkNam from $BoardTableNam
        where $BoardTablePkNam = $BoardTablePk )]]>
</sql:query>
</tstore:view>
</tstore:configuration>
```

A.3.3 Bunch Crossing Tables

Bunch Crossing (BC) Tables are actually memories implemented on the TCS chip that contain information for the PTC for a whole orbit. There are two BC-Tables per PTC the exact functionality of which can be found in the TCS manual. Due to their length of 4k it was decided not to load BC-Tables directly from the database, but rather store a link to a file in the GT AFS repository. The files used are ASCII files with a rather simple syntax:

- Comments are indicated by the '#' sign at the beginning of a line.
- The first column of the file indicates the number of the Bunch Crossing.
- The second column of the file gives the value for the according Bunch Crossing.

Listing A.3: Part of a BC-Table file.

```
#######
#Lines starting with the number sign
#are interpreted as comments
#######

0000 0000d00d
0001 0000ffff
0002 00001f3f
0003 00000000
0004 00000000
0005 00005667
```

A.3.4 Memory Files

Memory Files can be used for any kind of memory on a GT module and follow exactly the same syntax as BC-Table files. A.3. The method `loadMemoryFromFile` in the `GtBoard` class accepts the full path (including the filename) to a file following this syntax and the name of the item associated with the start address of the memory in the HAL AddressTableFile for the board and tries to load the memory with data from the file. This feature is made use of in the Configuration Operation of the GT Cell.

A.4 Code Snippets

A.4.1 Retrieving the GT_CONFIG table

Listing A.4: Querying for a row of the main table.

```
xdata::Table* GtCommonConfiguration::getGtConfigTableRowFromKey(
        CellXhannelTB* pXhannel,
        std::string key ) throw(CellException) {
try {
pXhannel->connect("urn:tstore-view-SQL:gtTstoreView","passWord");
}
catch (xcept::Exception& e) {
ostringstream msg;
msg << "Errormessage" << endl;
getWarning()->setLevel(CellWarning::ERROR);
getWarning()->setMessage(msg.str());
XCEPT_RAISE(CellException,msg.str());
return 0;
}

CellXhannelRequestTB* req =
  dynamic_cast<CellXhannelRequestTB*>(pXhannel->createRequest());

map<string,string> parameters;
parameters["TableNam"] = "GT_CONFIG";
parameters["TablePkNam"] = "GT_CONFIG_PK";
parameters["Pk"] = key;

try {
req->doGetQuery("getRowFrowTable", parameters);
pXhannel->send(req);
}
catch (xcept::Exception& e) {
   ...
}

xdata::Table *pGtConfigTable;

try {
        pGtConfigTable= req->getQueryReply();
  pXhannel->removeRequest(req);
}
catch(xcept::Exception& e) {
   ...
}
pXhannel->disconnect();
return pGtConfigTable;
}
```

A.4.2 Firmware Configuration

The code below is simplified for better readability. Try-catch blocks and error messages are omitted and names of variables are shortened.

Listing A.5: Firmware branch of the Configuration Operation.

```
void GtCommonConfiguration::configureBoardFirmware(
        CellXhannelTB* pXhannelTB,
        string serialNum,
        string colValue)
        throw (CellException) {

pXhannelTB->connect("urn:tstore-view-SQL:gtTstoreView","xray2000");
CellXhannelRequestTB* req =
 dynamic_cast<CellXhannelRequestTB*>(pXhannelTB->createRequest());

map<string,string> parameters =
 getParamMapForQueryNest("FIRMWARE", serialNum, colValue);

req->doGetQuery("getChildTableRowForBoard", parameters);
pXhannelTB->send(req);

xdata::Table *pBoardFwTable;
pBoardFwTable= req->getQueryReply();
pXhannelTB->removeRequest(req);
pXhannelTB->disconnect();

vector<string> cl = pBoardFwTable->getColumns();
for(vector<string>::iterator it(cl.begin());it!=cl.end();++it) {

string colNam = *it;
colVal = pBoardFwTable->getValueAt(0, *it)->toString();
if (colNam.find( "_PK", 0 )!=string::npos)
{//primary key column}
else if (colNam.find( "_FK", 0 )==string::npos)
{//no foreign key found}
else if ( colVal == "" )
{//empty foreign key}
else {
pXhannelTB->connect("urn:tstore-view-SQL:gtTstoreView","xray2000");
req =
 dynamic_cast<CellXhannelRequestTB*>(pXhannelTB->createRequest());

parameters=getParamMapForQuerySmpl("Firmware", key)

req->doGetQuery("getTable", parameters);
pXhannelTB->send(req);

xdata::Table *pFwTable;
pFwTable= req->getQueryReply();
pXhannelTB->removeRequest(req);
pXhannelTB->disconnect();


CellGTCrate& gtCrate=
        *(dynamic_cast<CellContext*>(context_)->getGtCrate());
GtBoard *pGtBoard=
        gtCrate.getBoardFromSerialNumber(serialNum);

pGtBoard->loadJtagChainFromDb( pFirmwareTable);
}}}
```

A.4.3 Using the GtHttpDownloader

Listing A.6: Downloading a file over HTTP.

```
string baseUrl =
 "http://cms-gmt-afs.web.cern.ch/cms-gt-afs";
string relativeUrl =
 "GT/GTL/firmware/cond1.svf";
string mirrorUrl =
 "/opt/l1global_afs_mirror/";

GtHttpDownloader d;
string fwFilename;

baseUrl += relativeUrl;
mirrorUrl += relativeUrl;
fwFilename = d.download2fileMirrored( baseUrl, mirrorUrl);

pGtBoard->loadMemoryFromFile( fwFilename, itemName );
```

Bibliography

[1] LHC Project Homepage, LHC - The Large Hadron Collider, `http://lhc.web.cern.ch/lhc/`

[2] C. Rubbia, A few considerations of strategy on the future of CERN, Proc. of the Large Hadron Collider Workshop, Aachen, Germany, Oct. 1990, **CERN-90-10-V-1**

[3] ALICE Collaboration, ALICE Technical Proposal, **CERN/LHCC 1995-71** (1995)

[4] ATLAS Collaboration, ATLAS Technical Proposal, **CERN/LHCC 1994-43** (1994)

[5] LHCb Collaboration, LHCb Technical Proposal, **CERN/LHCC 1998-4** (1998)

[6] CMS Collarboration, The Compact Muon Solenoid - Technical proposal, **CERN/LHCC 1994-38** (1994)

[7] CMS Collarboration, The Tracker Project – Technical Design Report, **CERN/LHCC 1998-6** (1998)

[8] CMS Collaboration, Addendum to the CMS Tracker Technical Design Report, **CERN/LHCC 2000-16** (2000)

[9] CMS Collaboration, The Electromagnetic Calorimeter Technical Design Report, **CERN/LHCC 97-33** (1997)

[10] CMS Collaboration, The Hadron Calorimeter, Technical Design Report, **CERN/LHCC 97-31** (1997)

[11] CMS Collaboration, The Magnet Project, Technical Design Report, **CERN/LHCC 97-10** (1997)

[12] CMS Collaboration, The Muon Project, Technical Design Report, **CERN/LHCC 97-32** (1997)

[13] Hannes Sakulin, Design and Simulation of the First Level Global Muon Trigger for the CMS Experiment at CERN, Dissertation (2002)

[14] CMS Collaboration, The TriDAS Project – The Level-1 Trigger Technical Design Report, **CERN/LHCC 2000-38** (2000)

[15] CMS Collaboration, The TriDAS Project – Data Acquisition and High-Level Trigger Technical Design Report, **CERN/LHCC 2002-26** (2002)

[16] R. Martinelli et al., Design of the Track Correlator for the DTBX Trigger, **CERN CMS Note 1999/007** (1999)

[17] A. Kluge, T. Wildschek, The Hardware Muon Trigger Track Finder Processor in CMS - Specification and Method, **CERN CMS Note 1997/091** (1997)

[18] A. Kluge, T. Wildschek, The Hardware Muon Trigger Track Finder Processor in CMS - Architecture and Algorithm, **CERN CMS Note 1997/092** (1997)

[19] A. Kluge, T.Wildschek, The HardwareMuon Trigger Track Finder Processor in CMS - Prototype and Final Implementation, **CERN CMS Note 1997/093** (1997)

[20] J. Ero, New Approach for the CMS Muon Trigger Track Finder Processor, Proc. of the Fifth Workshop on Electronics for LHC Experiments, Snowmass, Co, USA, Sept. 1999, **CERN/LHCC/99-33** (1999)

[21] D. Acosta et al., The Track-Finder Processor for the Level-1 Trigger of the CMS Endcap Muon System, Proc. of the Fifth Workshop on Electronics for LHC Experiments, Snowmass, Co, USA, Sept. 1999, **CERN/LHCC/99-33** (1999)

[22] TOTEM Collaboration, TOTEM, Technical Design Report, **CERN/LHCC 2004-002** (2004)

[23] V. Brigljevic et al., Run Control and Monitor System for the CMS Experiment, Computing in High Energy and Nuclear Physics, 2003, La Jolla, California

[24] C.-E. Wulz: Concept of the CMS First Level Global Trigger for the CMS Experiment at LHC, Nucl. Instr. and Meth. **A 473** (2001) 231

[25] A. Taurok, H. Bergauer, M. Padrta, Implementation and Synchronisation of the CMS First Level Global Trigger for the CMS Experiment at LHC, Nucl. Instr. and Meth. **A 473** (2001) 243

[26] M.Jeitler et al., The Level-1 Global Trigger for the CMS Experiment at LHC, Journal of Instrumentation (2007) P01006, `http://stacks.iop.org/1748-0221/2/P01006`

[27] Global Trigger Homepage, `http://wwwhephy.oeaw.ac.at/p3w/electronic1/GlobalTrigger/GlobalTriggerCrate.htm`

[28] Ph. Glaser, T. Noebauer et al., Design and Development of a Graphical Setup Software for the CMS Global Trigger, IEEE Trans.Nucl.Sci. Vol 53 Nr. 3 (2006) 1282-1291

[29] J. Varela, Integration of Run Control and Detector Control Systems, **CERN CMS Internal Note 2005/015** (2005)

[30] I. Magrans, C.-E. Wulz, J. Varela, Concept of the CMS Trigger Supervisor, **CERN CMS Note 2005/011** (2005)

[31] J. Gutleber and L. Orsini, Software Architecture for Processing Clusters Based on I2O, Cluster Computing 5/1 2002 55-64, (2002)

[32] Simple Object Access Protocol (SOAP) 1.1, `http://www.w3.org/TR/soap`

[33] J. Varela, CMS L1 Trigger Control System, **CERN CMS Note 2002/033** (2002)

[34] Very High Speed Integrated Circuit Hardware Description Language (VHDL), Standard IEEE-1076 (1993)

[35] Extensible Markup Language (XML) - `http://www.w3.org/XML/`

[36] HyperText Markup Language (HTML) Home Page, `http://www.w3.org/MarkUp/`

[37] X-DAQ Wiki - SOAP messaging, `http://xdaqwiki.cern.ch/index.php/SOAP_Messaging`

[38] X-DAQ Wiki - TStore, `http://xdaqwiki.cern.ch/index.php/TStore`

[39] CURL - groks those URLs, `http://curl.haxx.se/`

[40] GNU cgicc - a C++ class library for writing CGI applications, `http://www.gnu.org/software/cgicc/cgicc.html`

[41] Trigger Supervisor Homepage, `http://triggersupervisor.cern.ch/`

[42] Trigger Supervisor - User's Guide, `http://triggersupervisor.cern.ch/index.php?option=com_docman&task=doc_download&gid=32`

[43] Trigger Supervisor framework Documentation, `http://triggersupervisor.cern.ch/uploads/api_docs/ts/html/`

[44] AjaXell sourceforge project, `http://sourceforge.net/projects/ajaxell/`

[45] HAL (Hardware Access Library), `http://cmsdoc.cern.ch/~cschwick/software/documentation/HAL/index.html`

[46] JAL (JTAG Access Library), `http://cmsdoc.cern.ch/~hsakulin/jal/`

[47] HTTP - Hypertext Transfer Protocol, `http://www.w3.org/Protocols/`