The FEE Server Control Engine of the ALICE-TRD

Diplomarbeit

von

Uwe Westerhoff

Westfälische Wilhelms-Universität Münster

Institut für Kernphysik

— Januar 2009 —

Contents

List of abbreviations

ADC **A**naloge to **D**igital **C**onverter.

ALICE **A** **L**arge **I**on **C**ollider **E**xperiment.

DAQ **D**ata **A**quisition.

DCS **D**etector **C**ontrol **S**ystem.

DIM **D**istributed **I**nformation **M**anagement System.
Communication system for ECS/DCS.

ECS **E**xperiment **C**ontrol **S**ystem.

FERO **F**ront **E**nd **R**ead**o**ut **E**lectronic. Collective name for all electronics directly mounted on the top of a readout chamber.

FSM **F**inite **S**tate **M**achine.

GTU **G**lobal **T**racking **U**nit. Fast track finder of the TRD.

LHC **L**arge **H**adron **C**ollider. The largest collider woldwide. It is located in Swizerland.

MCM **M**ulti **C**hip **M**odule. Part of the front end readout electronics of the TRD.

ORI **O**ptical **R**eadout **I**nterface. Part of the front end readout electronics. Interface card to transfer data from the TRD to the DAQ system.

PASA **P**re**a**mplifier **Sh**a**per. One of the two chips in a MCM.

ROB **R**ead**o**ut **B**oard. A board directly mounted on the readout chambers.

ROC **R**ead**o**ut **C**hamber. Smallest independent detector part of the TRD.

TPP **T**racklet **P**re**p**rocessor. Part of each MCM. Looks for tracklet candidates.

TRAP One of the two chips in a MCM.

TRD **T**ransition **R**adiation **D**etector. One subdetector of the ALICE experiment.

TTC **T**iming, **T**rigger and **C**ontrol. The trigger system of ALICE.

1 Introduction

The standard model of elementary particles describes the particles we observe in the universe and the ways these particles interact.[1] According to the standard model all observed hadrons like protons or pions are composed of gluons and two or three quarks/anti-quarks. Quantum chromodynamics (QCD) describes the interactions between these particles.

In normal baryonic matter each quark, anti-quark or gluon is assigned to one hadron. Free (anti-)quarks or gluons have never been observed. This behavior is called confinement. Another important property of the QCD is asymptotic freedom. At high energies, which is equivalent to small distances, the coupling strength between quarks and gluons becomes weak.

In the very early universe, up to about 10 µs after the Big Bang the temperature was above a critical temperature $T_c = 150 - 190\,\text{MeV}$ [BMS07]. At these temperatures quarks and gluons are not confined to hadrons but can move freely. This state is called a quark-gluon plasma (QGP). When the universe expanded, it cooled down and the quarks and gluons were confined inside hadrons. In the present-day universe a QGP may only exist inside of neutron stars if the densities are high enough.

In a laboratory the QGP can be created by colliding two nuclei at ultra-relativistic center-of-mass energies. For a short time the energy density in the collision zone is high enough to create a QGP. Measuring the properties of the QGP and investigating the phase transition from a QGP to normal baryonic matter is very important. Such measurements can be used to check QCD calculations and they increase our knowledge about the history of the universe. The Super Proton Synchrotron (SPS) at CERN was the first accelerator which saw hints for the creation of a QGP [CER00]. In 2009 the new Large Hadron Collider (LHC) at CERN near Geneva will start its regular operation. It is capable to collide heavy ions at center-of-mass energies up to $\sqrt{s_{NN}} = 5.5\,\text{TeV}$ which is about 30 times more then RHIC[2] energies [BMS07]. However, most of the time the LHC will run proton-proton collisions mainly to look for the Higgs boson which is the last undetected particle of the standard model and search for physics beyond the standard model. ALICE as one of the four experiments at LHC is the only dedicated heavy ion experiment.

In this diploma thesis the control engine for the transition radiation detector (TRD) of ALICE has been redesigned, implemented and tested. Based on this software a procedure was developed to check proper cooling of the front end electronics of the TRD.

[1]Nowadays it is known that the standard model cannot be complete because it fails describing some important observations like the nature of dark matter.

[2]The Relativistic Heavy Ion Collider (RHIC) located at the Brookhaven National Laboratory at Long Island (USA) was the most powerful heavy ion collider before LHC was build

2 Theoretical background

2.1 The Standard Model

Until the second half of the twentieth century the number of known 'elementary' particles was very manageable. The electron was discovered by Thomson already in 1897. Rutherford discovered the proton in 1918 by shooting α particles in a nitrogen gas and analysing the resulting radiation. In 1930 the existence of the neutrino was postulated by Pauli to explain the beta decay. Two years later Chadwick verified the existence of another new particle, the neutron. Hence four particles (electron, proton, neutrino and neutron) were known which where sufficient to explain all known observations in particle physics at this time. The situation changed in the fifties and sixties of the twentieth century. First the muon was identified in 1947/48. In the following years lots of other 'elementary' particles were identified but an overall theory which described the particles and their properties was missing.

The development of such a theory took some time. In 1964, M. Gell-Mann and G. Zweig suggested the quark modell for the hadrons. According to this model most of the former 'elementary' particles were not elementary but composed of smaller particles, called quarks. In 1972, M. Gell-Mann and H. Fritzsch introduced color as a new quantum number. Both concepts became an important part of today's Standard Model of elementary particles.

Today, six leptons and six quarks and their anti-particles are known (table 2.1). The forces between particles are mediated by gauge bosons (see table 2.2). The mediating particle(s) of graviation is/are not discovered yet. All other particles, like protons or neutrons, are no fundermental particles. They consist of two or three (anti-)quarks and gluons which mediate the strong nuclear force between the quarks. The theory that describes the strong nuclear force is Quantum chromodynamics (QCD). It has a similar structure to Quantum electrodynamics which describes the interaction between charged particles. QED has the uncharged photon as the single gauge boson which mediates the electromagnetic force between positively or negatively charged particles. In contrast, QCD has eight gauge bosons named gluons which mediate the strong interaction between particles with color charge. The color charges in the QCD are green, red, blue and the anti-colors anti-green, anti-red and anti-blue. The quarks always carry one color or anti-color charge, whereas the gluons carry color and anti-color.

Free particles with a net-color charge have never been observed. This behavior is called confinement. Baryons like the proton or neutron consist of three quarks which all have a different color charge. Adding the three color charges result in 'white' which corresponds to a colorless particle. Mesons consist of a quark and an anti-quark. In mesons the anti-quark always has the corresponding anti-color of the quark, resulting in a colorless particle, too.

	first generation	second generation	third generation
Quarks	up (u)	charm (c)	top (t)
	down (d)	strange (s)	bottom (b)
Leptons	electron neutrino ν_e	muon neutrino ν_μ	tau neutrino ν_τ
	electron (e)	muon (μ)	tau (τ)

Table 2.1: Overview of the fundermental particles. Each of the particles have an anti-particle, too (not shown).

Interaction	Rel. Strength	Range	Mediators
Electromagnetic Interaction	$\alpha = \frac{1}{137}$	∞	Photon γ, spin: 1, mass $m_\gamma = 0$, interacts with electrically charged particles
Strong Interaction	$\alpha_s = \mathrm{O}(1)$ (depends on momentum transfer)	$\approx 1\,\mathrm{fm}$	8 gluons g_i, spin: 1, mass m_g=0; interact with the color charge of quarks; gluons are color charged, too $\rightarrow$ self interaction between the gluons
Weak Interaction	$\approx \alpha \cdot \frac{q^2}{m_W^2 c^2}$ (depends on momentum transfer q)	$\ll 1\,\mathrm{fm}$	2 charged bosons $(W^-; W^+)$ and one neutral boson (Z^0); spin: 1, $m_W = 80\,\mathrm{GeV}/c^2$; $m_Z = 91\,\mathrm{GeV}/c^2$; interact with all (anti-)leptons $(e^-, \mu^-, \tau^-, \nu_e, \nu_\mu, \nu_\tau)$ and all (anti-)quarks $(\mathrm{u}, \mathrm{d}, \mathrm{c}, \mathrm{s}, \mathrm{t}, \mathrm{b})$
Gravitation	$\alpha \cdot 10^{-36}$	∞	interacts with all particles

Table 2.2: Overview of the four fundamental interactions.

The potential between two quarks reflects the confinement. It can be approximated by [Sat90]

$$V_s(r) = -\frac{4}{3}\frac{\alpha_s}{r} + kr. \tag{2.1}$$

The potential increases proportional to the distance between two quarks. If one tries to separate a quark from a hadron, at some point the potential energy is high enough to form a new quark / anti-quark pair out of the vacuum. Instead of separating the quarks one creates two hadrons which both have no net-color charge.

The factor α_s in formula (2.1) is the running coupling constant. It can be written as

$$\alpha_s(Q^2) = \frac{12\pi}{(33 - 2n_f)\ln(Q^2/\Lambda^2)}. \tag{2.2}$$

In this equation n_f is the number of involved quark types, Λ a scale factor and Q the momentum transfer. In case of high momentum transfer which is equivalent to small distances the coupling constant becomes weak and the quarks can move freely. This property of the QCD is called asymptotic freedom.

2.2 The Quark-Gluon Plasma

In case of low densities and low temperatures, quarks and gluons are confined to the size of hadrons and build the normal hadronic matter. When density and temperature become very high there is a phase transition to a new state, the quark-gluon plasma (QGP). In a QGP quarks and gluons are not longer confined to hadrons but can move freely. Figure 2.1 shows the corresponding phase diagram.

Nowadays the only place where a QGP may exists is the core of neutron stars. But some 10 picoseconds after the Big Bang lasting for 10 microseconds the complete universe was filled with a QGP [BMS07]. To investigate this phase and to understand the properties of a QGP it has to be created in laboratory. The only possibility to get the necessary energy density is to collide heavy ions at highest available energies. A sketch of such a collision is shown in figure 2.2. Two nuclei, Lorentz contracted because of their speed, collide with each other. In the moment of collision scattering between the partons takes place. In case of a large momentum transfer during a scattering the scattered partons will form jets later (see section 2.3.2). The nuclei pass each other and in the collision zone an extremely high energy density is reached. Thousands or even tens of thousands of quarks and gluons are created. The created particles collide and after a time of about $1\,\mathrm{fm\,c^{-1}}$ thermal equilibrium is reached. Since a QGP is a collective state thermal equilibrium is a necessary condition to define variables like temperature or pressure. The QGP expands and cools down. The temperature drops below the critical temperature where a QGP can exists and the re-hadronisation takes place. The quarks and gluons are bound to hadrons again and the QGP vanishes. This moment is called chemical freeze-out. After the chemical freeze-out the number and the type of created hadrons are fixed. The cloud of particles expands further and at some point the mean free path of the particles becomes larger then the size of the cloud. This point in time is the kinetic freeze-out. Afterwards the energy distribution of the particles is fixed, too. The particles leave the collision zone and can be detected by particle detectors.

There is no possibility to directly investigate a QGP because of its small size and short

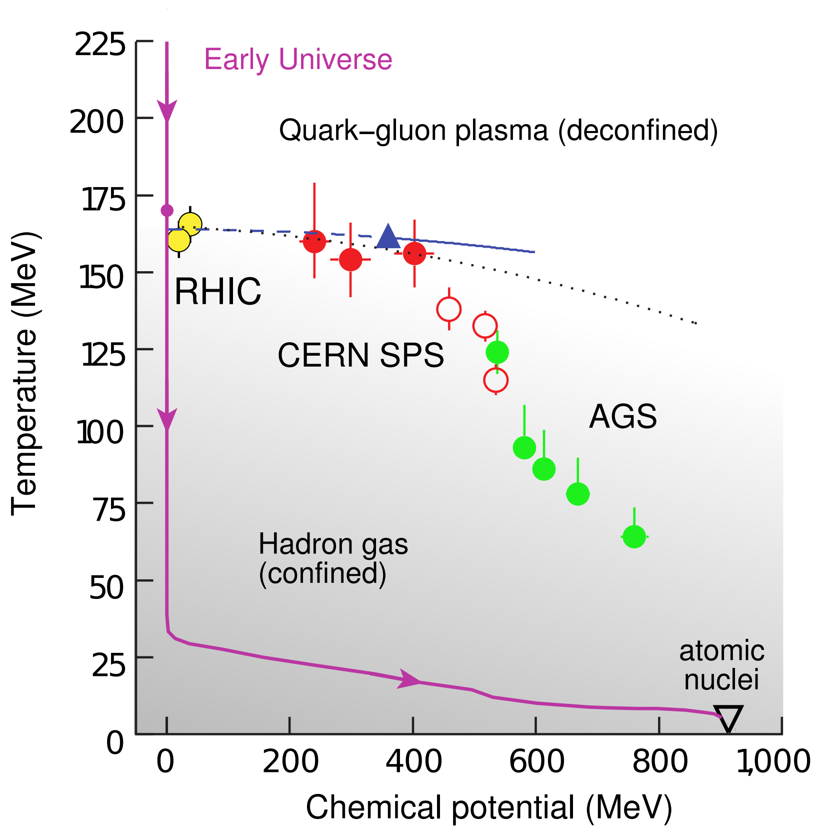


Figure 2.1: Phase diagram of strongly interacting matter. The data points are obtained from experimental data measured at the three accelerators AGS, SPS (CERN) and RHIC, using a thermal fit model to calculate the temperature T and the chemical potential μ_b at chemical freeze out. Filled circles are from analysis of mid-rapidity data, open circles from analysis of 4π data. Furthermore theoretical expectations for the phase boundary between confined and deconfined matter are shown. The dotted line is calculated with the bag model and the dashed/solid line is from lattice quantum chromodynamics calculations. The filled triangle indicates a possible position of a critical endpoint. The magenta line shows the evolution of the Universe in its first moments of existence. (Plot based on [ABMS06] and [BMS07]).

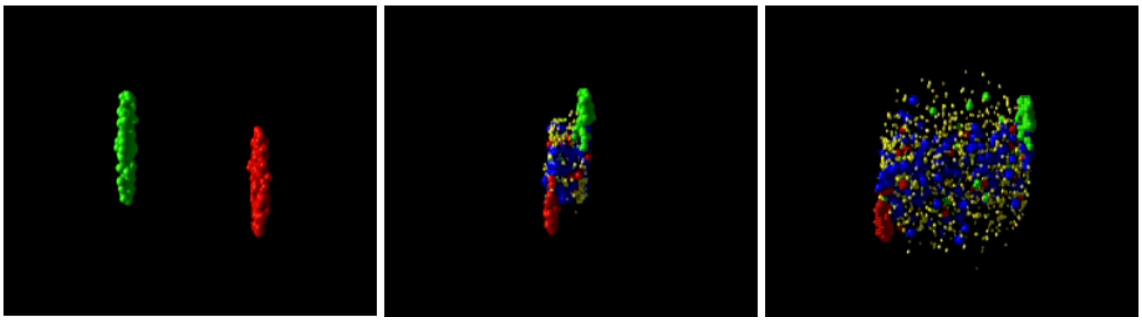


Figure 2.2: Simulation of two colliding lead nuclei at $\sqrt{s_{NN}} = 160\,\text{GeV}$ [Web07].

lifetime. Instead, the type of the created particles, their energies and their moving directions are recorded. These data are used to deduce properties of the QGP.

2.3 Signatures of the Quark-Gluon Plasma

There is no single non-ambiguous observable for the formation of a QGP. However, different signatures of a QGP are predicted. Although they can be explained by other mechanisms as well, the QGP is the best explanation for the simultaneous observation of different signatures.

This section gives a short overview about the most prominent signatures of a QGP.

2.3.1 J/ψ Suppression

The J/ψ is a bound state of a charm and an anti-charm ($c\bar{c}$) quark. Data for the production rate of J/ψ and other quarkonia states without the presence of a QGP can be obtained from proton-proton collisions or from collisions of light nuclei. In case of a QGP, the quarks are deconfined and move freely inside the plasma. At the time of re-hadronization it is very unlikely that the created c and $\bar{c}$ quarks are close together and hadronize to a J/ψ. Instead, they will be bound with the much more abundant (anti-)up, (anti-)down or (anti-)strange quarks to other hadrons. Therefore a suppression of the J/ψ production is expected (e.g. [MS86], [Sat90], [YHM05]).

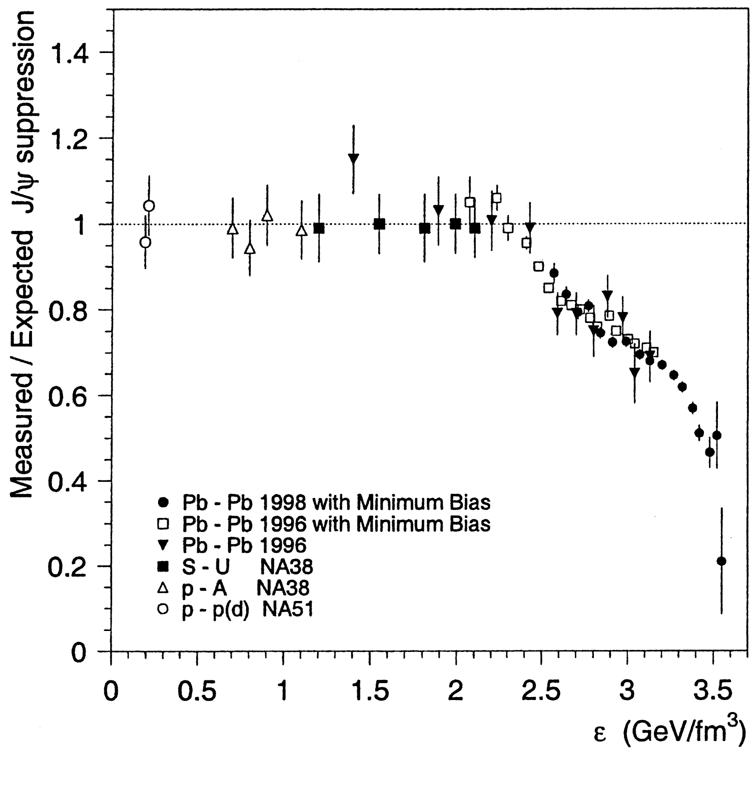


Figure 2.3: Measured J/ψ production yields, normalised to the yields expected assuming that the only source of suppression is the ordinary absorption by the nuclear medium [NA500].

Cold dense medium absorbs some of the created J/ψ. This effect is called normal suppression. But even if one takes this normal suppression into account, an additional suppression was seen by the NA50 collaboration (see figure 2.3). This additional suppression is likely caused by the presence of a QGP [NA500].

For LHC energies the expectation is different. At these collision energies a large number of charm and anti-charm quarks will be created. At hadronization there is a certain chance that charm and anti-charm quarks will be close together just for statistical reasons and form a J/ψ. Therefore an enhancement rather than a suppression is expected for the LHC data [BMS07].

2.3.2 Jet-Quenching

During the collision of nuclei in an accelerator, hard parton-parton collisions take place. Depending on the collision parameters the two partons can get large transverse momenta. Without the presence of a QGP the scattered partons fragment directly into hadrons which are moving in the direction of the primordial parton. This is called a jet. Since both scattered partons form a jet one sees two jets in an φangle of 180 °.

In a QGP the situation is different. The parton scattering takes place in a very early stage of the collision. In most cases the scattering does not happen just in the center of the collision zone. Therefore one of the partons has to travel a longer distance in the hot and dense QGP where it will loose energy. As a result, the correlation between the two jets should be distorted [YHM05].

2.3.3 Strangeness Enhancement

Strange particles are created always in pairs due to strangeness conservation. Therefore in collisions without a QGP two mesons have to be created. The creation of a kaon pair has a threshold energy of about 987 MeV. Since the kaon is the lightest meson with a strange quark the mentioned threshold energy is the general threshold energy for the creation of particle pairs with strange quarks. In the quark gluon plasma the $s\bar{s}$ can be produced directly. Therefore only the energy for the two quarks is needed. The threshold energy for the creation of a $s\bar{s}$ pair is about 300 MeV, much lower than for a kaon pair. Consequently, the number of produced strange particles will be higher in case of a QGP [YHM05].

2.3.4 Direct Photons

Direct photons are all photons not originating from hadronic decays. They can be subdivided in prompt photons emitted during hard scattering and thermal photons emitted from a thermally equilibrated phase. Since photons are not affected by the strong interaction and hadronization processes, they are a tool to study the different stages of a heavy ion collision.

Due to a large background from decay photons, direct photon signals are difficult to extract from the measured data. Uncertainties in the theoretical calculations complicate the interpretation of the data [KB04, ST01].

3 LHC and the ALICE experiment

3.1 The Large Hadron Collider

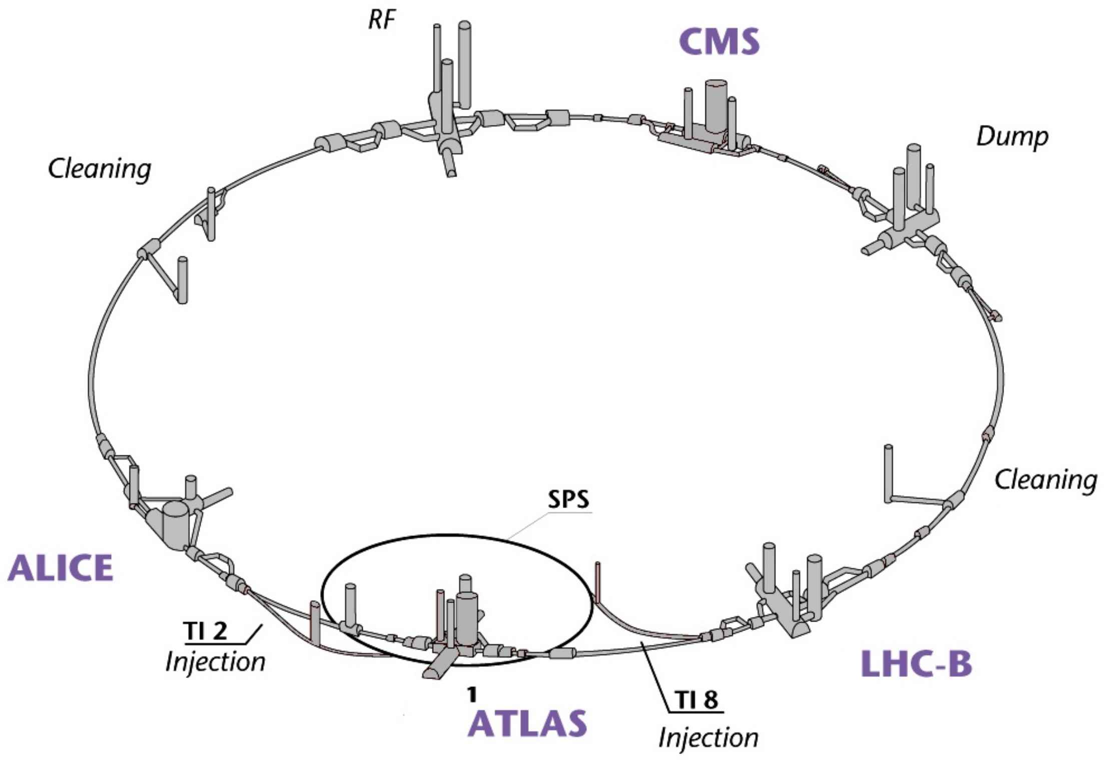


Figure 3.1: Map of the Large Hadron Collider and its four experiments [Car97].

In September 2008 the Large Hadron Collider (LHC) has started its operation at CERN near Geneva. It is build in the tunnel previously used by the Large Electron-Positron-Collider (LEP). The tunnel has a circumference of about 27 km and is between 50 m and 175 m below the surface.

The LHC provides proton-proton collisions with a center-of-mass energy up to $\sqrt{s_{NN}} = 14\,\text{TeV}$ and a high luminosity of $10^{34}\,\text{cm}^{-2}\text{s}^{-1}$. Several weeks a year there will be heavy-ion collisions with center-of-mass energies up to $\sqrt{s_{NN}} = 5.52\,\text{TeV}$ [CER08]. Until the start of LHC the most powerful accelerator was the Tevatron at Fermilab (Illinois, USA), which collides protons with anti-protons. The LHC exceeds the Tevatron energies seven times and the luminosity even by two orders of magnitude. The most powerful heavy-ion collider is RHIC at the Brookhaven National Laboratory (Long Island, New York). In heavy-ion mode the LHC provides center-of-mass energies which are 30 times higher than the RHIC energies and one order of magnitude higher luminosity.

The collisions provided by the LHC are analyzed by four large experiments. ATLAS, LHC-B, and CMS are mainly interested in the proton-proton collisions. The only dedicated heavy-ion experiment at LHC is ALICE. Figure 3.1 shows a map of the LHC with the four experiments.

The **ATLAS** experiment (A Toroidal LHC Apparatus) is the largest experiment at LHC. With a length of 46 m, 25 m height and 25 m width it is the largest volume particle detector ever constructed. One task of ATLAS is to find the Higgs boson. The Higgs boson is the last particle of the Standard Model which is not experimentally detected yet. From experiments it is known that the mass of the Higgs boson is at least 114.4 GeV/c^2 [AoG08]. ATLAS will be able to detect a Standard Model Higgs boson in an energy range from 80 GeV up to about 1 TeV with high significance [ATT99]. This energy range covers most of the theoretical expectations for a Standard Model Higgs boson. Furthermore, the ATLAS experiment looks for physics beyond the Standard Model like extra dimensions of space or SUSY particles which are dark matter candidates. Perhaps ATLAS will also see disintegration of microscopic black holes which may be created in the collisions.

The **CMS** (Compact Muon Solenoid) experiment is the competitor of ATLAS. In general it investigates the same areas of physics as the ATLAS experiment, especially the search for the Higgs boson. This competition is necessary. As soon as one of the two experiments claims to has seen the Higgs boson a second experiment is needed to verify the results. Already during the design phase it was kept in mind to guarantee the results of both experiments to be comparable. CMS has a total length of 22 meters and a diameter of 15 meters. [BDNFP06]

The **LHCb** (Large Hadron Collider Beauty) experiment will study CP violation and other rare phenomena in B meson decays. In its design it differs from the three other experiments. While the other experiments have full azimuthal coverage the LHCb experiment is a single arm forward spectrometer since the decay particles of the produced B mesons only have a small opening angle.

The **ALICE** experiment will be described in the next section in detail.

3.2 The ALICE Experiment

ALICE (A Large Ion Collider Experiment) is a detector focused on investigating heavy ion collisions. Its main goal is to investigate the properties of the quark gluon plasma which is expected to be created during the collisions of heavy ions in the LHC. However, ALICE will analyze proton-proton and light ion collisions as well. These data are needed for calibration and comparison purposes but there will be dedicated proton-proton analyses, too.

The ALICE detector is capable to identify and track hadrons, leptons and photons in a broad range of transverse momenta from about 100 MeV/c up to 100 GeV/c.

A schematic view of the detector is shown in figure 3.2. Most of the sub-detectors are located inside a big magnet which provides a magnetic field of 0.5 T. This magnet was built for the previous L3 experiment and is reused by ALICE.

The innermost sub-detector of ALICE is the Inner Tracking System (ITS) which is located from 4 cm to 44 cm in radial direction. It consists of six layers. The first two layers are pixel detectors, layer three and four are silicon drift detectors and the two outermost layers are double-sided silicon micro-strip detectors. The tasks of the ITS are to locate the

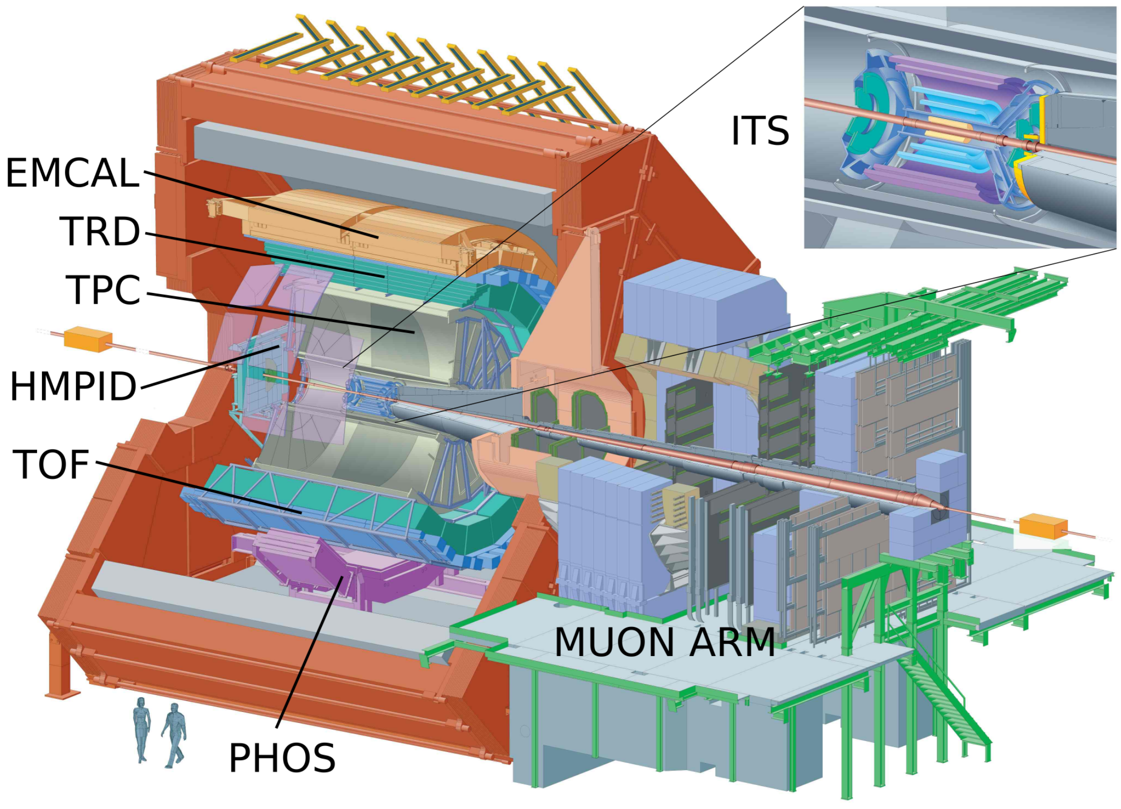


Figure 3.2: Layout of the ALICE detector [ALI].

primary vertex and secondary vertices of short-lived particles. Furthermore it improves the overall particle tracking capability of ALICE.

The second detector in radial direction is the Time Projection Chamber (TPC). It covers the space from 0.85 m to 2.5 m in radial direction and has a length of 5 m. The TPC is the main tracking detector of ALICE. It is capable to reconstruct the tracks of 20,000 charged particles per event. On the other hand the TPC is a slow detector due to the operating principle. Each charged particle crossing the TPC leaves a track of liberated electrons and ionized molecules inside the gas volume of the TPC. Due to a high electrostatic field inside the TPC the electrons drift to the end-caps. Multi-wire proportional chambers are mounted at the end-caps which detect the electrons. Hence a track is translated into a time signal: the further a track segment is away from the end caps the later the electrons of this track segment will arrive at the end-caps. Since the drifting of the electrons takes up to 88 µs the TPC is the slowest detector of ALICE [TPC00].

Next to the TPC the Transition Radiation Detector (TRD) is located. It covers the space from 2.9 m to 3.7 m in radial direction. This detector will be described in chapter 4 in detail.

The Time of Flight (TOF) is the outermost detector with full azimuthal coverage. It consists of 18 modules in azimuthal and 5 modules in beam direction, resulting in 90 modules in total. The active area of the detector is about 150 m^2. The TOF detector identifies particles by measuring their flight time. Therefore a very high time resolution is needed.

The detector is based on Multigap Resistive-Plate Chambers. Each particle crossing a chamber immediately causes an avalanche which is detected. With this method the TOF detector reaches a time resolution of 150 ps which is sufficient to identify particles in the momentum range from 0.3 GeV/c to 2.5 GeV/c [GA03].

Inside the L3 magnet there are some detectors with small acceptance. The High Momentum Particle Identification Detector (HMPID) is a ring-imaging Cherenkov detector. It is dedicated to extend the PID to momenta of 3.5 GeV/c for kaons and 5 GeV/c for protons. The Electromagnetic Calorimeter (EMCal) enhances the capabilities of ALICE to study jet quenching. It provides an efficient and unbiased fast trigger for high energy jets, and is able to measure of the neutral portion of jet energy [ECT08]. The Photon Spectrometer (PHOS) consists of an array of crystals and is dedicated to photon detection.

Outside the L3 magnet is a muon arm which tracks muons and measures their energy.

3.3 The ALICE Control System

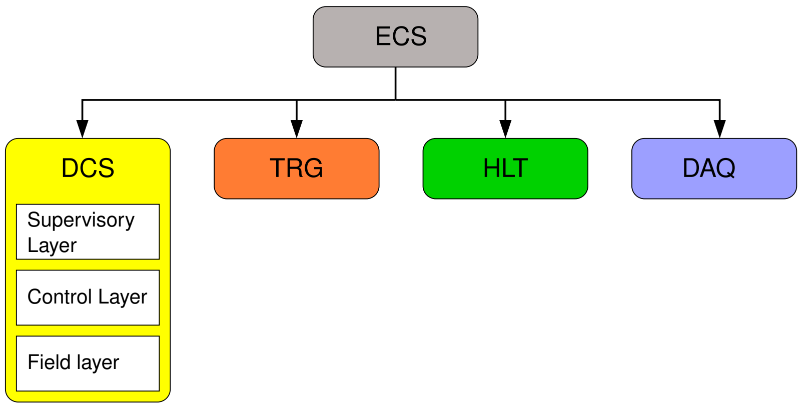


Figure 3.3: Scheme of a the ECS system. The three hierarchy levels of the DCS system are shown as well (see section 3.3.2). (based on [CCC+04]).

The ALICE detector consists of a large number of subsystems. Due to dependencies between the systems, coordination instances are needed. Each subsystem belongs to one of four activity domains (see figure 3.3): The Detector Control System (DCS) controls all subdetectors of ALICE and their supply systems like power supplies or cooling plants. The trigger system (TRG) evaluates inputs from trigger detectors, generates trigger signals in case of interesting events and distributes the trigger signals to the subdetectors. The High Level Trigger (HLT) is a large computer system. Each time the trigger system has caused the full read out of the subdetectors the HLT gets the data and performs more detailed analysis than the trigger system can do. Based on this analysis the data are either discarded or stored on hard disks/tapes by the DAQ system. The Data Acquisiton (DAQ) contains the systems to collect the data from the subdectors and to store them. These four activity domains are controled by the Experiment Control System (ECS). It provides an uniform user interface to control ALICE.

Obviously, neither the ECS nor the activity domains can include all the details how the individual subsystems have to be operated. Each subsystem has still its own control system. To keep the interfaces between the ECS, the activity domains and the individual control systems as small as possible, the concept of finite state machines is used at all levels of the control hierarchy.

3.3.1 Finite State Machines

A finite state machine (FSM) is a model of behavior. It is based on transitions and states. A state defines an actual configuration of the system. A transition describes how to get from one state to another. It is not possible to reach a state B from a state A if no transition is defined, which describes how to get the system from A to B. At any time the system is either in one of the states or performing a transition. The second situation is called a transition state.

In ALICE, finite state machines are used at all levels of the control hierarchy. But there is one important difference between the usual implementations of FSMs and the way they are implemented in ALICE. Usually a state defines all properties of a system, including the configuration. An experiment like ALICE has many possible configurations and it is not feasible to define a separate state for each of them. In ALICE separate states in the FSMs are only defined for 'fundamentally different' states of the detector. For example a 'fundamental' difference is if the detector is taking data or if it is just turned on. The set of possible states is the same for all subsystems even though not all subsystems can switch to every state. The different configuration data for a state are stored in configuration files or data bases of the individual subsystems.

When the ECS initiates a transition, each subsystem checks for itself which configuration is currently connected with the requested transition and applies this configuration. The drawback of this approach is that one has to log not only the states/transitions but also the applied configurations. Section 5.2.2 describes how this concept of a FSM is implemented in the control software for the frond end readout electronic of the TRD.

Most of the subsystems use the language SMI++ to implement their FSMs. The first version of SMI was developed for the DELPHI experiment. SMI is a programming language designed to provide a compact and easy-to-use syntax for writing finite state machines. Later, it turned out that SMI had some disadvantages. Therefore it was completely reimplemented in C++ and the result was called SMI++ [GF97]. In the TRD, the FSMs of the control systems in the upper layers (supervisory layer and control layer, fig. 3.3) are implemented in SMI++. However, the FSM in the control engine (see chapter 5) is implemented without using SMI++.

3.3.2 Detector Control System

The Detector Control System (DCS) is one of the four activity domains of the Experimental Control System. Its primary task is to ensure the correct and save operation of the ALICE subdetectors. This includes for example configuring and monitoring of the subdetectors, alarm handling and logging.

In the DCS, each subdetector, like the TRD or the TPC, is represented by its own control system which communicates with the DCS/ECS via interfaces defined by finite state machines. The details of the individual control systems may vary but the general design is always the same.

The control system of a subdetector consists of three functional layers: the supervisory layer, the control layer and the field layer (see figure 3.3).

Supervisory layer The top layer is the supervisory layer. It has two functions. First, when ALICE is operated via the ECS interface, this layer distributes the commands from the DCS to the subsystems in the control layer. Second, the software in the supervisory layer provides user interfaces to operate subsystems on their own. This mode is used during commissioning, to test configurations, and for debugging.

The software in this layer is based on PVSS II, a commercial product developed by the Austrian company ETM [EMT]. PVSS II is a SCADA (Supervisory Control and Data Acquisition) system which supports the creation of user interfaces, provides hardware access via different communication protocols and has ready-to-use components for alarm handling, logging and access control, just to name a few. Since PVSS does not support finite state machines directly, SMI++ (see subsection 3.3.1) was ported to PVSS.

Control layer The second layer is the control layer. The control layer mainly contains PCs which are directly connected to hardware, like high voltage systems or cooling plants. These PCs configure and monitor the connected hardware and report the hardware status to the supervisory layer. Most of these systems run PVSS, too. Furthermore this layer contains databases which store configurations for systems in the field layer.

Field layer The third layer is the field layer. It includes hardware like power supplies or cooling plants. Most of the systems do not have a complex control software because they are directly operated by systems in the control layer. However, in some cases like the DCS boards of the TRD (sec. 4.3.4), programmable computers are used to run control software in this layer.

3.3.3 Distributed Information Management System

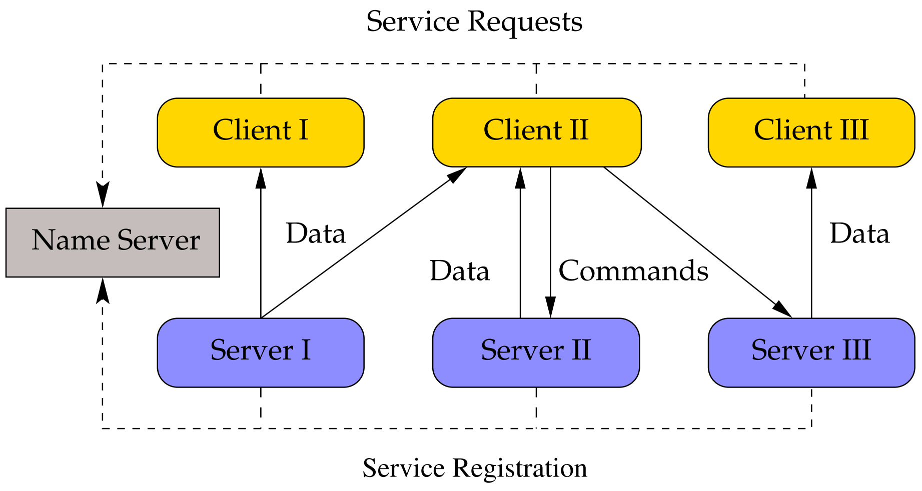


Figure 3.4: Scheme of a DIM system with three servers and three clients (based on [GD93]).

Each control system requires communication between the several components. For ALICE a reliable system is needed which is easy to use in very different kinds of control software and runs on various platforms. Furthermore the needed hardware resources should be as small as possible since some computers like the DCS boards (see section

4.3.4) only have very limited hardware resources. All these demands are met by the DIM system.

The Distributed Information Management System (DIM) [GD93], formerly known as DELPHI Information System, was developed by the DELPHI experiment at CERN. It is designed to provide an easy to use and robust way to exchange information between different devices in a network. For the actual data transfer DIM uses the TCP/IP protocol. The DIM libraries provide interfaces to C, C++, Java and Fortran.

DIM handles data transfers via named services. A named service is a user defined data structure with an arbitrary name. The name is used to identify the service in the DIM system. An example of a DIM system is shown in figure 3.4.

A DIM server which has named services to publish contacts the DIM name server each time it starts up. The server informs the DIM name server about the names and types of services it provides. As long as the server is running it sends watchdog messages to the name server. Due to this mechanism the name server has an up-to-date list of all available services in the network at any time. A client which wants to subscribe to a service asks the name server if the service is currently available. If the service is available, the name server returns the IP address of the server which provides the requested service. All further communication is done between the client and the server directly, without contacting the name server again. If the requested service is not available, the name server stores the request and informs the client as soon as the service becomes available again.

This design provides large reliability. Dead servers are detected by the name server and all clients get reconnected automatically as soon as the server is running again. A failure of the name server does not affect servers and clients which are running already. All three components can be migrated easily from one machine to another.

The DIM system supports two types of services. The first type are data services. This type is used to transfer data like sensor readings or status informations to clients. The clients can use one of three methods to connect to such a service:

- ONCE-ONLY — In this mode the client sends a separate request each time it needs the current data.
- TIMED — The client gets the current data in regular time intervals. The timing is handled by DIM.
- MONITORED — The client gets updated data each time the data change.

The second type of services are command services. They are used to transfer data like configuration or commands from clients to servers.

4 The ALICE Transition Radiation Detector

The ALICE Transition Radiation Detector (TRD) is located just outside the TPC in radial direction, starting at a radius of 2.9 m and extending to 3.7 m. It is designed to fulfill three tasks. The first one is electron-pion separation at momenta in excess of 1 $^{\mathrm{GeV}}/_{\mathrm{c}}$. In this energy region the pion identification via energy loss measurement in the TPC is no longer sufficient. Second, the TRD is a fast tracker. Therefore it can be used as a trigger for high transverse momentum electrons or jets with high E_t. Especially when looking for rare particles such a trigger is necessary to enhance the statistical significance of the measurement. Third the TRD adds up to six more points to the ITS/TPC tracks. Since the TRD is located outside the TPC in radial direction the integrated effect of the magnetic field to the particle trajectory is higher then inside the TPC. So the trajectory points from the TRD increase the momentum resolution of ALICE significantly.

4.1 Detector Design and Mode of Operation

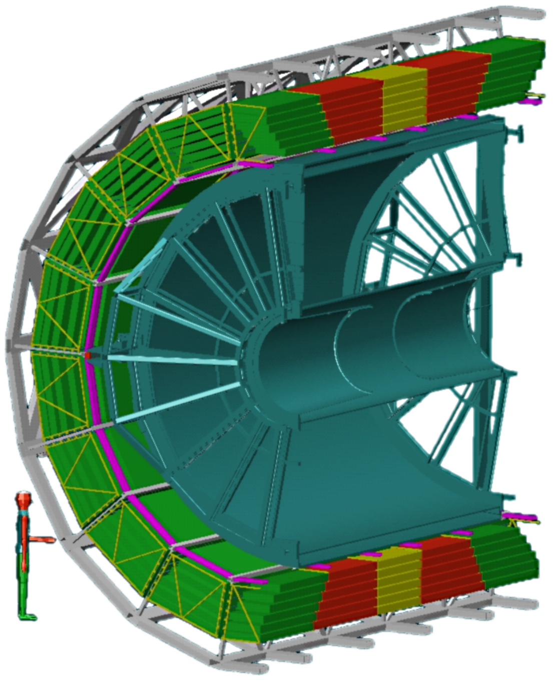


Figure 4.1: One half of the TRD with the TPC inside shown, too. Each supermodule has five stacks (green, red, yellow, red, green) with 6 layers per stack.

The TRD is build out of 18 supermodules in azimuthal direction. Each supermodule is about 7 m long and covers a pseudo-rapidity of $|\eta| = 0.9$. It is built out of 30 Readout

Chambers (ROC) organized in five stacks with six chambers per stack (see fig. 4.1) which leads to a total number of 540 chambers in the TRD. The TRD has two different ROC types with six different sizes per type. The C0 type are smaller chambers with six readout boards. This type is used in the central stack of the supermodules (yellow in fig. 4.1). The C1 type chambers have eight readout boards and are used in the other four stacks. The size of a chamber increases form 91 cm $\times$ 122 cm (C0 type, layer 0) to 113 cm $\times$ 145 cm (C1 type, layer 5).

A ROC is the smallest independent unit of the detector. It consists of a radiator, a drift region, a multi-wire proportional chamber and the read-out electronics mounted on the top of the chamber. A cross section of a chamber is shown in figure 4.2 on page 25. The total active area of the TRD chambers is 683 m^2 and the total gas volume is about 27 m^3[TRD01].

4.1.1 Transition Radiation

Ultra-relativistic charged particles ($\gamma \gg 1$) passing the boundary between materials with different dielectric constants emit transition radiation. In case of a boundary between vacuum and a medium a passing particle emits transition radiation with a total energy of [Jac75]

$$I = \int_0^\infty \frac{dI}{d\nu} d\nu = \frac{z^2 e^2 \gamma \omega_p}{3c} = \frac{z^2}{3 \cdot 137} \gamma \hbar \omega_p. \tag{4.1}$$

In the equation $\nu = \frac{\omega}{\gamma \omega_p}$ with ω_p is the plasma frequency of the medium. It can be shown that the transition radiation is emitted in a small cone ($\simeq 1/\gamma$) around the particle path [Dol93]. The frequence of the transition radiation extends up to the X-ray region [Dol93, Jac75].

The dependence on the factor γ in (4.1) makes transition radiation attractive for electron/ positron identification. Electrons or positrons with an energy of $E = 1\,\mathrm{GeV}$ have a γ-factor of

$$E = \frac{mc^2}{\sqrt{1 - v^2/c^2}} = \gamma \cdot mc^2 \;\Leftrightarrow\; \gamma = \frac{E}{mc^2}, \tag{4.2}$$

$$\Rightarrow \gamma = \frac{1\,\mathrm{GeV}}{\frac{511\,\mathrm{keV}}{c^2} \cdot c^2} \approx 2000. \tag{4.3}$$

After electrons/positrons, pions are the next lightest charged particles which are created in significant amount in the collisions. They have a mass of $m = 139.6\,\mathrm{MeV/c^2}$ [AoG08]. A 1 GeV pion has a γ factor of about

$$\gamma = \frac{E}{mc^2} \Rightarrow \gamma = \frac{1\,\mathrm{GeV}}{\frac{139.6\,\mathrm{MeV}}{c^2} \cdot c^2} \approx 7 \tag{4.4}$$

Therefore it is very unlikely that charged pions create transition radiation.

The constant factor $1/3\cdot137$ in equation (4.1) indicates that the emission of a transition radiation photon at a single boundary is unlikely. Therefore many boundaries are needed.

4.1.2 Readout Chamber Design

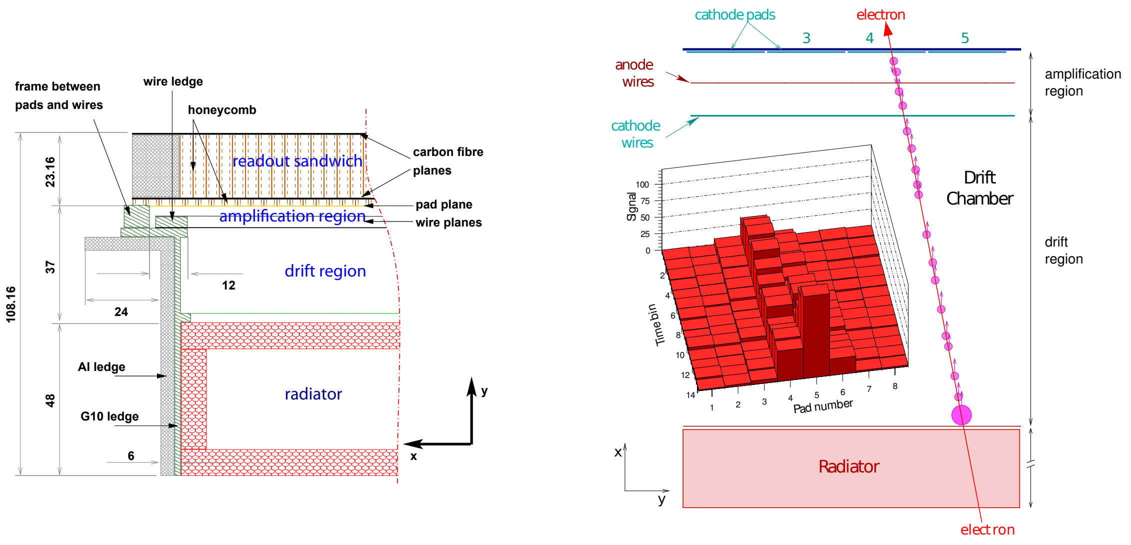


Figure 4.2: Right: Cross-section of a TRD chamber ([Mah04]). Left: An electron is passing the chamber and leaves a track of ionized particles. Embedded is the resulting signal on the cathode pads [TRD01].

The TRD readout chamber can be divided into four parts (see figure 4.2): radiator, drift region, amplification region and the readout section.

In the radiator charged particles with a large γ factor create transition radiation. To achieve a good radiation yield a large number of boundaries in the radiator is needed. In case of the TRD a fibre mat radiator was chosen. The main radiator material are polypropylene fiber mats whereas the covers and the walls of the radiator are made out of Rohacell HF71 foam reinforced by glass fiber sheets. To increase the mechanical stability not the complete volume of the radiator is filled with the fiber mats. The top and bottom covers of the radiator are connected with 8 mm Rohacell plates which form a grid like structure. The fields of the grid are filled with the polypropylene fiber mats. Fortunately also the Rohacell is a quite good radiator material so that the dead material inside the radiator is minimized. Figure 4.3 shows the microscopic structure of the Rohacell and the fiber mats. One can see clearly the huge amount of boundary layers which are essential for a high transition radiation yield.

Usually one uses a stack of tense foils as a radiator (foil radiator) due to the better transition radiation yield compared to a fibre mat radiator. In case of TRD this was not possible because the space frame structure needed for the tense foils would add too much inactive material which decreases the efficiency of the TRD and would affect the detectors outside the TRD, too. Secondly the radiator should add as much as possible to the mechanical stability of the chamber.

Directly at the top of the radiator a 25 µm aluminized Mylar foil is glued. This foil is the entrance window to the drift region of the chamber. The Mylar foil is biased with a negative potential of -2.1 kV whereas the cathode wires which separate the drift region from the amplification region are at ground potential. The drift and the amplification region are filled with a gas mixture of 85% xenon and 15% CO_2.

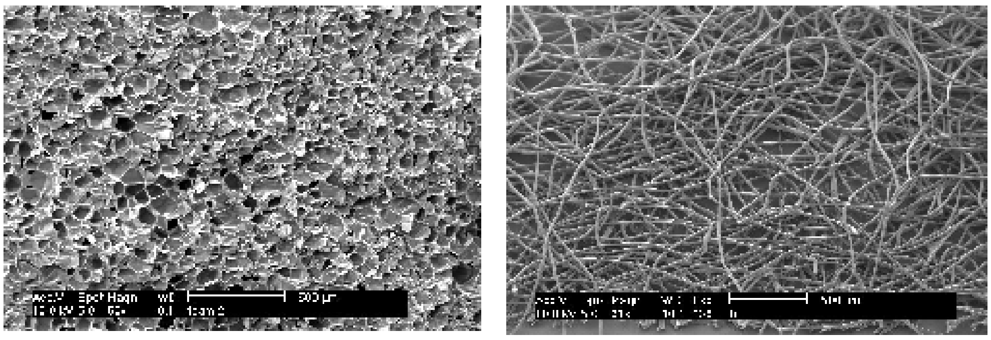


Figure 4.3: Scanning electron microscope images of the radiator materials. Left image: Rohacell HF71 foam; right image: fiber mat (images from [TRD01]).

Charged particles crossing the chamber leave a track of electrons and ionized gas molecules. However, the number of electrons liberated by the primary particle is not sufficient to get a measurable signal. Therefore an amplification is needed which is done in the amplification region.

The amplification region is a multi-wire proportional chamber. It is separated from the drift region by the cathode wire plane. The cathode wires are at ground potential. They shield the electromagnetic field created by the anode wires. The anode wires are biased by a positive potential of about +1.5 kV to +1.6 kV.

The pad plane terminates the amplification region at the other end and is the bottom part of the readout section. The pad plane is at ground potential. It consists of printed circuit boards that contain the copper cathode pads. The size of the cathode pads depends on the chamber size but the average size is about 6.2 cm^2. To get mechanical stability, the backing of the printed circuit boards is made of carbon fiber reinforced Rohacell. The only function of the readout sandwich is to add more mechanical stability to the complete chamber. The last part of the readout section are the readout boards which are mounted at the top of the readout sandwich (not shown in figure 4.2). The readout boards will be described in section 4.3.1 in detail.

4.1.3 Signal Generation

A particle first passes the radiator. In case of ultra-relativistic electrons there is a high probability to produce transition radiation in the X-ray region of the spectrum. After the radiator the particle passes the drift region and liberates electrons along its path. Most of the transition radiation photons that reach the drift region are absorbed within the first centimeter of the drift region[1] and liberates electrons, too. The liberated electrons drift towards the cathode wires whereas the xenon ions drift to the entrance window due to the field gradient between the entrance window and the cathode wires. When the electrons pass the cathode grid, the steep field gradient close to the anode wires accelerates the electrons and starts an avalanche. The electrons liberated in the avalanche quickly reach the anode wires and are absorbed. However, the created gas ions drift much slower and

[1] A typical transition radiation photon has an energy of 10 keV which results in an absorption length of 1 cm in xenon [TRD01].

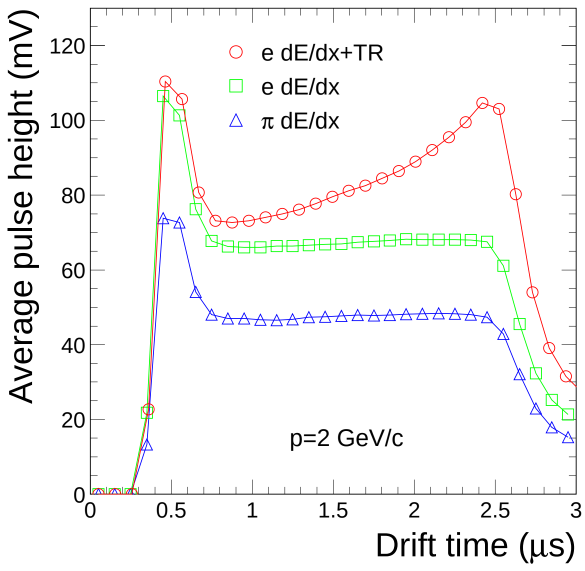


Figure 4.4: Average pulse height as a function of drift time for pions and electrons with a momentum of 2 GeV/c (with and without transition radiation) (image taken from [And04]).

induce a measurable image charge on the cathode pads. This charge is read out by the front end read out electronics every 100 ns.

Figure 4.4 shows the time evolution of the signal for different particles. One can divide the signal in four sections. The first three points are recorded before the particle crosses the chamber. Then the particle flies through the chamber and liberates electrons along its path. First the electrons liberated in the amplification region reach the anode wires. Since electrons from both sides of the anode wire plane reach the wires the number of avalanches and therefore the number of created ions is high. This results in a sharp increase of the signal. A few time bins later the electrons liberated in the amplification region all have reached the anode wires. Now the first electrons liberated in the drift region near the cathode wires reach the anode plane. Since no electrons reach the anode plane from the pad plane side anymore, the number of avalances decreases. This leads to a decreasing signal. In case of electrons, at the end of the drift time additional electrons liberated by the transition radiation reach the anode plane. The number of avalanches and therefore the number of created ions increases again and leads to a rising signal.

4.2 Infrastructure Services

A supermodule needs for operation four infrastructure services: low voltage and cooling for the front end readout electronics and high voltage and the correct gas mixture for the drift and amplification region of the chamber.

Low Voltage System

The voltages for the frontend readout electronics of the TRD are provided by 89 water-cooled Wiener PL512/M low voltage power supplies. Each power supply has an ethernet connection and can be addressed separately by the detector control system. The power supplies are located outside the L3 magnet and are connected to the supermodules by thick copper cables. All together, in normal running conditions the Wiener power supplies have to provide an electrical power of more than 65 kW for the supermodules plus the power lost in the cables between the power supplies and the supermodules.

In the supermodule the power is distributed by copper bars on the inner sidewalls of the supermodules. The individual ROBs are connected to the copper bars by short cables. The MCMs on the ROBs require four supply voltages for correct operation.

Each MCM consists of two chips – a charge sensitive preamplifier (PASA) and a digital chip named TRAP. The PASA requires 3.3 V and the analog part of the TRAP requires 1.8 V for operation. Because both voltages supply analog parts of the electronics they are called analog voltages. The corresponding ground is the analog ground (A_GND). The digital part of the TRAP requires 1.8 V and 3.3 V. These two voltages are called digital voltages. The corresponding ground is the digital ground (GND).

The voltages on the ROBs can be switched on and off by power regulators on the ROBs which are controlled by the DCS boards of the readout chambers. The main task of the voltage regulators is to reduce the voltages to exact 1.8 V and 3.3 V respectively. Due to voltage drops in the copper bars of the supermodule, the Wiener power supplies are configured to deliver about 2.8 V on the 1.8 V lines and 4.1 V on the 3.3 V lines. The voltage drops in the power cables between the power supplies and the supermodules are compared by the power supplies automatically. The power supplies measure the voltage at the end of the power cables via separate lines and increase/decrease their output voltage until the configured voltage is measured at the end of the power cable.

The DCS boards require 3.3 V for operation. Since they have internal voltage regulators the usual supply voltage is about 4 V. The DCS boards get their power via a separate channel. Each supermodule has a power distribution box (PDB) built in. The power distribution box is connected on the one side to a Wiener power supply and on the other side a separate cable connects each DCS board in the supermodule with the PDB. Inside the PDB a DCS board controls the power distribution and can switch on and off the DCS boards of the supermodule individually.

Cooling System

One supermodule of the TRD contains about 4,000 MCMs, 30 DCS boards and other electronics. All these parts produce heat during their operation. This heat must be dissipated in order to keep the components within their safe operating temperatures. During the development of the TRD it turned out that air cooling is not sufficient, mainly because the gap between the top side of the electronics of one layer and the bottom of the chambers of the next layer is too small. Therefore a water cooling system became necessary.

Most of the heat is produced by the MCMs, the power regulators and the ICs on the DCS board and the ORIs. To cool these components they have thin aluminum plates glued on top. Aluminum tubes are glued to the top of these plates and conduct the cooling water. The aluminum tubes are connected by silicon hoses with the main cooling

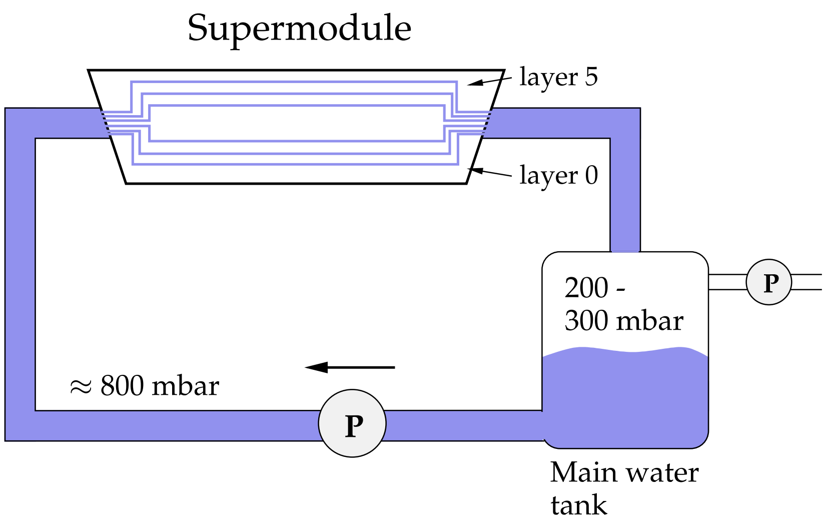


Figure 4.5: Schematic view of the cooling system. The right pump removes air form the main water tank to keep the pressure level whereas the left pump circulates the cooling water.

lines mounted on the walls of the supermodule. The cooling plant which circulates and cools the water is located outside the detector.

Since the electronics is very water sensitive and the supermodules are not accessible once installed inside the ALICE detector, the cooling system is an under-pressure system (see figure 4.5). The main cooling water tank in the cooling plant is kept at a pressure level of $200 - 300$ mbar by a pump which removes air out of the tank. A second pump circulates the water through the supermodules. Due to the significant low-pressure in the main tank the pressure level in all pipes and tubes of the cooling system is below atmospheric pressure.

The main feature of this system is obvious. If somewhere in the supermodule a leak occurs air is drawn in the cooling tube instead of water leaking out. As long as the amount of incoming air is smaller then the amount of air the pump can remove from the tank it is even possible to keep the whole system in normal operation.

Figure 4.5 shows a schematic view of the cooling system and figure 4.6 (page 31) shows a readout chamber with cooling installed in a supermodule. Chapter 6 describes how the proper cooling of the MCMs can be monitored.

High voltage

Each readout chamber requires two different high voltages (HV): a negative HV for the drift region and a positive HV for the anode of the amplification region. For a complete supermoduel this results in 30 positive and 30 negative HVs. Since each voltage should be controable individually, 2×30 HV channels are required per supermodule.

The high voltages are provided by modules produced by the company ISEG [ISE]. Each HV module provides 32 HV channels with negative voltage (module type EDS 20 025n_504) or positive voltage (module type EDS 20 025p_203). Therefore each supermodule needs one EDS 20 025n_504 and one EDS 20 025p_203 HV module for operation, resulting in 18×2 HV modules for the complete TRD. The HV modules are located in crates outside the L3 magnet of ALICE. The crates are connected with PCs which can control each channel separately.

The HV modules and the supermodules are connected with HV multiconductor cables. One cable connects one HV module with the corresponding connection on one supermodule. Each HV multiconductor cable has 37 leads. 30 leads are used for the HV supply and the other leads are used as current return lines. Inside the supermodule the HV is distributed to the chambers via individual high-voltage cables.

Gas Supply

The TRD needs a mixture of xenon and carbon dioxide for operation. The gas system of the TRD is a closed loop system because of the high price of xenon. One challenge during the design of the gas system was that xenon is a heavy gas and just the height difference of 7 m between the lowest and the heighest readout chamber in the TRD results in a pressure difference of 2.5 mbar. But the overpressure in the chambers has to be limited to be below 1 mbar to avoid damage. To solve this problem, the gas system is segmented in 14 height sections. Gas pipes are used between the gas distribution modules – located halfway between the TRD 100 m below ground and the surface – and the detector to provide a constant gas flow to all supermodules [Lip06].

Each supermodule has three gas inlets in stack 0 of layers 0, 2, and 4. The gas flows through the layer until it reaches the end of the chambers in stack 4. There it is redirected to the next upper layer and flows back to the chamber in stack 0. Therefore the gas outlets are at stack 0 of the layers 1, 3, and 5. There the gas is collected and sent to the gas recovery. The gas recovery first separates the CO_2 with membranes. Afterwards oxygen is removed and then the xenon gas is recovered in a cryogenic xenon recovery plant [Lip06].

The gas quality is monitored by the GOOFIE system [DA07]. The GOOFIE system measures permanently the drift velocity and the gas gain. Based on these measurements different gas parameters can be monitored, including the actual level of nitrogen contamination. These information are important parameters for the analysis of the data recorded by the TRD.

4.3 Front End Readout Electronics of the TRD

The front end readout electronics (FERO) of the TRD includes all electronics which is mounted on the top of the readout chambers. The main components are the multi-chip-modules (MCMs) which read out the cathode pads, digitize and preprocess the data, the optical readout interface cards (ORIs) which send the data from the readout chambers to the trigger and the data acquisition system and the detector control system boards (DCS boards) which configure and control the MCMs, the ORIs and the readout boards.

This section gives an overview of the used hardware and the communication networks in the FERO whereas the next chapter describes the control software used to operate the FERO.

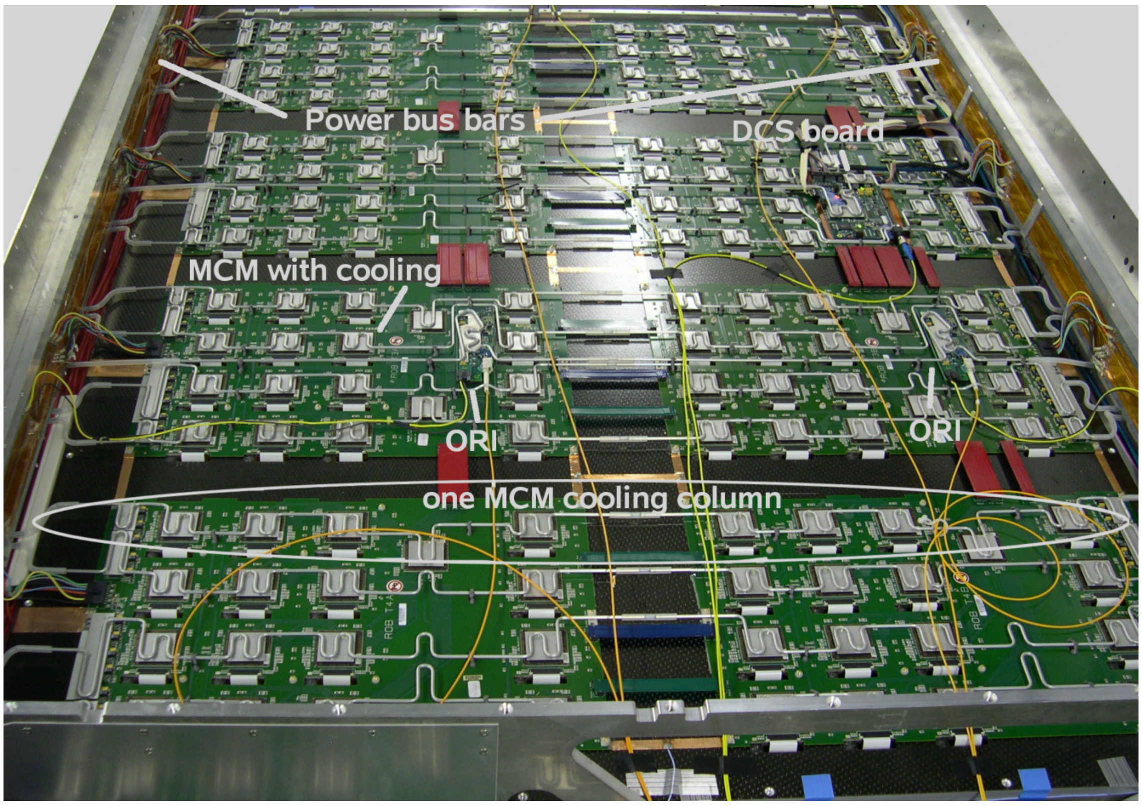


Figure 4.6: Overview of a readout chamber installed in the supermodule. The power bus bars provides the low voltage for the electronics and the cooling keeps the electronics within their safe operating temperatures.

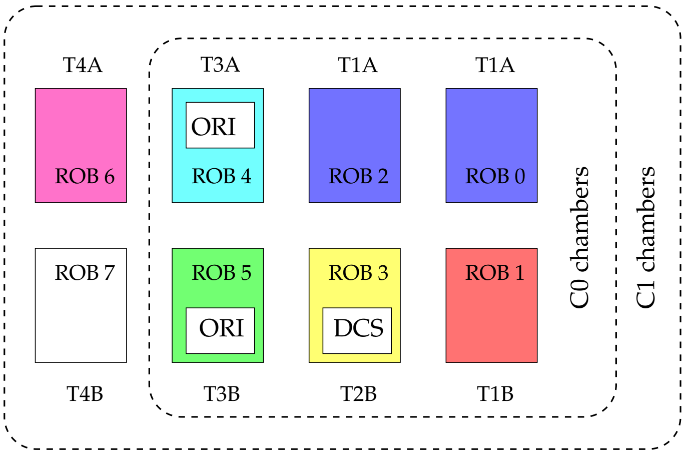


Figure 4.7: The two chamber types used in the TRD and the position of the different readout boards. Each readout board has at least 17 MCMs soldered on top but only the T2B type has a connector for a DCS board and only the two T3 types have a connector for ORI cards (drawing based on [RS06]).

4.3.1 Readout Boards

The readout boards (ROBs) are the boards for the chamber electronics. They have circuits for the power supply of the MCMs and the ORIs, for the communication between the MCMs and the DCS board (SCSN Bus, section 4.3.5) and for data transfer from the MCMs to the ORIs (the Readout Tree, section 4.3.6). Every readout board has a size of 30 cm $\times$ 46 cm.

Soldered on each readout board are power regulators, supplementary electronics, a few connectors and at least 17 MCMs. The power regulators provide 1.8 V and 3.3 V for the MCMs and can be switched on and off by the software running on the DCS board.

There are seven different types of readout boards: T1A, T1B, T2B, T3A, T3B, T4A and T4B. The types T1A, T1B, T4A and T4B are essentially the same. They only differ in the position of connectors. All these types have 17 MCMs and no special connectors. The type T2B has 17 MCMs, too. Additionally, this type has a special connector for the DCS board. The types T3A and T3B have a special connector where the ORI is plugged in. In contrast to all other types, both types have one additional MCM, called half camber merger. In total, a readout chamber with eight boards has one readout board of type T1B, T2B, T3A, T3B, T4A, T4B and two readout boards of type T1A each. On a six board chamber the T4A and the T4B boards are missing. The position of the different components are shown in figure 4.7.

The connections between the ROBs are done via bridges which are plugged in to connectors on the ROBs.

4.3.2 Multichip Modules

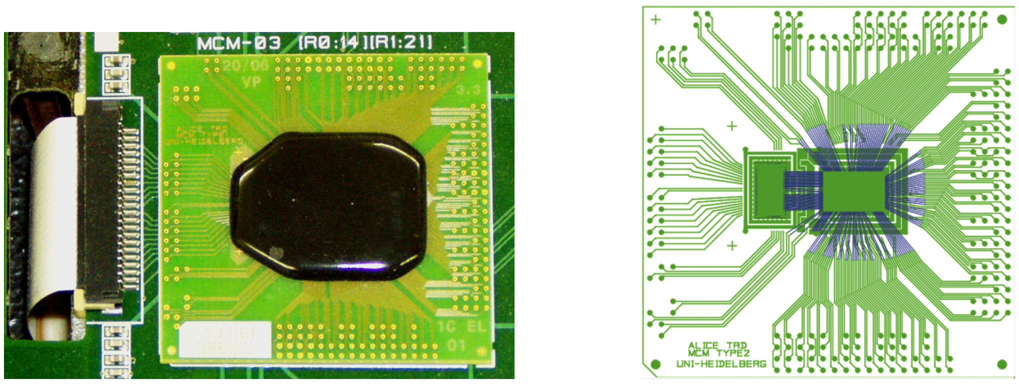


Figure 4.8: The left figure shows a picture of an MCM (black) with its plate (light green) and the data cable (white), which connects the MCM with the pad plane. The right figure shows the internal structure of the MCM and its plate. The left rectangel is the PASA chip and the right rectangle is the TRAP chip of the MCM. The blue lines are thin soldered wires which connects on the one hand the PASA chip with the TRAP chip and on the other hand the two chips with the lines to the vias. The bue lines show only some of these wires. The vias are the green dots at the edge of the MCM. In reality the vias have a golden color (see left figure).

The Multichip Modules (MCMs) are soldered to the top of the readout boards. They have two tasks. First they read out the induced charge on the cathode pads, digitize and filter the data. Each MCM is capable to read out 18 pads every 100 ns. Second the MCMs calculate tracklets. A tracklet is a line which parameterizes the path of a particle through the chamber. The tracklet information are shipped to the GTU. The GTU uses the tracklet parameters to reconstruct the path of the particles inside the TRD. Based on the track information one can define various trigger conditions.

One MCM contains two different chips (see figure 4.8). The first one is a pure analog chip with 18 shaping preamplifiers (PASA) and the second one (TRAP chip) contains ADCs and the complete digital part of the MCM. The chips are connected via chip-to-chip bonding.

Each preamplifier is connected to one readout pad of the chamber. The preamplifiers shape the signal and increase the amplitude to use the full input range of the ADCs. To avoid border effects between the MCMs and for the tracklet calculation some overlap between the pads each MCM processes is needed. Therefore, three preamplifiers of each MCM are read out by the neighboring MCMs, too (see fig. 4.9). This results in a total number of 21 ADCs per MCM. The readout scheme for the pad plane is shown in figure 4.9. The ADCs are 10 bit ADCs with a digitization rate of 10 MHz and a default input range of ±1 V [Ang06]. By using attenuation factors it is possible to measure signals larger then the default input range.

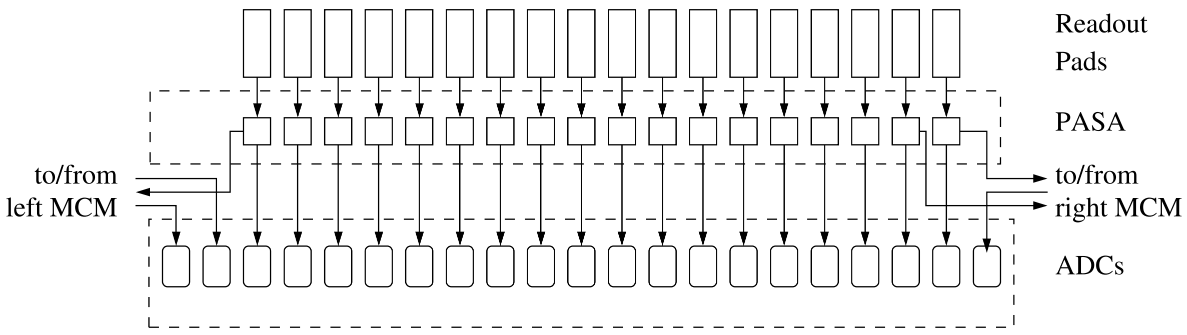


Figure 4.9: Readout scheme of the pad plane (based on [Ang06])

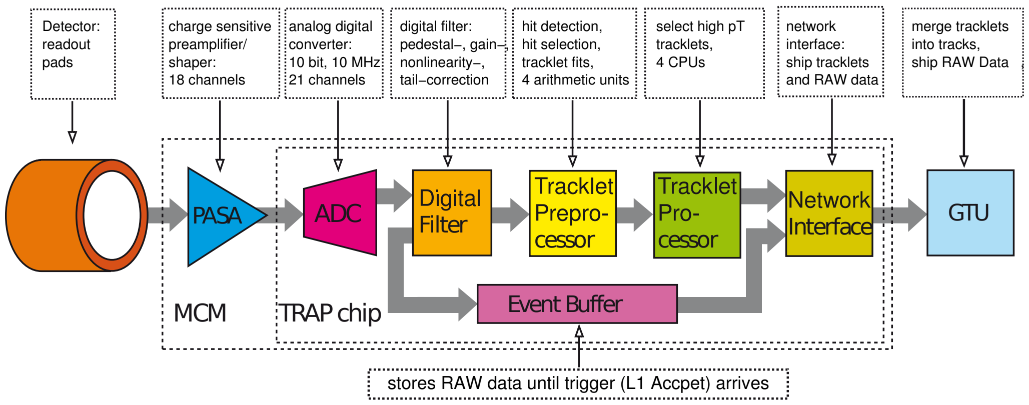

Detector: readout pads
charge sensitive preamplifier/ shaper: 18 channels
analog digital converter: 10 bit, 10 MHz 21 channels
digital filter: pedestal–, gain–, nonlinearity–, tail–correction
hit detection, hit selection, tracklet fits, 4 arithmetic units
select high pT tracklets, 4 CPUs
network interface: ship tracklets and RAW data
merge tracklets into tracks, ship RAW Data
PASA
ADC
Digital Filter
Tracklet Prepro-cessor
Tracklet Pro-cessor
Network Interface
GTU
Event Buffer
MCM
TRAP chip
stores RAW data until trigger (L1 Accpet) arrives

Figure 4.10: Scheme of the data processing in the MCMs

Figure 4.10 shows the data processing in the MCM. After the ADCs have digitized the PASA output, the digitized signal can be processed by five different filters. Each channel has its dedicated set of filters and filter parameters. All filters have a bypass and can be switched on and off separately. The five filters are:

1) Nonlinearity Correction This filter corrects the nonlinearities of the signal caused by the shaping preamplifier.

2) Pedestal Correction The pedestal correction determines the individual baseline of each channel, subtracts them and adds a common baseline.

3) Gain Correction The gain correction scales the signal within a range of $\pm 12\,\%$. This filter compensates the slightly different amplification factors of the PASAs and differences between the ADCs within a ROC.

4) Tail Cancellation The tail cancellation suppresses the signal from the ion tail in the amplification region. After each amplification process the created ions lead to a fast increasing signal. The ions drift slowly away from the cathode pads, resulting in a slowly decreasing signal. When the next avalanche occurs there is still a residual signal. The tail cancellation calculates this residual signal and subtracts it.

5) Crosstalk suppression The signals from two neighboring pads are not independent. Due to their capacity a charge induced in one pad also affects the neighbor pads. The crosstalk suppression estimates this effect and removes it from the signal. However, this filter will not be used in the experiment because otherwise the power consumption of the MCMs would be to high.

The ADCs and filters are running continously and their output is stored in a pipeline. When a pretrigger signal arrives, the contents of this pipeline and the following samples are saved in a memory called event buffer. Each chain has its own event buffer which can store 64 samples (10 bit data + 1 parity bit per sample). At the same time the data are sent to the tracklet preprocessor (TPP). The preprocessor detects and calculates the position of hits on the pad plane and provides some parameters for the tracklet calculation. Since the preprocessor has four arithmetic units it can handle up to four hits in each timebin. If there are more then four hits the four hits with the biggest deposited charge are selected.

After the end of the drift time, or more precisely, after a configurable number of samples have been taken, the four RISC CPUs of the tracklet processor are started. During the data acquisition they were switched off to reduce noise. The four CPUs use the parameters determined by the TPP to calculate tracklets. Each processor can calculate one tracklet. If there are more then four tracklet candidates, the four candidates with the highest total charge are selected. After the tracklet calculation is finished, the tracklet parameters (slope and offset of the line representing the particle path and the electron probability) are sent via the readout tree to the ORIs. Compared to the size of raw data the tracklets are small. Per tracklet only one 32 bit word is needed.

If the trigger system decides that the event is interesting, it sends another trigger signal, the L1 signal, to the readout chambers. In this case the MCMs send the tracklet information and the content of the event buffer where the filtered raw data were stored. To decrease the data volume the MCMs support zero suppression. Only the raw data of channels which fulfil configurable conditions like the channel being part of a tracklet or the total signal sum is above a threshold are send.

The MCMs have no nonvolatile memory. Therefore the complete configuration has to be transmitted to the MCMs after each power up. This includes calibration values for the ADCs, the filter settings and settings for the tracklet preprocessor. The settings are transmitted to the MCMs via the SCSN bus and written to the configuration registers of the TRAP chip. The TRAP chip has 434 configuration registers in total with sizes between 1 and 32 bits [A$^+$05]. Not all registers are used to store configuration values. Writing to some registers initiates special actions of the TRAP like reading the internal temperature sensor (see chapter 6) whereas other registers contain counters or status information of the chip. A complete list of all registers, their individual size and their function is given in the TRAP user manual [A$^+$05].

The four tracklet processors which do the tracklet calculation are freely programmable CPUs. Each CPU has an instruction memory (IMEM) of 4096 times 24 bit (4096 'words') which contains the program for the CPU. Just like the configuration, the programs for the CPUs have to be transmitted to the MCMs via the SCSN Bus. The current programs have a size of 1532 words for the first CPU, 1665 words for the second CPU, 1367 words for the third CPU and 1412 words for the forth CPU. Most parts of the four CPU programs are identical. The different sizes of the configurations are caused by the fact that the programs do not only contain the routines for the tracklet calculation but test routines and other special functions, too. Since there is space left in the IMEM the programs can be extended in the future.

4.3.3 The Optical Readout Interface

The Optical Readout Interface (ORI) is an optical transmitter card which transfers the data from the MCMs to the global tracking unit (GTU). The ORI receives the data from the half chamber merger (see 4.3.6), converts and serializes the data and sends it via an optical fiber at 2.5 Gbit/s to the GTU.

The four main components of an ORI are a CPLD chip (Complex Programmable Logic Device) which provides the interface between the half chamber merger and a serializer chip, the serializer chip, an optical driver chip for the laser diode and the diode itself.

The ORI has a JTAG and an I^{2}C interface. The JTAG interface provides access to the CPLD chip. It can be used to reprogram the CPLD and to access some configuration registers in the CPLD. The I^{2}C interface is used to program and control the laser driver chip [LARP06].

Two MCMs on each ROB equipped with an ORI support a special operation mode. In this mode commands addressed to these MCMs are forwarded via JTAG or I^{2}C to the ORI. Data from the ORI are received by the MCMs and stored in registers of the MCMs where they can be read out via SCSN. Using these MCMs, programs running on the DCS board have access to the CPLD and the optical driver chip on the ORI. To avoid access conflicts both MCMs must not be in the special operation mode at the same time.

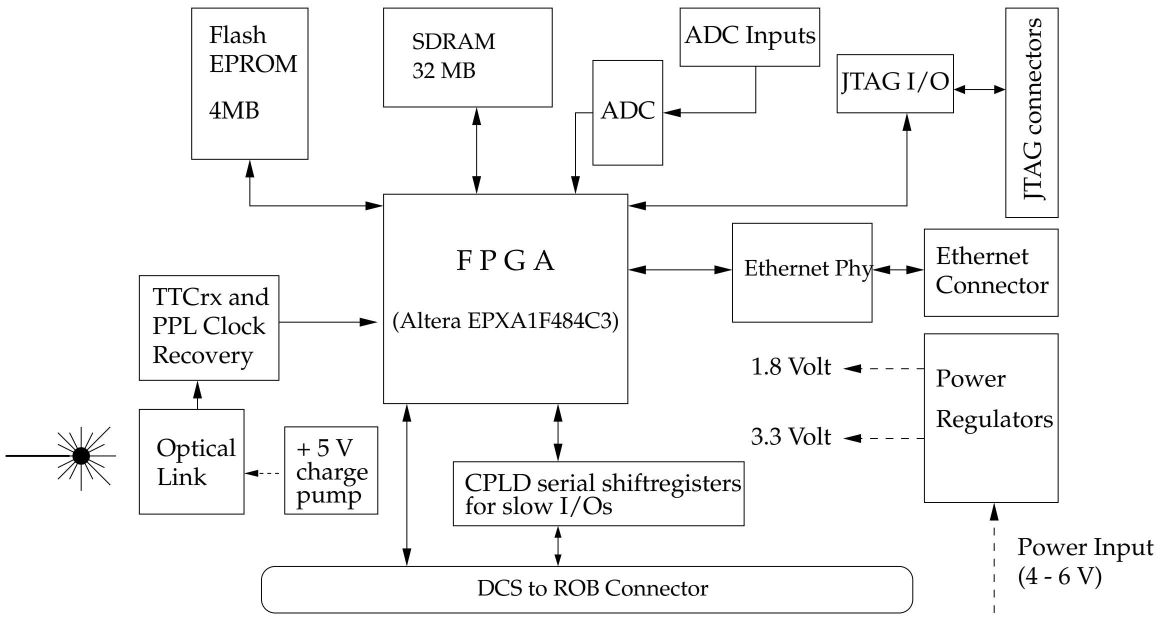


Figure 4.11: Schematic diagram of the DCS board. The diagram shows the main components, their approximate position on the DCS board and the communication between the components (based on [G$^+$]).

4.3.4 The DCS Board

All the electronics on a readout chamber need to be configured and controlled. Furthermore, different environmental conditions like supplied voltages or temperature in the supermodule have to be monitored. All these tasks are performed by the DCS board.

The DCS board is a small autonomous single board computer which is used by the TRD, TPC and the PHOS detector. In case of TRD it is mounted directly at the top of the T2B readout boards.

The DCS board has a full custom design and was mainly developed at the university of Heidelberg. Its central processing unit is an Altera Excalibur FPGA with an embedded ARM processor core. The FPGA provides the necessary flexibility to use the DCS board for different purposes. For example the SCSN Bus interface is implemented completely in the FPGA. Therefore no dedicated chips are needed which would be useless in the other detectors. The configuration for the FPGA and a small Linux operating system is stored in a 8 MB Flash ROM. The DCS board loads automatically the FPGA configuration and boots the Linux system after powering up. In case of failure in the FPGA or the Linux system mutual reconfiguration of the boards is feasible via a JTAG chain. Each DCS board has two JTAG connectors, one master and one slave connector. In case of a failure one can log in on the corresponding master DCS board of the damaged DCS board and starts reconfiguration. Further elements of the DCS board are a 32 MB SDRAM chip, a 10 MBit Ethernet PHY transceiver chip, an optical interface for receiving trigger signals and an ADC with a multiplexer for the digitization of analog sensor values. [GKLT05]

At the DCS boards on the readout chambers of the TRD, the ADC is used to monitor a NTC resistor for temperature measurements and to measure the supply voltages for the readout boards. A schematic diagram of the DCS board is shown in figure 4.11.

Using Linux as operating system has some advantages. It simplifies the software development since the usual GNU tools (compiler, debugger and libraries) can be used. Furthermore the Linux system maps the hardware interfaces to character or block devices which can be used like normal files. Programs can use the standard C/C++ procedures for reading and writing files to access the SCSN Bus or the ADC for example, without needing special libraries.

However, the processing power and the memory of the DCS board are very limited. The Linux system including all user programs has to fit in 4 MB since the other Flash ROM space is needed for the FPGA configuration. Therefore the normal Linux tools cannot be installed on the DCS board. Instead BusyBox [BBO] is is used. BusyBox is a single executable which provides stripped down versions of usual Linux programs like ls, cp, grep, tar and many others. As C library the uClibc library [UCL] is used. The uClibc is designed to be a C library for embedded Linux. It provides most of the features of the standard glibc but needs much less space then the glibc. For C++ programs the library libstdc++ is installed on the DCS board.

Another consequence of the limited resources is that programs cannot be compiled on the DCS board itself. All software including the Linux system itself are compiled on a normal PC with an ARM cross-compiler and copied to the DCS board afterwards. First time, the Linux system has to be copied via the JTAG connection to the DCS board. Once the Linux system is running it is also possible to copy a new Linux version via the ethernet interface to the DCS board.

The Linux system provides an ssh server which is accessible via the ethernet interface. Operators can log in to the DCS board using the normal ssh program and execute commands on the DCS board or copy files with scp to the DCS board. This manual access is used during development, for debugging and to update software on the DCS board.

In normal operation the DCS boards are controlled by the detector control system (section 3.3). The DCS boards mounted on the ROCs are powered up by the power control unit (PCU) (see section 4.2), the DCS board boots the Linux system and the Linux system starts the FEEServer. The FEEServer is a Linux program running on the DCS board which controls the chamber. It will be described in chapter 5.

4.3.5 The SCSN Bus

The Slow Control Serial Network[2] (SCSN) is the dedicated network for the communication between the DCS board and the MCMs on one ROC. The SCSN bus [Gar02] was developed for the TRD but the network design is general. It can be used in other systems, too.

The SCSN bus is a full-duplex daisy-chain network which uses LVDS[3] for data transmission. The DCS board acts as the bus master and the MCMs are the bus slaves. The SCSN bus supports up to 126 slaves controlled by one master. The data between master and slaves are exchanged in fixed size packets called frames. The structure of the frames is shown in figure 4.13 and listing 4.1. Each frame starts at the bus master and terminates there. The slaves only forward or alter the frames but cannot create frames on their own. Therefore a direct communication between slaves is not possible.

[2]'slow' means that the network is not used for fast data readout.

[3]Low-voltage differential signaling, or LVDS, is a differential signaling system. It uses a difference in voltage between two wires to encode the information.

Compared to a star like network, a daisy-chain network has some advantages and some disadvantages. The main advantage and the reason why this network topology was chosen is the much lower number of connections on the bus master (DCS board). Instead of connecting each slave with the master separately, in a daisy-chain network only connections between the master and the first slave and between the master and the last slave are required. A second advantage is the shorter length of the lines. Typically the distances between two neighboring slaves are shorter than between slave and the master.

The big drawback is missing reliability. As soon as a line or a slave breaks the complete network is dead. To reduce this problem the SCSN bus uses not one ring like normal daisy chain networks but two independent rings. In the case of a failure, the full-duplex network can be split in two half-duplex networks, excluding the damaged part. The two slaves which are nearest to the damaged part are set to the so-called bridging mode by a special command from the master. Instead of forwarding frames to the next slave on the same ring, a slave in bridged mode sends the frame back on the other ring (see figure 4.12).

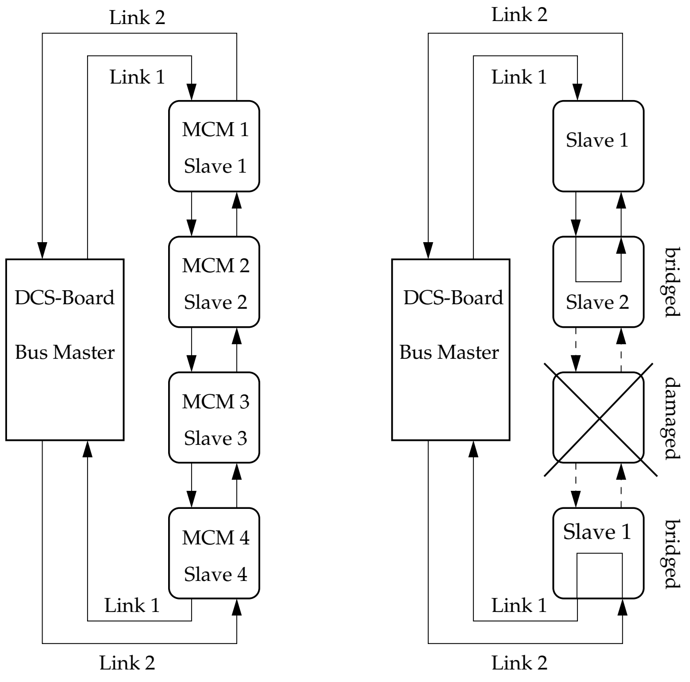


Figure 4.12: On the left picture a normal SCSN bus with four slaves (MCM 1-4) and a bus master (the DCS board) is shown. All frames are sent via link 1. Link 2 is not used. In the right picture MCM 3 is broken. It cannot process or forward frames. Therefore both neighbor slaves (MCMs 4 and 2) are set to bridge mode. In this mode they send the received frames back on the other link. The lines to MCM 3 are not used any more (visualized by the dotted arrows). MCM 1 is not affected and remains in normal mode. As a consequence the address schema changes. Since MCM 4 now gets the frames via link 2 it becomes slave 1. If there were more MCMs after the broken one they also would get new slave addresses.

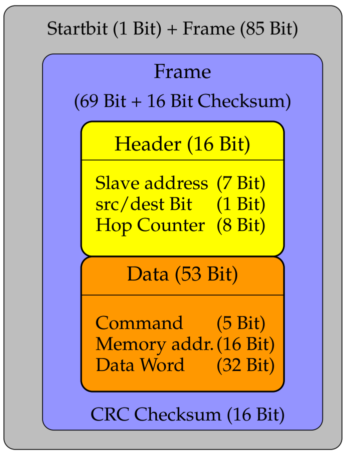


Figure 4.13: Structure of a SCSN Bus frame. The two outermost parts are added by the SCSN driver on the DCS board before sending and are removed after receiving a frame. Therefore the data structure representing a frame in the control software running on the DCS board only contains the header and data fields (drawing based on [Gar02]).

```
typedef struct {
  unsigned int mcmaddress;   // slave address of the target MCM
  unsigned int source_bit;   // source bit, must be set to 1
  unsigned int hopcounter;   // hopcounter, must be set to 0
  unsigned int command;      // Command to be executed within the
     MCM
  unsigned int address;      // Memory address in the target MCM
  int data;                  // Payload (data for the MCM)
} FrameSend;

typedef struct {
  unsigned int error_flag;   // error flag
  FrameSend the_frame;       // the received frame
} FrameReceived;
```

Listing 4.1: C/C++ structure representing a frame. Before sending the frame the SCSN driver calculates and adds the CRC sum and the start bit (compare fig. 4.13). When the frame has returned the driver checks the CRC sum and sets the error flag if the CRC sum does not match.

The addressing scheme is quite simple. Each frame contains an address field and a hop counter (see figure 4.13 and listing 4.1). A slave receives a frame either from the bus master or from another slave. It compares the address field with the hopcounter field. If both values are not equal or the src/dest bit is set to zero it just increases the hop counter by one and forwards the frame to the next slave. If the values are equal and the src/dest bit is set the slave processes the information in the data section of the frame. After processing, the slave sets the hop counter and the src/dest bit to zero and forwards the frame to the next slave.

In case of TRD quite often all MCMs in the bus should process the same data frame. Therefore the SCSN bus supports broadcasts. If a slave receives a frame with an address field set to 127 and a src/dest bit of one, it processes the information in the data section.

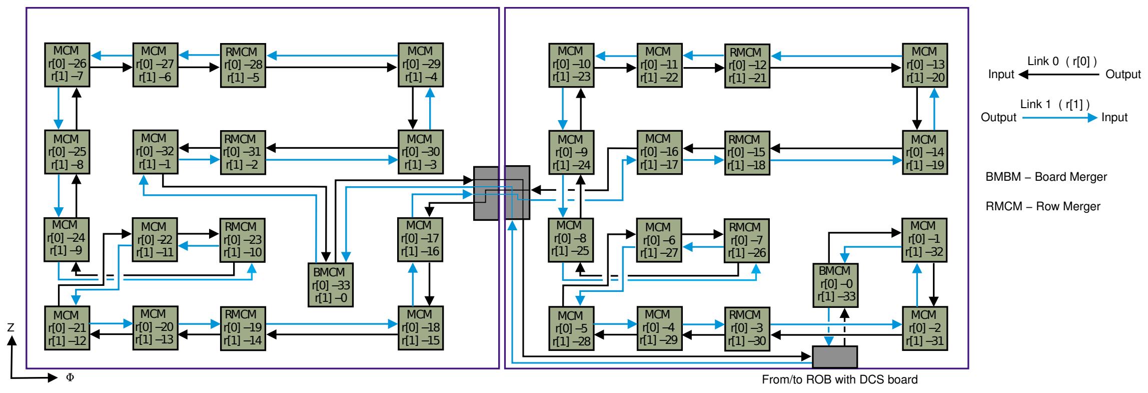


Figure 4.14: Cabling of the MCMs in linkpair one. The black lines represent the cabling of link zero and the blue lines represent the cabling of link one of the linkpair.

Afterwards only the hop counter is incremented by one and the frame is forwarded to the next slave. Neither the hop counter nor the src/dest bit is set to zero.

In a bridged bus, things become more complicated. Due to the bridging the addressing scheme for some slaves changes. The slaves 'after' the damaged part get the frames via the other link. Therefore the last MCM in the unbridged bus is now slave one, the second last is slave two and so on. The slaves before the damaged part get the frames via the same link as in an unbridged bus. Here the addresses remain the same. The first MCM is slave one, the second is slave two, etc (see figure 4.12). The master has to know to which part of the now split network a slave belongs and what its new address is. In a bridged network also the src/dest bit becomes very important. Since the frames travel the same way back they were forwarded broadcast commands would be executed two times. Therefore slaves in bridged mode always sets the src/dest bit to zero. Frames with a zero src/dest bit are never processed by a slave even if it is a broadcast frame or destination and hopcounter do match.

On a ROC there are three (on ROCs of type C0) or four (ROC type C1) independent SCSN Bus system called linkpairs. Each linkpair connects the MCMs of two ROBs. The MCMs on the ROBs T1A and T1B belong to linkpair zero, the MCMs on the second T1A type ROB and on the T2B ROB belong to linkpair one and so on. The DCS board is the bus master for all linkpairs. But it can communicate only with one linkpair at the same time. The linkpair to which the DCS board is connected at a time is selected by the software running on the DCS board.

To address MCMs on a ROC one can use the linkpair where the MCM belongs to and its position in this linkpair, called slave id. But this numbering causes two problems. First the cabling scheme and therefore the slave numbering scheme is not intuitive (see figure 4.14). Second the numbering depends on the bridging status of the bus. Therefore the AliceID was introduced as a second numbering scheme to identify MCMs on a ROC. The AliceID reflects the position of the MCM on the readout chamber instead of its position in the linkpair. For the AliceID an eleven bit address is used. The lowest seven bits of the AliceID identifies the MCM or the MCM group on the readout board (for numbering scheme see figure 4.14) and the four uppermost bits identify the readout board or a group of readout boards. Table 4.1 gives an complete overview of the available MCM groups.

The AliceID is used in the configurations and inside the control engine to identify the MCMs. However, to access an MCM still the linkpair number and the current slave id is needed. This conversion is done by a class in the control engine when the SCSN frames are created (see section 5.3).

AliceID (bit pattern)	addressed MCMs / ROBs
xxxx0yyyyyy	MCM with slave ID yyyyyy, e.g. xxxx0000110 corresponds to MCM 6 on ROB xxxx
xxxx1000001	all column merger MCMs
xxxx1000010	all board merger MCMs
xxxx1000100	half chamber merger on C0 chamber
xxxx1001000	half chamber merger on C1 chamber
xxxx1010000	all MCMs at the edge of the ROC (depends on side)
xxxx1100000	all MCMs except for edge and column mergers chips
xxxx1111111	all MCMs (broadcast)
1000xxxxxxx	MCMs on ROB 0 (T1A)
1001xxxxxxx	MCMs on ROB 1 (T1B)
1010xxxxxxx	MCMs on ROB 2 (T1A)
1011xxxxxxx	MCMs on ROB 3 (T2B)
1100xxxxxxx	MCMs on ROB 4 (T3A)
1101xxxxxxx	MCMs on ROB 5 (T3B)
1110xxxxxxx	MCMs on ROB 6 (T4A)
1111xxxxxxx	MCMs on ROB 7 (T4B)
0001xxxxxxx	ROBs on A side (ROBs 0, 2, 4, 6)
0010xxxxxxx	ROBs on B side (ROBs 1, 3, 5, 7)
0100xxxxxxx	ROB 4 and 5 (T3A, T3B)
0000xxxxxxx	all ROB.

Table 4.1: Composition of the AliceID. To get a complete AliceID one has to combine one entry from the upper part of the table with one entry from the lower part of the table, e.g. 110100000110 addresses MCM 6 on ROB 5. Meaningless combinations are ignored. The tasks of the different MCMs (column merger, bord merger, half chamber merger) are explained in section 4.3.6 and the positions of the ROBs are shown in figure 4.7 on page 31.

Measurements

Figure 4.15 shows the signal on both LVDS lines between two MCMs during the transmission of a ping frame. A ping frame is a frame which is just forwarded from MCM to MCM without initiating any further action. This ping frame shows an important feature of the SCSN Bus. Although the size of a frame is fixed to 85 bit (compare figure 4.13) the number of transmitted bits can be bigger. If a frame contains more then 8 equal bits, stuff bits are inserted in the frame. The stuff bits have the opposite value to the sequence of equal bits. The ping frame shown in in figure 4.15 has a long sequence of bits with value 0 (between 0.9 µs and 3.1 µs). Therefore 5 stuff bits with value 1 were inserted, resulting in a total frame size of 90 bits. The stuff bits are necessary because the bit transitions in

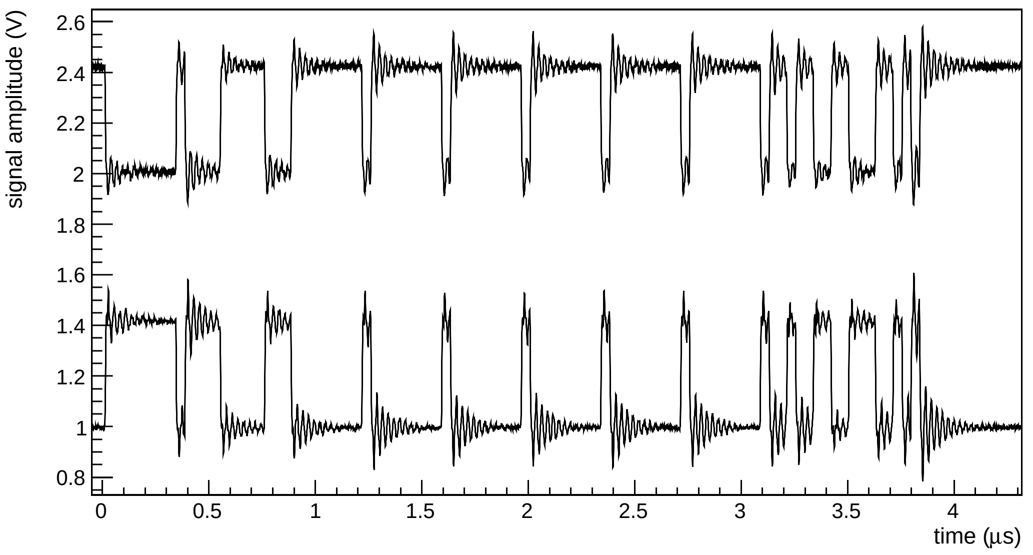


Figure 4.15: A ping frame measured during its transmissionn between two MCMs. The curves show the signal on both LVDS lines. For the plot the upper curve was shifted by +1 V to separate the two signals.

the frames are used to synchronize the internal clocks of the MCMs. Furthermore, long series of 0 bits are interpreted as signal timeouts and cause transmission errors.

The clock of the SCSN network runs at 72 MHz and the transmission speed is set to one third of the network clock, resulting in a transmission speed of 24 MBits/s [Gar02]. Therefore one expects for a frame with 90 bits a transmission time of 3.75 µs which fits very well with the result shown in figure 4.15.

In figure 4.16 one of the two input and output LVDS lines of an MCM were measured respectively. For this measurement a read frame was used. A read frame causes the MCM to read the CPU register set in the memory address field of the frame and to store the read value in the data section of the frame. Therefore the input and output signal differs significantly. One can see that the MCM needs about 0.14 µs to process the frame. Same measurements were done with other frame types (ping, write, bridge frames - for further explanation of the frame types see appendix B and [A^+05]). It turned out that the processing time in the MCM does not depend on the frame type within the measurement uncertainty.

In a third measurement the total transmission time for a complete linkpair with 34 MCMs was determined. A series of ping frames was send by the default sending routine of the control engine through the linkpair and the signals at the input line of the first MCM and at the output line of the last MCM were measured. Figure 4.17 shows the result. Each frame needs about 135 µs to travel through the linkpair and there can be eight frames in the linkpair at the same time. The exact travel time for an SCSN frame varies slightly, depending on the current status of the arbiters in the MCMs. The reason for the big gap between the first and the second frame is unknown. Neglecting this gap

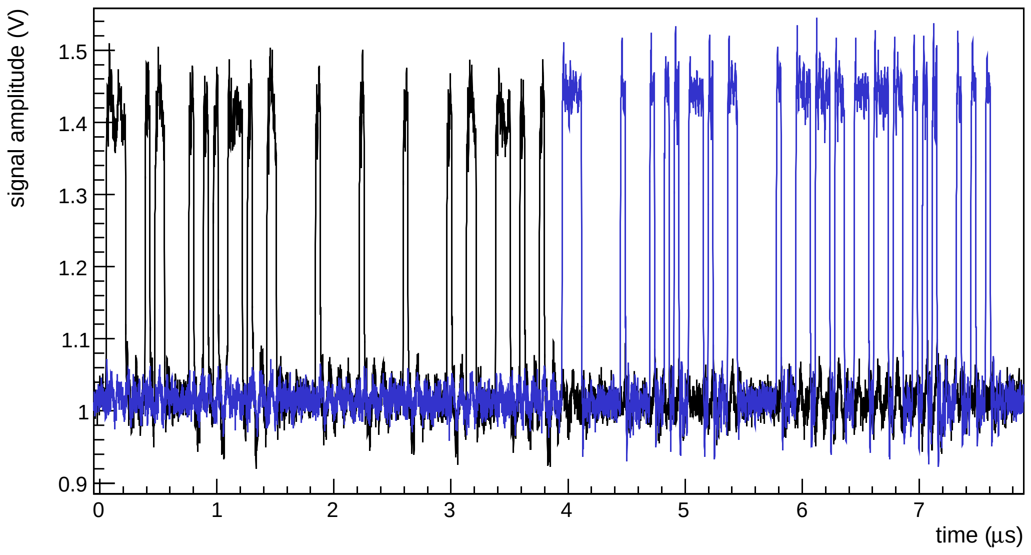


Figure 4.16: Signal of an incoming (black) and outgoing (blue) read frame on an MCM. The outgoing frame differs clearly from the incomming frame since the outgoing frame contains the read data.

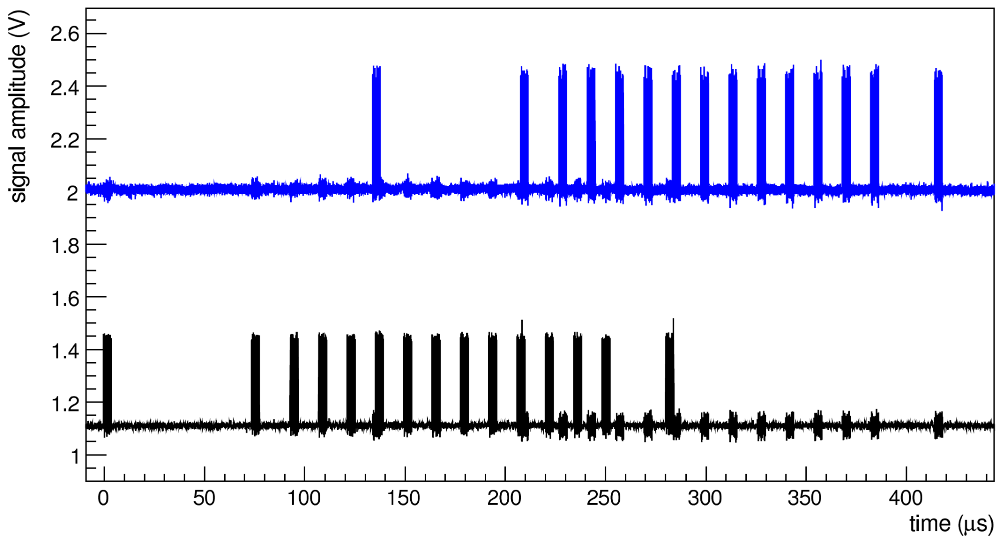


Figure 4.17: A series of ping frames measured at the input line of the first MCM (black) and at the output line of the last MCM (blue) of a linkpair with 34 MCMs. For the plot the upper curve was shifted by +1 V to separate the two signals.

the transmission speed of the SCSN bus is about

$$\frac{1}{135\,\mu\mathrm{s}} \cdot 8\ \mathrm{frames} \approx 60.000\,\frac{\mathrm{frames}}{\mathrm{s}} \tag{4.5}$$

or 5.3 MBits/s.

4.3.6 The Readout Tree

The Readout Tree is a second network on the ROC. While the SCSN bus is used to control the MCMs, the Readout Tree transfers the tracklet data and the raw data from the MCMs to the ORIs. Furthermore it is used to distribute trigger signals. It operates at 120 MHz with an effective bandwidth of 240 MBytes/s. Each connection in the Readout Tree consists of 10 lines. Eight lines are used for data transfer, one line transfers parity bits and one is a spare line [Ang06].

The Readout Tree is designed as a hierarchical network with four levels of hierarchy. On the lowest level of hierarchy the normal MCMs are located. They do the data acquisition and the tracklet calculations. Each readout board has twelve normal MCMs. The next level contains the column mergers. Each readout board has four column mergers. These MCMs do the same as the normal MCMs but they have an additional function. They collect data sent by the three other MCMs in their column and forward them together with their own data to the board merger. Each readout board has one board merger. The board mergers do no data acquisition but collect the data from the four column mergers and forward them to the half chamber merger. Only a readout board with an ORI (types T3A and T3B) has one half chamber merger. Like the board mergers a half chamber merger does no data acquisition. It collects the data from the three or four board mergers (depending on the chamber type) of one half of the chamber and forwards the data to the ORI.

The readout tree distributes the trigger signals, too. Each time a trigger signal arrives at the DCS board via the TTC fiber[4] or is generated on the DCS board by software, the DCS board sends a signal via a control line to the two half chamber mergers on the ROC. Afterwards the trigger signal is distributed via the readout tree exactly the other way around to the data flow.

4.4 Front End Readout Electronics Control System

The general control system of ALICE has been described in section 3.3. This section gives an overview of the control system for the front end readout electronics (FERO) of the TRD and how it is integrated in the ALICE control system.

In normal operation mode the FERO control system runs as a part of the detector control system (section 3.3.2). It receives transition commands for its FSM from the detector control system and reports the resulting state changes. Like the other DCS subsystems, the FERO control system is organized in the three layers: supervisory layer, control layer and field layer. The structure of the system and the control flow is illustrated in figure 4.18.

[4]The TTC fiber is an optical fiber which connects the DCS board with the trigger system. The DCS board only receives trigger signals via this fiber and cannot send anything using this fiber.

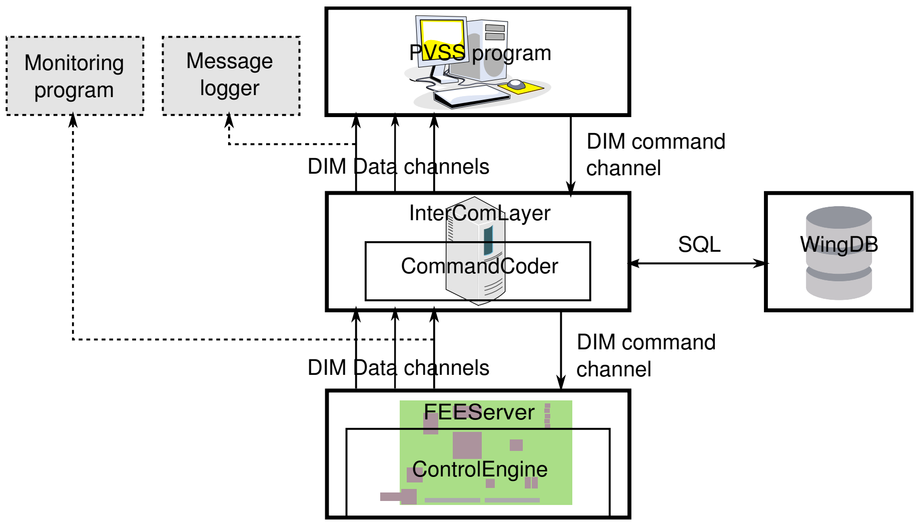


Figure 4.18: The TRD front end readout electronics control system. The figure shows the four main components (PVSS, ICL, wingDB and FEEServer) and two examples of additional monitoring software.

At the top level in the supervisory layer a PVSS system is located. It controls the FERO and is the interface for the DCS system. Alternatively, the PVSS system can be run stand alone, for example to test or debug the FERO system. For this case the PVSS system provides an user interface to operate the system manually.

From the supervisory layer point of view the TRD front end electronics is controlled by the means of tags. A tag is just an integer which identifies a transition in the finite state machines of the front end electronics (see section 5.2.2) along with a configuration for the electronics. Every time a readout chamber (ROC) should get a new configuration / switch to a new FSM state, PVSS sends the corresponding tag and the identification string of the ROC to the InterComLayer (ICL) in the control layer. The ICL has to translate the tag to a configuration for that ROC. However, this functionality is not implemented in the ICL itself but encapsulated in a separate library called CommandCoder. To translate the tag to the corresponding configuration the CommandCoder queries the database wingDB.

The wingDB contains command sequences and general information about the chambers like the chamber type or information about damaged MCMs. For each tag and for each ROC to be configured the database is queried separately.

The CommandCoder collects all the command sequences and information and merges these data in a large data structure – the configuration. Finally the ICL sends the configuration to the FEEServer (see chapter 5) running on the DCS board of the ROC.

Figure 4.19 shows the data structure of the configuration. The configuration is subdivided in different parts. It always starts with the cfdat_header.

The cfdat_header contains some general information about the configuration and offsets to the four main sections in the configuration (error, rocinfo, basecfg, tempcalib). One important field in cfdat_header is hd_maker. This field contains the string identifying the transition in the FEEServer FSM belonging to the configuration. The offsets are used to locate the beginnings of the four section. Technically, a program which wants to access

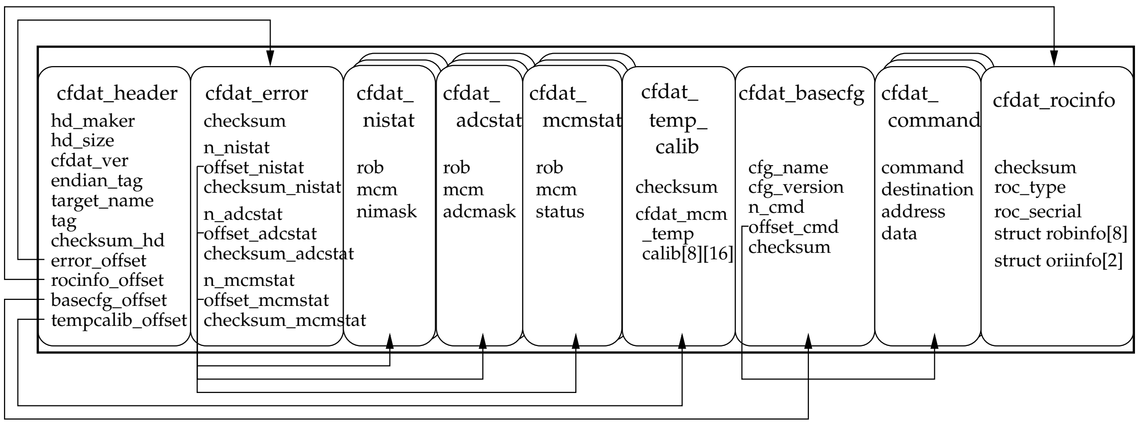


Figure 4.19: Data structure of the configuration composed by the Command Coder.

one of the sections takes the pointer to the beginning of the complete configuration and just adds the offset value to the pointer. The resulting pointer points to the beginning of the selected section. The advantage of this approach is flexibility. The size of the different sections can vary and it is even possible to omit a complete section. If a section is omitted the corresponding offset value is set to zero.

Section cfdat_error contains offsets to lists of three different types of hardware error in the MCMs: errors in the MCM network interface (cfdat_nistat), damaged ADCs (cfdat_adcstat) and overall MCM damages (cfdat_mcmstat). Each damaged MCM gets an own entry for each error type. Since the number of entries in the lists is not fixed, the variables n_nistat, n_adcstat, and n_mcmstat stores these numbers. Together with the offset information in cfdat_error the beginning and end of each list can be determined. If all MCMs in a chamber work fine, the offsets are set to zero and the three lists are not present.

The calibration values of the MCM temperature sensors are stored in section cfdat_temp_calib. Each element of the array cfdat_mcm_temp_calib stores the calibration values for one MCM. Up to now this array is not filled or used since the calibration values are not determined. For details concerning the MCM temperature sensors see chapter 6.

The section cfdat_basecfg contains the name and the version of the current configuration and an offset to a list of SCSN commands. Each SCSN command is represented by one cfdat_command. The variable offset_cmd in cfdat_basecfg contains the offset to the beginning of this list and n_cmd contains the number of commands.

The list of SCSN commands is the core of each configuration. The MCMs have no non-volatile memory. Therefore all settings and even the assembler program for the MCM CPUs have to be loaded each time the MCMs are powered up. All this information is stored as SCSN command sequences in the database and later send to the MCMs by the FEEServer. Furthermore, special commands like switching on or off the power regulators or selecting a test are stored as SCSN commands, too. These SCSN commands, which are not send to the MCMs but processed in the FEEServer directly, are called extended SCSN commands.

The last section, cfdat_rocinfo, contains some general information about the ROC and the hardware on it. It contains the ROC type, the serial number of the ROC and two structs with information about the ROBs and the ORIs on the ROC.

Most of the sections have one or more variables to store checksums. The checksums can be used to ensure that no bits have flipped during the transfer from the Command Coder to the FEEServer. Up to now the checksums are neither created nor checked. The functionalities are not implemented yet.

One task of the InterComLayer is to compose and send the configuration. But the ICL has a second task. Each FEEServer publishes 10 DIM data channels: 1 message channel, 1 acknowledge channel, 1 channel containing the FEEServer FSM status, 1 channel for the DCS board temperature, and 6 channels for voltage sensor readings (see section 4.2). In total, the TRD has 540 FEEServers running. It turned out that PVSS can handle a maximum of 1000 changes in the DIM data channels per second. Otherwise it may crash.

The ICL connects to all DIM channels of the FEEServers and re-publishes the channels. Before re-publishing, it merges all message channels to one channel and suppresses the voltage sensor channels of the DCS boards which have no sensing wires to the power bus bars (see section 4.2). The acknowledge channels are handled by the ICL and not re-published. The ICL takes care that in the re-published DIM channels the maximum update rate PVSS can handle is not exceeded. Therefore the PVSS program connects only to the republished channels even if the original channels of the FEEServers are still available.

Additional monitoring software can either connect to the republished DIM channels or to the channels provided by the FEEServers. Figure 4.18 shows two examples. A database which logs all messages from the FEEServers connects to the merged message channel of the ICL whereas a program monitoring the acknowledge channels has to connect directly to the FEEServers.

5 The FEEServer Control Engine

The TRD has about 65,000 MCMs in total which need to be configured after each power up of the detector. In particular the production tolerance of the PASAs and ADCs in the MCMs and the tolerances between ROCs require an individual configuration for each MCM. Furthermore the power regulators on the chambers have to be switched on and off, a permanent temperature monitoring is required and the status of the chambers have to be monitored and reported to the upper layers of the detector control system. All these tasks are carried out by the FEEServer control engine.

This chapter gives an overview about the different parts of the control engine and how they interact with each other. Since the control engine is comprised of about 20,000 lines of code in total it is not possible to mention every function. To get a detailed up-to-date documentation of all classes, functions and variables the program doxygen[1] should be used. Doxygen parses the source code files, extracts comments in the source code written by the programmers and creates an interactive documentation in HTML format. The source code of the control engine can be obtained from the subversion repository of the Physics Institue of the University of Heidelberg[2].

Not all parts of the control engine were developed as part of this thesis. Different people contributed functions or classes: Venelin Angelov, Thomas Dietel, Ken Oyama and Kai Schweda.

5.1 General Design

The FEEServer of the TRD consists of two parts: the FEEServer core and the control engine.

The FEEServer core is a C/C++ program which runs on the DCS board. The only task of the FEEServer core is to act as a DIM server. It provides a DIM command channel to receive commands and DIM data channels to publish information. The first version of the FEEServer core was developed by Sebastian Bablok and Benjamin Schockert (both ZTT FH-Worms) and was implemented in C. Due to stability problems the parts of the FEEServer core needed by the TRD were rewritten in C++ and implemented as a new program.

Like the DCS board, the FEEServer core is not detector specific. All detector specific code is outsourced to the control engine. Each time the FEEServer core gets commands or configurations via DIM it calls the suitable function from the control engine.

The control engine of the TRD is split into the two libraries libTRD and trdCE. Both libraries are written in C++. The libTRD provides basic functionality like classes to com-

[1]Doxygen and its documentation can be obtained from www.doxygen.org. The program is open source and available for Linux and Windows.

[2]The control engine consists of two parts. The trdCE and the FEEServer core can be obtained from https://alice.physi.uni-heidelberg.de/svn/trd/feeserver/trunk/control-engine and the libTRD from https://alice.physi.uni-heidelberg.de/svn/trd/libTRD/trunk

municate with the MCMs, power control routines, and a logging system. The trdCE contains all high level functions like a finite state machine, handling of incoming commands from the FEEServer core, and classes for hardware tests. The main reason for splitting the control engine in two libraries is flexibility. On the DCS Board not only the FEEServer is installed. Many other tools are installed, too. They are mainly used for debugging and testing. Nearly all of these tools need hardware access but no complex control logic. By separating the basic functionality from the control logic, the tools can use the comfortable hardware interface provided by libTRD without having to use the quite complex control logic in the trdCE.

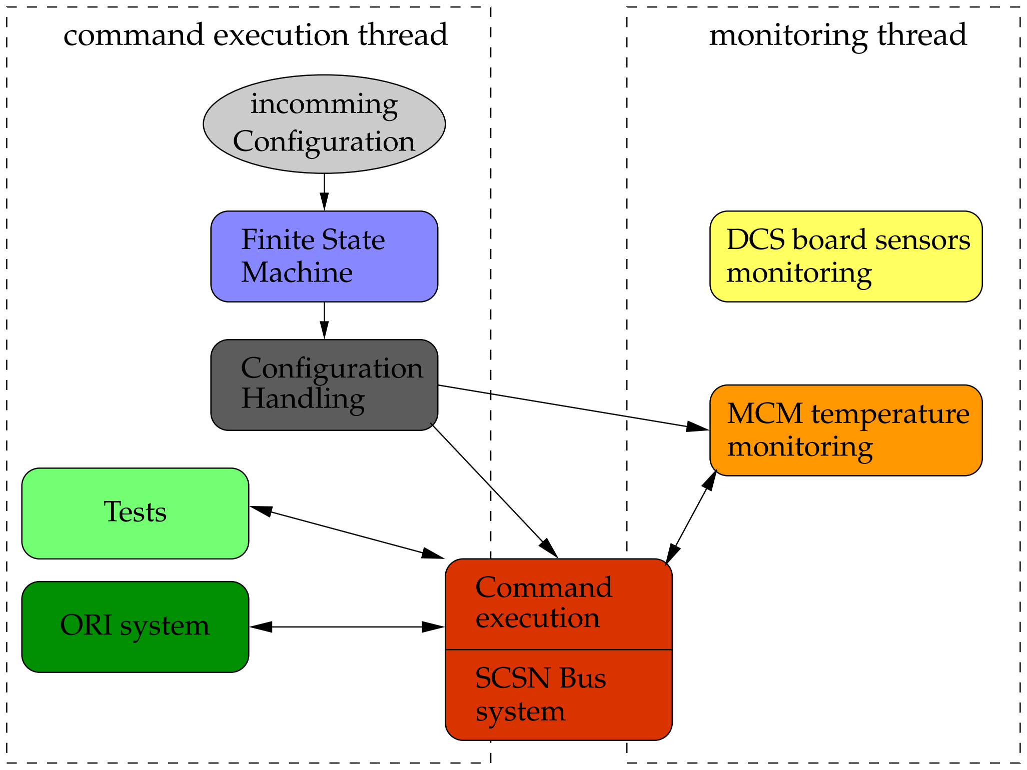


Figure 5.1: General control flow in the control engine

Figure 5.1 shows the general command flow in the control engine. The individual blocks in the figure represent different subsystems in the control engine. Some of them represent only one or two classes whereas other blocks represent complete class hierarchies. The tables C.2 and C.1 list all classes in the trdCE and libTRD with a short description.

The finite state machine (FSM) is the highest level in terms of control flow. It controls all other components of the control engine except the sensor monitoring. The sensor monitoring runs independently of the other parts in its own thread. Both the FSM and the sensor monitoring are implemented in the class CEStateMachine (trdCE). The class CEStateMachine is the central class of the whole control engine.

The classes in the section for configuration handling process the data received from the InterComLayer (see section 4.4). The configuration information is extracted from the received data and distributed to the corresponding classes. Furthermore, the list of

SCSN commands is extracted and forwarded to the command execution system. Important classes in this section are ROCControl (trdCE), ROCInfo (libTRD) and patch_maker (libTRD).

The test block contains classes for seven hardware tests. On the one hand the tests are started by extended SCSN commands. Therefore the tests are initiated from classes belonging to the command execution block. On the other hand the tests use the classes in the command execution block to send commands to the MCMs. All tests are part of the trdCE.

The ORI system contains four classes to communicate and to configure the ORIs. Most of the functionality is used only by the ORI test (see section 5.4.5). All classes of the ORI system belong to trdCE.

MCM temperature monitoring contains three classes to read out the internal MCM temperature sensors. The relevant class is TempControl (trdCE), the other two classes are just utility classes.

The command execution block handles the execution of SCSN commands. Extended SCSN commands are handled by the class ROCExecutor (trdCE) and not forwarded to the MCMs. The normal SCSN commands are forwarded to the SCSN system (libTRD) which sends the commands to the MCMs.

One system is not shown in figure 5.1. The logging system is used by all classes and therefore it has no fix position in the control flow. Most parts of the logging system are part of the libTRD.

A list of all classes with short descriptions is given in the appendix: Table C.1 lists the classes of libTRD and table C.2 lists the classes of trdCE.

5.2 Control and Interface Classes

5.2.1 Initialization of the Control Engine

When the FEEServer starts, it first creates three DIM channels for the control engine: a MESSAGE channel to output log messages, an ACK channel to send acknowledge messages and a CMD channel to receive commands from the ICL.

Afterwards it creates an instance of the class CEStateMachine and the constructor of CEStateMachine is called. The constructor of class CEStateMachine has three tasks.

First, it registers eight DIM data channels: T_ENV to publish the reading of the DCS board temperature sensor, STATE to publish the current state of the control engine FSM and V_A3V3, V_D3V3, V_A1V8, V_D1V8, V_AGND, V_GND to publish voltage readings. All eight channel names have TRD_FEE_xx_y_z_ as common prefix were xx is the supermodule number, y the layer and z the stack. To measure the voltages each DCS board has a connector which is connected to the ADC on the DCS board. From this connector, sensing wires are routed to the end of the power bus bars at the supermodule walls. Usually the sensing wires of the DCS board in stack 0 are connected with the power bus bars at the A-side walls of the supermodule and the sensing wires of the DCS board in stack 1 are connected with the power bus bars at the B side. The DCS boards in the stacks 2 to 4 have no sensing wires and therefore cannot measure the voltages. However, five of the 18 supermodules are powered from the other side. In these supermodules the DCS boards in stack 3 and 4 are connected to the power bus bars.

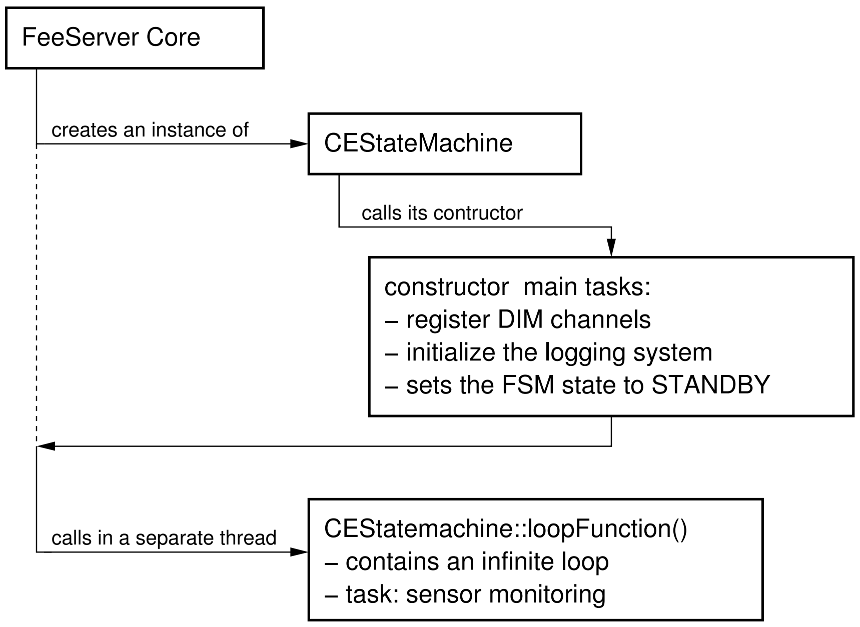

FeeServer Core
creates an instance of
CEStateMachine
calls its contructor
constructor main tasks:
– register DIM channels
– initialize the logging system
– sets the FSM state to STANDBY
calls in a separate thread
CEStatemachine::loopFunction()
– contains an infinite loop
– task: sensor monitoring

Figure 5.2: Control flow during the initialisation of the control engine. The control flow is symbolized by the solid lines. The dashed line is for illustration purpose only.

Second, the constructor initializes the logging system (for details see section 5.5). Third, it initializes the FSM and sets the state of the FSM to STDBY.

When the control engine is initialized, the FEEServer core creates a separate thread and calls the function **int** CEStateMachine::loopFunction(**void**). This function contains an infinite loop. The function reads the DCS board temperature sensor and the voltage sensors by calling **float** voltage_sensor::read_voltage(**int**) and **float** temp_sensor::read_temp(**int**) (see section 5.2.4) and updates the corresponding DIM data channels in regular intervals. If the MCMs are switched on later and the MCM temperature monitoring is activated this function triggers the regular readout of the MCM temperature sensors, too. After the initialization is finished the system waits for incoming commands from the ICL.

5.2.2 The Finite State Machine

Figure 5.3 shows the state diagram of the TRD frontend readout electronics. The finite state machine (FSM) of the control engine implements all these states execpt the states OFF and READY.

Description of the finite state machine states

In state OFF the supermodules have no power. The DCS boards are off and the FEEServer with the control engine is not running. Therefore OFF cannot be reached by the control engine itself.

When the DCS board is powered up, the FEEServer starts. The control engine is initialized and the FSM has the state STDBY. In this state only the DCS boards have power. All other parts of the chamber electronics including the MCMs and ORIs are still switched off. Since the DCS board has power, the sensors on the DCS board are already active and

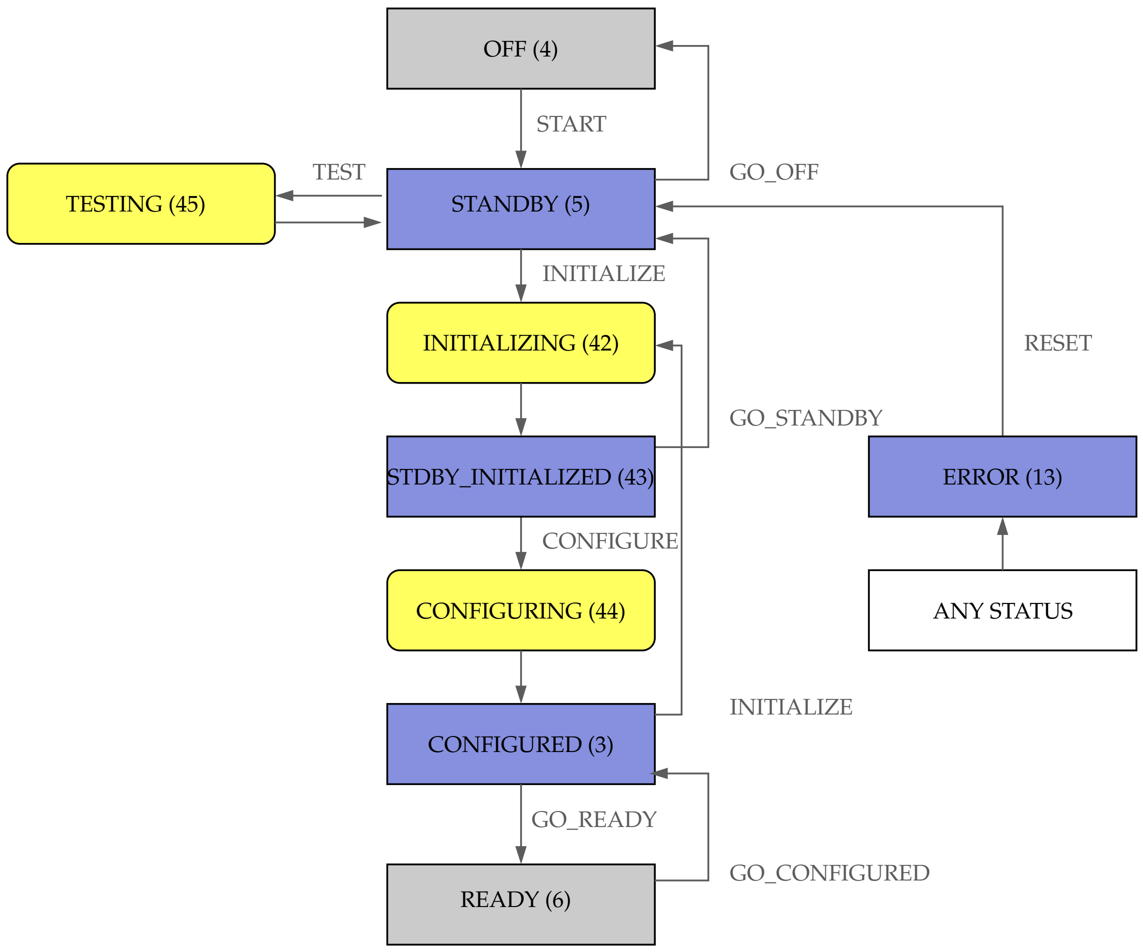


Figure 5.3: State diagram of the frontend reaout electronics. Transitional states are marked with yellow background. The numbers in brackets are the official numbers of these states in the ECS of ALICE.

their values are monitored by the control engine and published via DIM. In this state the command INITIALIZE or the command TEST can be sent via the InterCom layer.

Sending the command TEST runs one out of different tests to check the electronics. Which test is run is selected by additional data send together with the TEST command. An overview of the different tests is given in section 5.4. During a test, the FSM is in state TESTING. After the test is finished the FSM switches back to STDBY or in case of a failure to state ERROR.

If the command INITIALIZE is sent, the FSM switches to state INITIALIZING. During INITIALIZING the front end electronics are powered up. The control engine switches the power regulators of the readout boards on and all parts of the FERO get power. The MCMs are reset to default values. After the initialization is finished, the FSM switches to state STDBY_INITIALIZED.

When the FSM is in state STDBY_INITIALIZED the MCMs can be configured. The command CONFIGURE starts the process. Together with this command a configuration with about 2100 to 2500 commands is sent. Since the MCMs have no permanent internal memory all configurations and even the assembler program which runs on the MCM CPUs have to be loaded after each power up. Most commands in the configuration write parts of the assembler program to the MCMs. During the configuration process the FSM is in state CONFIGURING. When the MCMs are configured successfully, the FSM switches to state CONFIGURED.

As shown in the FSM diagram there are also transitions defined to go back to previous states. The command INITIALIZE can initiate a transition from CONFIGURED to STDBY_INITIALIZED, too. The command GO_STANDBY initiates the transition from STDBY_INITIALIZED to STDBY. During this transition the power regulators on the ROBs are switched off.

The state ERROR is a special state. It cannot be reached by any usual transition. Each time an error occurrs – an invalid transition request, a hardware or software error, or a failed test – the FSM switches to state ERROR. To get back to normal operation the transition command RESET has to be sent. Afterwards the FSM is in state STDBY regardless of the previous state. This behavior was chosen because in case of error any part of the system can be in an undefined state. Going back to STDBY forces a complete reconfiguration. Maybe a complete reconfiguration is not necessary in all cases but separating different error conditions is difficult and error-prone. The advantage of a reconfiguration is a well defined configuration afterwards, independent of the error reason.

Implementation of the finite state machine

The FSM is implemented in function triggerTransition(**char** *_cfdat) of class CEStateMachine as a nested switch-case structure. The outer switch-case separates the requested transition and the inner switch-case checks if the requested transition is defined in the current state. The current state of the FSM is stored in the class variable CEStateMachine::curr_state. Listing 5.1 shows an extract of the switch-case structure.

```
switch(action)
{
    case INIT:
        switch(curr_state) {
            case(STDBY):
            case(STDBY_INIT):   // to allow configuration reload
                without the necessity to go back to STDBY
                curr_state=INITIALIZING;
                // do something
                curr_state=STDBY_INIT;
                break;
            default:
                invalidTransition(action);
            } break;
    case TEST:
        switch(curr_state) {
            case STDBY:
                curr_state=TESTING;
                // do something
                curr_state=STDBY;
                break;
            default:
                invalidTransition(action);
        } break;
    case CFG:
        switch(curr_state) {
            case STDBY_INIT:
                curr_state=CONFIGURING;
                // do something
                curr_state=CONFIGURED;
                break;
            default:
                invalidTransition(action);
    } break;
}
```

Listing 5.1: Extract of the switch-case structure representing the FSM.

5.2.3 Handling of Incoming Configurations

The configurations are created by the CommandCoder of the ICL and sent via a DIM command channel to the FEEServer core. The data structure of the configurations was described in section 4.4. This section gives an overview how the configurations are processed inside the FEEServer.

Each time the FEEServer core receives a configuration, it calls the function CEStateMachine::triggerTransition(**char** *_cfdat) from library trdCE. This function contains the FSM logic described in the last section and represents the main control function of the control engine.

First the function opens the SCSN Bus interface via the mutex mechanism (see section 5.3.2) to have exclusive SCSN access during the whole configuration process and not to be disturbed by the second thread. Afterwards, function string triggerTransition(**char** *) evaluates the variable hd_maker (compare figure 4.19 on page 46) of the received configuration. This field contains the requested transition. If the transition is not allowed in the current state, the FSM switches to state ERROR. Otherwise the corresponding transitioned state is set. In case of the transition from STANDBY to STDBY_INITIALIZED or in case of

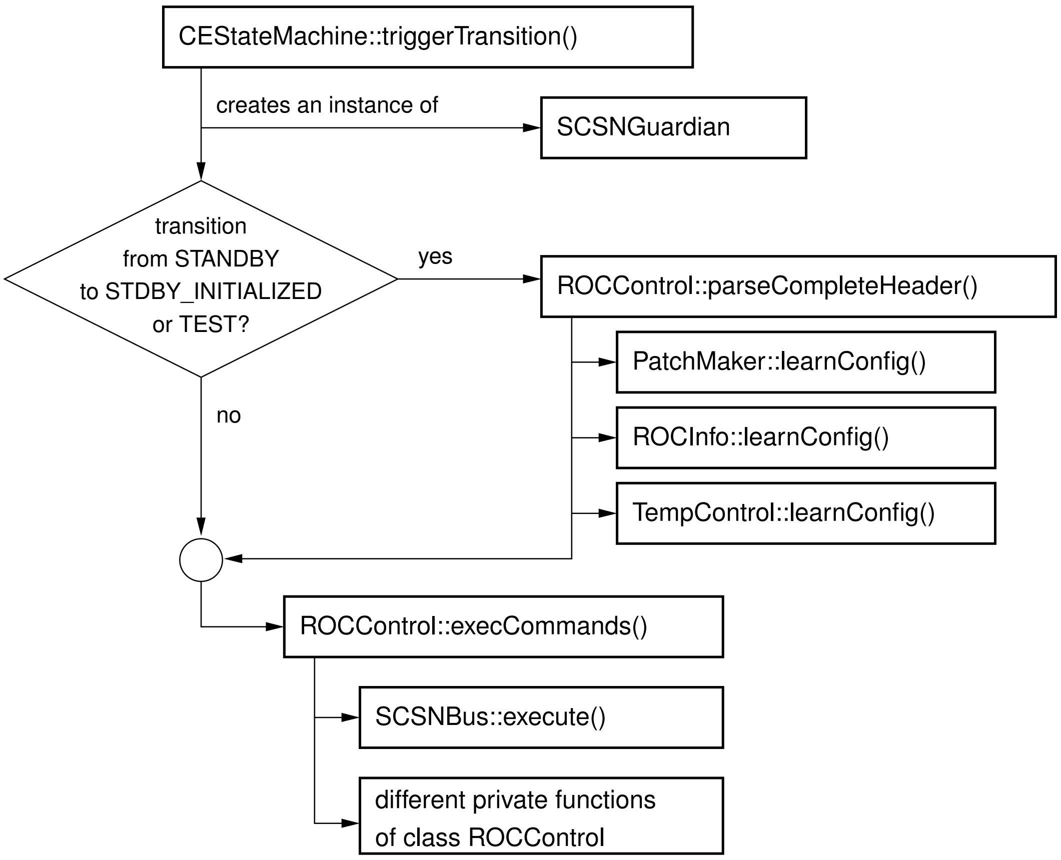

CEStateMachine::triggerTransition()
creates an instance of
SCSNGuardian
transition from STANDBY to STDBY_INITIALIZED or TEST?
yes
ROCControl::parseCompleteHeader()
PatchMaker::learnConfig()
ROCInfo::learnConfig()
TempControl::learnConfig()
no
ROCControl::execCommands()
SCSNBus::execute()
different private functions of class ROCControl

Figure 5.4: Overview of the control flow during the handling of incomming configurations. Only the main steps are shown. For some more details see text.

a test, string triggerTransition (**char** *) calls the function ROCControl::parseCompleteHeader (cfdat_header *header) first. This function configures the complete control engine. It extracts the offsets to three of the four sections in the configuration and calls the functions which process the information in these sections:

- The cfdat_error_header is passed to patch_maker. Based on these data the patchmaker creates the necessary SCSN commands which cause the MCMs to deactivate their damaged parts. Furthermore it creates a list of damaged MCMs which the class SCSNBus uses to bridge the linkpairs if necessary.
- The rocinfo_header is passed to class ROCInfo. ROCInfo stores the general chamber informations and provides get-functions to access this information. For example the function ROCInfo::getRocType() returns the roc type or ROCInfo::getNLinks() returns the number of linkpairs.
- The cfdat_temp_calib is passed to class TempControl. Since no calibration values are available yet cfdat_temp_calib is set to zero and the data are not processed further.

In case of any valid transition string CEStateMachine::triggerTransition(**char** *) calls **void** ROCControl::execCommands(cfdat_header *header). The function checks if the version of the configuration is compatible with the version of the control engine, writes the name of the configuration to the logging system and extracts the list of SCSN commands. The list of commands is passed to **void** ROCExecutor::execute(cfdat_command *cmds, **unsigned int** n_cmd).

The class ROCExecutor handles the execution of all SCSN commands. It is derived from class SCSNBus. The function **void** ROCControl::execute(cfdat_command *cmds, **unsigned int** n_cmd) loops over the list of SCSN commands. Blocks of normal SCSN commands[3] are passed to the inherited function **void** execute(cfdat_command *cmds, **unsigned int** n_cmd) of class SCSNBus. Extended SCSN commands are handled by private functions of ROCExecutor. Table B.1 in the appendix gives an overview over all defined extended SCSN commands. After the list of SCSN commands is processed successfully the control flow returns to function string CEStateMachine::triggerTransition(**char** *). The function updates the FSM state and unlocks the mutex. Afterwards the system is ready to process the next transition request.

In case of an extended SCSN command, like the command to run tests, it is possible that the function called to handle this command creates a list of SCSN commands which need to be executed. Usually one would create just a new instance of class ROCExecutor and use this new instance to execute the commands. But in case of the control engine this class and some other classes store configuration values and cannot be re-created when an instance of these classes is needed. Furthermore, classes developed for the libTRD use the class SCSNBus from libTRD to access the SCSN Bus but in the control engine the SCSN Bus access is provided by ROCExecutor. All classes from libTRD should automatically use ROCExecutor when running in the control engine.

To meet all these requirements a pair of classes was developed by Thomas Dietel: trd_factory for the libTRD and TRDCEFactory for the libce. Both classes are designed as singleton classes[4] [GHJV94].

[3]Commands with a number less then 16, see table B.1.

[4]A singleton class has some properties of a global variable. During runtime only one instance of the class can exist and this instance is accessible from every part of the program.

The only task of trd_factory and TRDCEFactory is the creation and administration of class instances. The class trd_factory stores pointers to the instances of SCSNBus, ROCInfo, rstate_reader, patch_maker, and ttcrx_stat . The class TRDCEFactory is derived from trd_factory. Therefore it administrates all pointers from trd_factory and additionally the pointers to ROCExecutor and TempControl. For each administrated class, trd_factory and TRDCEFactory provide a function which returns a pointer to this class. Internally trd_factory / TRDCEFactory check if an instance of this class already exists, creates an instance if it does not exists and returns the pointer of the instance. In case of the classes SCSNBus / ROCExecutor a special programming technique guarantees that inside the control engine every time a pointer to ROCExecutor is returned, even if a pointer to SCSNBus was requested.

5.2.4 Sensor Monitoring

A readout chamber has different sensors to monitor. The DCS board contains sensors to monitor the voltages in the power bus bars and a temperature sensor to measure the environment temperature. Additionally, each MCM has an internal sensor to measure the temperature inside the MCM.

The basic class for the readout of the DCS board sensors is the class adc_device (libTRD). All sensors on the DCS board are connected to an ADC. The class adc_device provides functions to read the ADC values.

The class voltage_sensor is derived from adc_device. It provides the function **float** voltage_sensor::read_voltage(**int**) which reads one of the six voltages digital/analog 3.3 V, digital/analog 1.8 V, analog ground, or digital ground (see section 4.2) using the ADC readout functions of class adc_device and returns the result. The parameter of function read_voltage(**int**) determines which voltage is read:

parameter value	sensor which is read out
1	analog ground
2	digital 3.3 V
3	digital 1.8 V
4	analog 1.8 V
5	digital ground
6	analog 3.3 V

The sensors are calibrated and the conversion from the ADC values to the corresponding voltages is done according to the formula

$$\text{voltage in V} = 1.386958 \cdot 10^{-4}\,\text{V} \cdot \text{ADC value} - 1.272417\,\text{V}. \qquad (5.1)$$

The class temp_sensor provides the function **float** temp_sensor::read_temp(**int**) which reads out an NTC temperature sensor attached to the DCS board. The parameter of the function has historical reasons and must be 1. The temperature sensor is connected to the same ADC as the voltage sensors. Therefore class temp_sensor is derived from class adc_device and uses its functionality to read out the ADC values. The conversion from the ADC values U_{ADC} to a temperature is done with the formula

$$\text{temperature in°C} = \frac{\ln(\frac{3.3-U_{ADC}}{U_{ADC}}) - 11.6753}{-0.0394864}°C - 273.15°C \qquad (5.2)$$

The calibration values and most of the functions in the classes adc_device, voltage_sensor and temp_sensor were taken from existing C programs.

The temperature sensor readout in the MCMs requires more elaborate readout functions. In principle, each MCM temperature sensor requires an individual calibration. Furthermore the sensors do not work reliably and therefore a runtime identification of non-working sensors is required. In this section only the main functions used for the temperature readout are described. A detailed description of the temperature sensors themself is given in chapter 6.

The class TempControl (trdce) encapsulates the readout of the temperature sensors. Each time the function **void** TempControl::measure(**void**) is called, the current temperature sensor value of all MCMs on the readout chamber is read out and printed to the logging system. An example of the output in case of printing absolute ADC values is given in listing 6.1 on page 79. The output format can be set via the extended SCSN command SCSN_CMD_MCMTEMP (see table B.2). To read out the sensors, first a write frame is send to all MCMs to trigger a measurement and afterwards the measured values are read out via SCSN read frames.

TempControl provides the function TempControl::learnConfig(**struct** cfdat_temp_calib*) to set calibration values for each MCM individually. However, it turned out that an individual calibration of the temperature sensors is not feasible (see chapter 6). Therefore TempControl::learnConfig(**struct** cfdat_temp_calib*) is mainly kept for historical reasons. As a consequence, the function **void** TempControl::measure(**void**) cannot print meaningful temperatures but prints out the raw ADC values of the temperature sensors or the difference between the first measured ADC value after switching on the MCMs and the current ADC value, depending on configuration.

The identification of non-working temperature sensors is performed during the transition from STDBY to STDBY_INIT. The function CEStateMachine::triggerTransition() calls TempControl::apply(**void**). This function reads out each MCM temperature sensor ten times and stores the results in the instances of MCMChip. For each MCM the class TempControl contains one instance of class MCMChip which stores the last ten sensor readings and status information.

Afterwards the function **bool** TempControl::calcStatus(**void**) is called. This function calls **bool** MCMChip::calcStatus(**void**) for each instance. **bool** MCMChip::calcStatus(**void**) uses the previously stored ten values to check if the temperature sensor does work or not and sets the corresponding status in class MCMChip. Then **bool** TempControl::calcStatus(**void**) counts the number of non-working sensors. If this number is below a threshold which can be set via the extended SCSN command SCSN_CMD_MCMTEMP (see table B.2), the MCMs are switched off for a short time. It turned out that this procedure increases the number of working sensors significantly (see chapter 6). After the MCMs were switched on again, the identification procedure is repeated.

In the output of **void** TempControl::measure(**void**) non-working sensors are represented by -1 in case of the absolute ADC values are printed since the possible range for real values is 0 to 1023, or by 999 in case of ADC differences are printed out.

5.3 The SCSN Bus Class System

The SCSN Bus class system is the most complex system in the control engine. It provides an easy to use interface to the SCSN Bus. It includes the class SCSNBus as its central class, the class SCSNBusBridge as an utility class for SCSNBus which handles the bridging, the class SCSNGuardian and the file scsn_ids.c which contains functions to convert MCM addresses, to calculate the ROB position of MCMs and some more. All classes of the SCSN system belong to libTRD.

5.3.1 Configuration

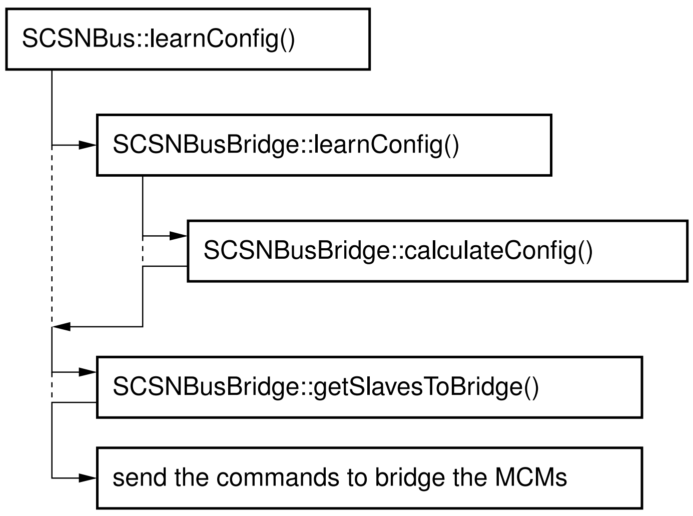

SCSNBus::learnConfig()
SCSNBusBridge::learnConfig()
SCSNBusBridge::calculateConfig()
SCSNBusBridge::getSlavesToBridge()
send the commands to bridge the MCMs

Figure 5.5: Configuration process of the SCSN Bus class system. Only the main steps are shown.

Before the SCSN Bus system can be used, it needs to be configured. The class SCSNBus provides the function learnConfig() to configure the system. Each time learnConfig() is called first the complete old configuration is deleted. This includes bridging information, information about the chamber type or the currently active linkpair. In the hardware, the bridge mode of all MCMs is reset, too.

SCSNBus::learnConfig() accepts two sets of parameters. The simple version **void** learnConfig(**int**) expects the chamber type (0 or 1) as the only parameter. This version is used by some tests and most of the stand-alone utilities based on libTRD. It just resets everything and sets the chamber type. Particularly, all MCMs remain unbridged. The information about the chamber type is needed to determine the number of linkpairs on the chamber.

The second version **void** learnConfig(**const** vector<SCSNDefects> &, **bool** [4][2], **int**) expects three parameters: a list of damaged MCMs / broken SCSN lines between MCMs, an array of damaged lines between the DCS board and the MCMs and the chamber type. Each entry in the list contains the extended ALICE ID of two MCMs. If both IDs are identical then the MCM with this ID is damaged. If the IDs are not equal the line from the first MCM to the second one is broken. Since there are two lines between each pair of MCMs the order of the IDs is important.

The DCS board has no ALICE ID. Therefore information about broken lines between the DCS board and the first/last MCM of each link has to be stored seperately. For this purpose an array was choosen. The first dimension of the array identifies the linkpair and the second one identifies the line (0: line between the DCS board and the first MCM in link 0; 1: line between the DCS board and the first MCM in link 1). For example, if the line between the DCS board and the first MCM in link 1 of linkpair 3 is broken, [3][1] has to be set to false and the other 8 entries of the array are set to true.

All information about damages in the SCSN Bus is stored in class SCSNBusBridge. The class SCSNBus contains four instances of this class, one for each linkpair. Both versions of SCSNBus::learnConfig() call the function learnConfig(**const** vector<SCSNDefects> &, **bool** [2], **int**, **int**) of class SCSNBusBridge for each of the instances. The first parameter is the list of damaged MCMs / broken SCSN lines between MCMs. The second parameter is a list containing the status of the lines between the DCS board and the first MCM in the linkpair. Each element in the list has either the value true or false. The value true means the corresponding line is OK, false means the line is broken. The first element in this list is the status of the line between the DCS board and the first MCM in link 0, the second element is the status of the line between the DCS board and the first MCM in link 1. The third parameter of SCSNBusBridge::learnConfig(**const** vector<SCSNDefects> &, **bool** [2], **int**, **int**) is the ROC type and the last one is the linkpair.

The class SCSNBusBridge contains the array mcm_info with 37 elements[5] of type MCMInfo, one element for each MCM in the linkpair. The array index is the slave ID of the MCMs in an unbridged linkpair which uses link 0 for the SCSN communication. In this array all information about the MCMs and how they can be reached is stored. Listing 5.2 shows the internal structure of MCMInfo.

The function SCSNBusBridge::learnConfig(**const** vector<SCSNDefects> &, **bool** [2], **int**, **int**) transfers the informations stored in the parameters to the MCMInfo array. It sets line_defect[0], line_defect[1] and damaged. If the line between the DCS board and the first MCM in a link is broken, the link is marked as completely dead because the DCS board cannot reach any MCM via this link.

```
typedef struct {
    bool line_defect[2];
    // line_defect[0] stores the status of the SCSN line between
        the
    // MCM and its next neighbour in link 0, line_defect[1] stores
    // the status of the line between the MCM and its next
        neighbour
    // in link 1. (default value for both entries: false)
    bool bridged;    // true if the MCM is bridged, false otherwise
        (default value: false)
    bool damaged;    // true if the MCM is damaged, false otherwise
        (default value: false)
    bool avail;      // true if the MCM can be reached via SCSN,
        false otherwise (default value: true)
    short dest;      // current slave address of the MCM
    int link;        // link of the SCSN bus to be used to
        communicate with the MCM (default value: 0)
```

[5]The first element (index 0) is never used and the last two elements are only used in linkpairs with 36 MCMs.

```
    int hopcount;    // Expected value of the hopcounter field in
        the SCSN frame after a
                     // frame has returnd from this MCM to the DCS
                        board
                     // (default value: number of MCMs in the
                        linkpair +1)
}MCMInfo;
```

Listing 5.2: Structure of MCMInfo

After the data are transferred to the array the function SCSNBusBridge::calculateConfig(**void**) is called. The function calculates the other values in MCMInfo.

First, SCSNBusBridge::calculateConfig(**void**) loops over the array mcm_info and checks for each element if both variables line_defect and damaged are set to false. If all variables are set to false, the linkpair has no damages. All MCMs remain unbridged and link 0 of the linkpair is used for sending. The variables dest, hopcount and link (see listing 5.2) are set to default values:

dest = default_slave_address,
hopcount = number_of_slaves_in_linkpair,
link = 0.

Second, it is checked if only lines in link 0 but no MCMs are broken. In this case link 1 can be used for the SCSN communication and no MCMs need to be bridged. Since now the frames shall travel exactly the other way around compared to using link 0, dest, hopcount and link get new values:

dest = number_of_slaves_in_linkpair−default_slave_address+1,
hopcount = default_slave_address−1,
link = 1.

If either at least one MCM in the linkpair or lines in both links are broken, the linkpair needs to be bridged. In a first step each error is bridged seperately and the status of the MCMs is set accordingly. Next the two outermost MCMs with status bridged are searched and the status of all MCMs in between is set to **not** avaialabe. Finally the new addressing scheme is calculated.

The procedure should be illustrated with an example. Assuming, the SCSN line in link 0 between the MCM with the slave ID 4 and 5 is broken and additionally MCM 12 is damaged (see figure 5.6). In this case the function learnConfig(**const** vector<SCSNDefects> &, **bool** [4], **int**, **int**) has set the variable line_defect[0] for MCM 4 to true and the variable damaged for MCM 12. In a first step calculateConfig(**void**) sets the status of the MCMs 4 and 5 and of the MCMs 11 and 13 to bridged. In the second step MCM 4 and MCM 13 are identified as the both outermost MCMs in bridge mode. The status of all MCMs in between is set **not** avaialabe because these MCMs are not reachable any longer. In the last step the new addressing scheme is calculated. Listing 5.3 shows the corresponding source code for the address recalculation.

Furthermore the expected hopcounter for returning broadcasts has to be calculated. Due to the bridging one has splitted the linkpair in two parts and therefore two values have to be calculated, one value for each part of the linkpair (listing 5.3).

```
for(int sl=1; sl <= no_of_slaves; sl++) {
  if(sl <=sl_left) {
    mcm_info[sl].dest = sl;
```

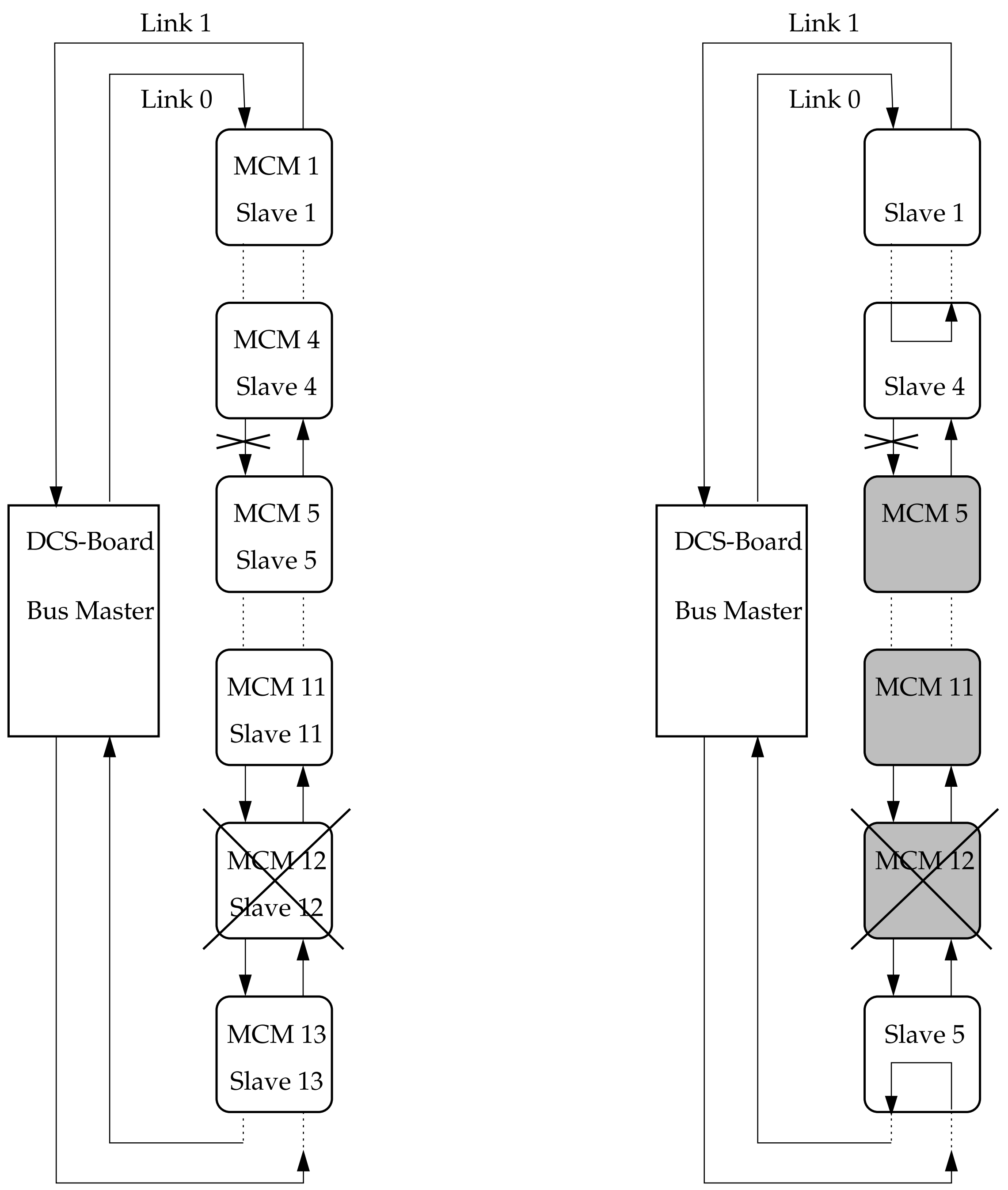


Figure 5.6: Example of bridging a linkpair with 17 MCMs: The left image shows a damaged line between MCM 4 and 5 and a damaged MCM 12. The right image shows the resulting bridged linkpair. After bridging, the linkpair works again but all MCMs between MCM 4 and 13 are not reachable any longer (drawn in gray). The dotted lines indicate not-drawn MCMs.

```
          mcm_info[sl].hopcount = sl_left*2 - sl -1;
          mcm_info[sl].link=0;
        }
        if(sl >= sl_right) {
          mcm_info[sl].dest = no_of_slaves - sl +1;
                    mcm_info[sl].hopcount = (no_of_slaves-sl_right)*2 - (
                        no_of_slaves-sl);
                    mcm_info[sl].link=1;
        }

        broadcast_info.data[0].hopcount_write=sl_left*2 -1;  // left
            part of the linkpair
        broadcast_info.data[1].hopcount_write=(no_of_slaves - sl_right )
            *2 +1; //right part of the linkpair
      }
```

Listing 5.3: Code to recalculate the slave IDs of the MCMs and the broadcast hopcounters in case of a bridged linkpair. sl is the slave ID of the MCM in an unbridged linkpair, no_of_slaves the total number of MCMs in the linkpair (34 or 36), sl_left the number of MCMs in the 'left' part and sl_right the number of MCMs in the 'right' part of the linkpair

When the calculation is finished the control flow returns to learnConfig() of SCSNBus. At this point it is known which MCMs have to be bridged but the MCMs themselves are not bridged yet. The function SCSNBus::learnConfig() calls * SCSNBusBridge::getSlavesToBridge (**void**) for each linkpair to get the list of MCMs to bridge. In each linkpair either no, one, or two MCMs have to be bridged. The function SCSNBus::learnConfig() creates the correct SCSN commands and sends them to the MCMs (see next section). If no MCMs have to be bridge this step is skipped.

Afterwards the configuration process is finished and the control flow returns to the function which has called SCSNBus::learnConfig().

5.3.2 Access Synchronisation

In the FEEServer two threads run in parallel. One thread handles commands and configurations received from the ICL and the other thread monitors the DCS board sensors and the MCM temperature sensors.

At this point a problem occurs. The sensor readout in the second thread runs independently of command handling thread but both threads need hardware access. Neither can the SCSN bus be accessed by both threads at the same time nor is it possible to read out the DCS board sensors and access the SCSN bus in parallel. To solve this problem the mutex mechanism[6] is used.

In the control engine the mutex administration and the interface to open and close the SCSN Bus are encapsulated in the class SCSNGuardian (libTRD). The constructor of SCSNGuardian locks the mutex and opens the SCSN bus interface and the destructor unlocks the mutex and closes the SCSN bus interface. Each time a thread needs SCSN bus

[6]The mutex mechanism is a programming technique for inter-thread synchronization. One thread locks the mutex as soon as it enters a function block which cannot run in parallel. If the other thread now tries to lock the mutex because it also enters a function block which cannot run in parallel, this thread is stopped as long as the mutex remains locked. When the mutex is now unlocked by the first thread the second thread can lock the mutex and continues execution.

access it creates an instance of SCSNGuardian. After the access is not needed anymore the thread deletes the instance.

Using the destructor to unlock the mutex has an advantage. When a function that locked the mutex is left, the mutex is released, independent of how the function is left – by a usual return or by throwing an exception.

5.3.3 Sending of SCSN Commands

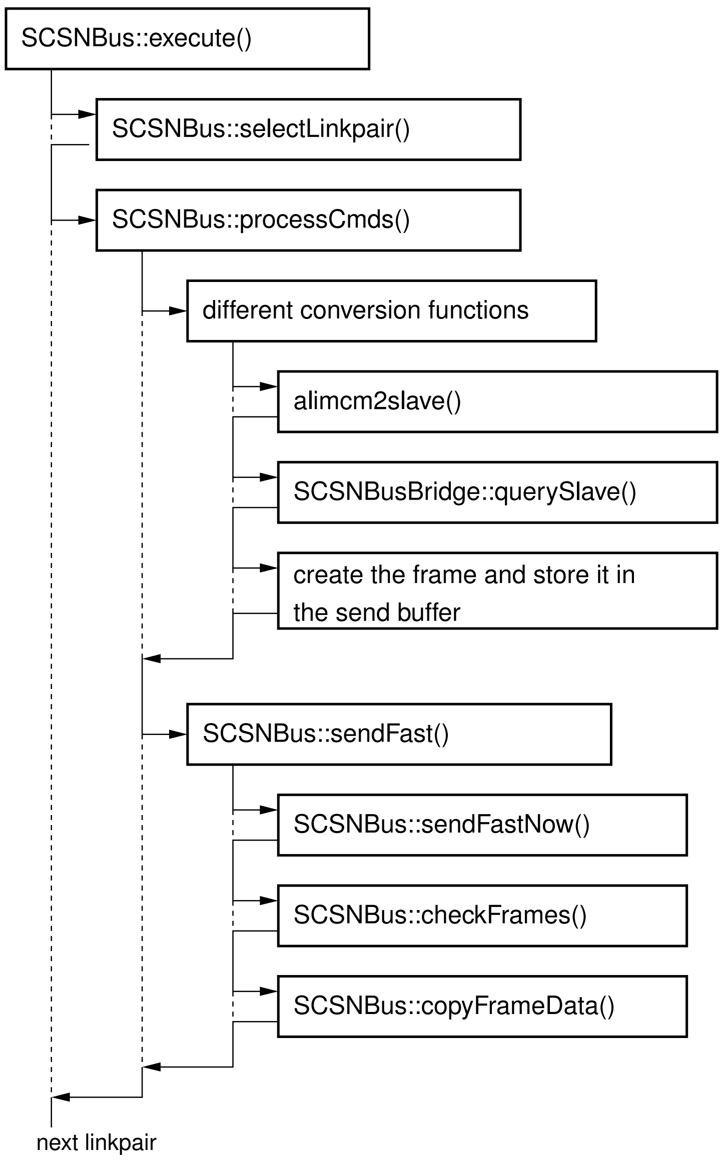


Figure 5.7: Control flow within the SCSN Bus class system for sending SCSN commands. The control flow (solid arrows) is simplified and only the main stepts are shown. The dashed lines are for better orientation only.

The class SCSNBus provides the public function **void** execute(**struct** cfdat_command*, **unsigned**, **int**) to send a list of SCSN commands. The first parameter is a pointer to the list of SCSN commands. Each list entry is a struct, containing the SCSN command, the ALICE ID of the destination MCM, a memory address in the MCM and a data field. The destination MCM is either a single MCM, a group of MCMs (board mergers, half chamber mergers, etc.) or all MCMs (broadcast). The second parameter is the number of elements in the list and the last one is the linkpair to use. The last parameter can be omitted. In this case a loop runs over all linkpairs. The function **void** execute(**struct** *cfdat_command, **unsigned** , **int**) controls the conversions. It calls processCmds(**struct** cfdat_command *, **unsigned**) for the creation and sending of the SCSN frames. The function loops through the list of commands and calls the matching conversion function for each SCSN command type. The conversion functions basically all do the same.

In the list of SCSN commands the target MCMs or MCM groups are identified by extended ALICE IDs. Before sending the commands, the class SCSNBus has to convert the IDs to the corresponding slave ID / link /linkpair information. An MCM always belongs to the same linkpair but the link and the slave ID depend on the bridging status of the linkpair. Furthermore, MCM groups addressed by an extended ALICE ID have to be expanded to a list of individual MCMs because the SCSN Bus supports only the addressing of individual MCMs or broadcasts by design.

Therefore the conversion functions first convert the extended ALICE ID to a list of slave IDs for an unbridged linkpair. If the extended ALICE ID addresses an MCM or a MCM group which is not the currently active linkpair, the list is empty and the next command in the command list is processed. In case of a single MCM which is in the linkpair or the broadcast address, the list contains only one slave ID, in all other cases the list contains more then one slave ID.

Afterwards the conversion function calls **const** RoutingInfo *SCSNBusBridge::querySlave (**int**) for each slave ID in the list to convert the just determinded default slave ID to the current slave ID of the MCM. If the linkpair where the MCM belongs to is not bridged and link 0 of the linkpair is not damaged, the default slave ID and the current slave ID of the MCM are identical (compare section 5.3.1). Based on the information in the command list and on the information from **const** RoutingInfo *querySlave(**int**) the frames are created and stored together with additional information in the send buffer.

The send buffer is an array of of type FramesToSend (see listing 5.4) and has a fixed size. Each element of the array stores all data of one frame. The frame itself is stored in the variable the_frame. The other variables contain informations necessary to send the frame correctly and to check the returning frame later.

```
typedef struct {
  int cmd_seq_no; // Command sequence number, particularly
      important for broadcast reads
  int frame_no;    // Frame number
  int fd_w;        // No. of File handler (link) where this frame
      should be written
  int fd_r;        // No. of File handler (link) where this frame
      should be read from when returning
  int hopcount;    // Expected hopcounter when frame has returned
  int expt_source;// Expected source bit when frame has returned
  unsigned extali_id;// ALICE ID of the target slave
  unsigned orig_id;  // ALICE ID from the incomming command list.
      For single MCMs this is equal to extali_id but for mcm groups
       both values differs
```

```
    FrameSend the_frame;
}FramesToSend;
```

Listing 5.4: Structure of struct FramesToSend

If the linkpair is bridged the conversion functions have to handle two special cases. First, it is possible that the target MCM is not reachable. In this case frames addressed to the MCM are not added to the send buffer and consequently not sent later. Second, broadcasts have to be handled differently. In an unbridged linkpair it is sufficient to send one frame with the broadcast address as address to reach all MCMs. In a bridged linkpair the linkpair is split in two parts. Therefore two broadcast frames have to be send, one frame for each half of the linkpair.

When all commands are processed or the send buffer is full, SCSNBus::processCmds(**struct** cfdat_command*, **unsigned**) calls the function sendFast(**void**). First sendFast(**void**) calls sendFastNow(**int**) for each of the two links of the current linkpair.

The function sendFastNow(**int**) does the actual sending of the frames. It loops through the send buffer of class SCSNBus, checks for each frame if the target MCM belongs to the current linkpair / link combination and writes the frame to the correct file handle (`/dev/scsn1` or `/dev/scsn2`). During the file handle determination the function takes into account that interface `/dev/scsn1` is connected to the physical link 0 in case of linkpair 0 and 1 and connected to the physical link 1 in case of linkpair 2 and 3. `/dev/scsn2` is connected in opposite direction.

The two file handles are the interfaces to the SCSN bus functions of the FPGA on the DCS board. The FPGA provides a hardware buffer where SCSN frames can be stored before they are sent. Up to 16 frames can be stored in this buffer. Since the function sendFastNow() fills the buffer much faster than the frames are sent, sendFastNow() writes 16 frames to the file handler and then waits for the returning frames. When the frames return, sendFastNow() reads them from the file handler and stores the frames in the buffer for received frames in class SCSNBus. Afterwards the next bunch of 16 frames is sent until all frames for the current linkpair / link combination are sent and received again.

After all frames have been sent and received again, the program flow returns to sendFast(**void**). This function calls checkFrames(**void**) next. checkFrames(**void**) checks the received frames for different kinds of errors: the error bit of the frame must not be set, the destination, memory address and command field of the sent and received frame have to have the same values, and the hopcounter value and the source bit have to meet expectations. The expected values for hopcounter and source bit are extracted from the variables hopcounter and expt_source of the corresponding entry in the send buffer. Any error during these checks results in throwing an exception and the FSM usually goes to state ERROR. Some of the tests (see next section) catch the exception and in such cases the FSM remains in state TESTING until the test is finished. If the check is successful, sendFast() calls copyFrameData() which copies the relevant parts of the received frames to a data buffer.

The public functions std :: vector<FramesResult> **const** &SCSNBus::getResults(**void**) and std :: vector<FramesResult> **const** * SCSNBus::getResultsPointer(**void**) of class SCSNBus provide access to this buffer. The only difference between these functions is the return type. In the control engine only some of the tests request these data.

The handling of the commands SCSN_CMD_READ, SCSN_CMD_BRIDGE, and SCSN_CMD_PAUSE differs more or less from this scheme.

The differences for SCSN_CMD_READ are small because only broadcasts require a different handling. One would expect that sending the SCSN_CMD_READ command to the broadcast address results in one answer frame from each MCM in the linkpair. However, the concept of the SCSN bus hardware is one frame in – one frame out (see section 4.3.5). Therefore the function SCSNBus::readData() which is called by execute(**struct** *, **unsigned**, **int**) to handle SCSN_CMD_READ creates a separate frame with the read command for each MCM in the linkpair. These frames are then processed as described above.

The command SCSN_CMD_BRIDGE sets MCMs to bridge mode. It is used by SCSNBus ::learnConfig() and by the bridge test. In normal configurations this command should not be used. Before the command is processed all remaining frames in the send buffer are sent because a SCSN_CMD_BRIDGE always modifies the address scheme of the linkpair. The function SCSNBus::learnConfig() creates and sends the bridge frames on its own. A bridge request from the bridge test is processed by the function **void** SCSNBus::Bridging(**unsigned** , **unsigned**, **int**).

The command SCSN_CMD_PAUSE has two completely different purposes, depending on the value in the data field. The dual use of this command has historical reasons. In both cases first the frames stored in the send buffer are sent. If the value in the data field is bigger than 0 the program halts for the time in micro seconds given in the data field. Such a wait command is used when the MCMs need some time to process the previous commands. If the value in the data field is smaller than zero the status of the state machine of the MCM named in the dest field is read. This functionality is needed by some tests.

5.3.4 Transmission Speed

In section 4.3.5 measurements of the throughput of the SCSN bus hardware were shown. The throughput achieved by class SCSNBus much lower because the class has to compose the SCSN frames, send them via the SCSN bus, receive the returning frames and check them. For a list of 5000 ping/write/read SCSN commands, it takes about 1.2 s between the moment the class SCSNBus gets the list and the moment the class signalize that the last frame has been received. This results in a transmission speed of 4200 frames per second or 361.2 kBits/s. Because an SCSN frame has only 32 bit of payload (the data field of the frame) the netto data transmission rate for the SCSN access in the control engine is about 134.4 kBits/s.

5.4 Hardware Test Functions

The control engine of the FEEServer provides various tests to check the ROC electronics. The tests can run only from state STDBY. The reasons for this were already discussed in section 5.2.2. During a test is running the FSM is in the state TESTING.

All tests are mutually independent so they can be run in any order. However it is impossible to run more then one test on a chamber concurrently. To launch a test the same overall procedure is used as for the transition from STDBY to STDBY_CONFIGURED: The electronics is switched on, configured and the list of SCSN commands is executed (see section 5.2.3).

The tests are started by the extended SCSN command SCSN_CMD_RUN_TEST. The data field determines which test is executed. The function ROCExecutor::executeROCCmds which processes all SCSN commands calls the static function **static** TestClass* TestFactory :: createTest(**int**). The function creates an instance of the requested test and returns the pointer to the instance. Afterwards ROCExecutor::executeROCCmds uses this pointer to call the function **int** doTest(**void**) of the created test. This function runs the test and returns the number of errors occurred during the test. Since all tests are derived from the abstract class TestClass which defines the public function **int** doTest(**void**), ROCExecutor ::executeROCCmds() can use the same function calls for all tests. Details about the implementation of the individual tests can be found in the doxygen documentation of the source code.

If the test has been run successful – i.e. the number of errors returned by the test is zero – the FSM switches back to state STDBY and the electronics are powered off. In the other case an exception is thrown and the FSM of the control engine switches to state ERROR. The electronics remains in the status it had when the error was detected. The reason for switching to ERROR and not going back to STDBY in case of a failed test is simple. External programs running test sequences automatically need a simple interface to check whether a test was successful or not. Parsing all of the log output is much more complicated then just monitoring a status channel. However, this approach requires to reset the ERROR state before running another test even if the next test would not be affected by the detected error.

Presently, seven tests are implemented.

5.4.1 Bridge Test

The bridge test is used to identify and locate dead MCMs or broken SCSN Bus lines between MCMs.

At the beginning of the test all bridging information in class SCSNBus and all bridged MCMs are reset by reconfiguring the class SCSNBus via the function learnConfig(). Afterwards the test sends the bridging command to the MCM with ALICE ID 0 via the first link of the linkpair. The MCM switches to bridging mode. Then a ping frame is sent via the same link. Since now the linkpair is bridged this frame should not return to the DCS board on the first link but on the second one. Hereupon the de-bridging command is sent to MCM 0. The same procedure is repeated for the second link. This procedure is repeated for all MCMs in all linkpairs.

On the basis of the information gathered during the test a list of unreachable MCMs is generated. This list can be stored in a database and will be used by the class patch_maker and SCSNBus to bridge the MCMs.

5.4.2 Laser ID Test

Each MCM has a serial number, called Laser ID. This name was selected because a laser is used to blow fuses in a controlled fashion on the MCM. The resulting circuit encodes the Laser ID. Unfortunately, this number is not totally unique, nevertheless it can be used to identify MCMs. An advantage of the Laser ID in contrast to other possibilities to identify MCMs is that this number can be read out via the DCS board.

This test sends a read command to all MCMs and the MCMs send back their respective Laser ID. Afterwards the Laser IDs are compared to the IDs stored in the class ROCInfo which got these data during the configuration process. Each difference is reported as an error.

5.4.3 Reset Test

The Reset Test checks whether the reset command works correctly for all MCMs. A bit pattern is written to the memory address 0x0D46 of each MCM. The test currently uses the pattern 0xACACACAC, but it is also possible to use other bit patterns. The pattern is then read back from the MCMs to check it was correctly written. After finishing this preparation a reset frame is sent via the SCSN Bus to all MCMs. Then the memory address is read out again. If the reset was successful the memory address will now contain the default value 0x0000FFFF. A deviating value indicates that something went wrong during the test and an error is reported.

5.4.4 Shutdown Test

This test checks the functionality of the voltage regulators of the ROBs. Each linkpair has voltage regulators that control the power for the MCMs in that linkpair. One task of the voltage regulators is to switch the power for the MCMs on and off.

During the test the voltage regulators of different linkpairs are switched on and off. First all regulators are switched off, then the regulators for linkpair three are switched on, then linkpair zero is switched on additionally, then linkpair zero is switched off and linkpair one is switched on and so on. After each change a ping frame is sent to all linkpairs. The ping frames sent to linkpairs which are powered on are expected to return. Ping frames sent to linkpairs powered off must not return. To avoid that damaged MCMs affects the results the first and last MCM of each linkpair is set to bridge mode before the ping frames are send. Any deviation from the expected result is reported as an error.

The main purpose of this test is to identify power regulators which are switched on permanently. In case of power regulators which are permanently off the corresponding MCMs get no power. This would be detected while configuring the supermodule or taking data immediately.

5.4.5 ORI Test

The task of the ORI Test is to read out different sensor values and the EEPROM configuration of the ORI. The connection to the ORI is established via a modified i2c protocol named j2c. The DCS board cannot communicate directly with the ORI but two MCMs on each board with a ORI can put in a special operation mode when they forward bidirectional data and commands from the SCSN Bus to the ORI.

Therefore at the beginning of the test one of the two MCMs is set to the j2c forward mode. Afterwards the DCS board reads all relevant values from the ORI, compares with the default values and closes the connection.

The test prints the EEPROM configuration, the measured optical power output of the ORI, the laser diode temperature, different voltage values and some more. In particular the optical power of the diode is interesting because this information can be used to localize problems when the GTU gets only a weak optical input signal.

5.4.6 Memory Tests

Three memory tests are available to check the internal memory of the CPUs in the MCMs. First, initial patterns of 0xABCD are written into memory areas starting at 0xF000 of each CPU. Then the patterns are read back by a second CPU: CPU0 is read out by CPU1, CPU1 is read out by CPU2, CPU2 is read out by CPU3 and CPU3 is read out by CPU0. Each time the read out pattern differs from the expected pattern, the address of the memory at fault is subsequently stored in other memory areas of the CPUs. At the end of the tests these memory areas are read out and a detailed report is created, containing the addresses of all damaged memory areas.

The tree memory tests for DMM, DDD, and IMM memory (for detail see [A^{+}05]) mainly differ in the memory addresses that are checked. Additionally, in case of the DMM and DDD test, the board merger and half chamber merger are allowed to have memory errors. Therefore they are not checked during DMM and DDD test.

5.4.7 Network Interface Test

The Network Interface test checks the connections of the Readout Tree (see sec. 4.3.6). First a bit pattern is written to a memory area of each MCM via SCSN Bus and it is checked that this was done successfully. Afterwards a pretrigger frame is sent to each MCM triggering the data transfer to the MCMs in the next higher level of the Readout tree. Then the memory area where the received data are stored is read out via the SCSN Bus and compared with the expected pattern. This procedure is repeated with different bit patterns to increase the sensitivity of this test.

Only the spare and the parity line of each connection is not checked with the procedure described above. Therefore in a second pass two other lines are configured as spare line and parity line. After the reconfiguration the complete test is repeated.

Every difference between the received and expected pattern is reported as an error. Additionally the exact position of the broken line is determined and reported.

5.5 Error Handling and the Logging System

5.5.1 Error Handling in the Control Engine

The error handling in the control engine is based on the exception mechanism of C++. Each time something unexpected happens which the current function cannot handle it executes the code in listing 5.5. Usually one would create an instance of a class which stores information about the error and then call the C++ function **throw**() with the class as parameter. However, the ARM cross compiler used to compile the control engine for the DCS board does not support this technique. It can handle only the build in types of variables. Therefore a different implementation was chosen.

The class ErrorInfo is implemented as a singleton class. It is derived from the C++ class ostringstream to get streaming functionality. Each time an error occurs, first the error type is set (line one in listing 5.5). At the moment 19 different error types are defined but the types most frequently used are ebug and efatal. The type ebug indicates an error caused by a bug in the control engine. This type of error should not occur in normal operation. The type efatal is used for all errors which prevent the further execution of the control engine and requires action from the operator. After the error type is set, the error message is written to the class. As already mentioned it is not possible to use classes in the catch-throw mechanism. Therefore an integer value of 42 is thrown.

Usually the exceptions are caught by CEStateMachine::triggerTransition(). The function prints the error type and the message stored in ErrorInfo to the logging system and sets the status of the FSM to state ERROR.

```
ErrorInfo::getInstance()->setErrortype(type);
*(ErrorInfo::getInstance()) << "An error message" << endl;
throw(42);
```

Listing 5.5: Code block executed in case an error is detected.

5.5.2 The Logging System

In the control engine log messages are created in most of the functions. Many log messages are relevant only during development or debugging, other log messages are of general interest because they indicate errors in the hardware or problems with a configuration. To handle these different types of log messages and to keep the source code clear a central logging facility was developed.

The logging system of the libTRD / trdce provides one global logging stream named tlog. This stream can be connected to four different types of output channels: log file, standard out, the syslog facility of Linux and DIM channels. The number of output channels connected to tlog is not limited.

Each output channel and each message has a log level. The available log levels are ecritical, eerror, ewarn, einfo, and edebug. The log level ecritical is the highest log level and edebug the lowest log level. Every message streamed to tlog is forwarded to all output channels but only printed out by the output channel if the log level of the message is equal or higher to the log level of the channel. Otherwise the message is ignored by the channel. The log level of an output channel is set during its initialization and usually not changed afterwards. The log level of a message is set by the stream operator loglev(loglevel). If a log message does not contain the stream operator the log message gets the same log level as the previous one.

The logging system consists of four classes plus one class for each output channel type. The central class is LoggingStream. The logging stream tlog is the only instance of LoggingStream in the control engine. The class LoggingStream is derived from std::ostream and therefore it has the same functionality as the built in C++ streams. Additionally, LoggingStream provides functions to add and remove an output stream, to set the log level of a stream and to set the log level of a message. However, the class itself contains no functionality but calls the corresponding functions of class LogBufferManager. Therefore LoggingStream contains a pointer to an instance of LogBufferManager.

The class LogBufferManager is derived from std::stringbuf. Its task is to administrate the different output channels. It implements all functions provided by the class LoggingStream. A special function in LogBufferManager is the function **int** sync(**void**). Each time a message terminated with <<endl; is send to tlog this function is automatically called

at the end of the message. LogBufferManager contains a list of pointers to all output channels. The function sync(**void**) loops through the list and calls the function **void** sync(string message, string source, loglevel) of each output channel. These sync functions decide depending on the loglevel if the message should be printed out and do the actual output.

All classes for the different output channel types are derived from the abstract class LogBuffer. At the moment there are four different output classes available. The class CoutLogBuffer provides an interface to STDOUT, the class FileLogBuffer writes the log messages to a file, the class SyslogLogBuffer prints it out to the Linux logging system and the class ODimLogBuffer publishes the log messages via the FEEServer to the DIM system.

Even though the system is quite complex the usage is simple. During the initialization of the control engine one DIM logging channel and one syslog channel are created. The syntax for the DIM channel is for example tlog.addStream("dimlog", **new** ODimLogBuffer(einfo));. The parameter dimlog is the name of the channel and used to identify the channel. The second parameter ODimLogBuffer(edebug) specifies the type of the new channel and the log level of this channel. The syntax for the syslog channel is the same. After the initialization of the two channels, the command tlog << loglev(einfo) << ''The log message '' << endl; results in a log message of log level einfo in the DIM system and in the syslog of Linux.

6 Temperature Monitoring of the Front End Readout Electronics

6.1 Temperature Monitoring Inside the Supermodule

Monitoring the temperature inside the supermodule is important for different reasons. First of all measuring the temperature is the only technically feasible solution to check the proper cooling of all MCMs. The silicon hoses or the small aluminum tubes can be blocked and in consequence the corresponding MCM column remains uncooled. Furthermore a silicon hose can be laced up due to the under-pressure when not properly routed. Especially in layer five this has happened several times. As long as these problems are detected during supermodule integration they can be fixed. After a supermodule is finished, it is very difficult if not even impossible to fix cooling problems in the supermodule.

During normal TRD operation the most probable failure in the cooling system is a breakdown of the cooling plant. Such a failure should be detected by the monitoring software of the cooling plant and results in a shutdown of the TRD. However, monitoring the temperature in the supermodules provides a good extension and a fallback system for cooling monitoring. Lastly, the temperature inside the supermodule is also interesting for physical reasons. The drift velocity in the Ar-CO_2 gas mixture inside the chambers depends on the temperature. Even though the thermal coupling between the gas mixture in the chambers and the ambient air is weak and the drift velocity in the gas mixture is monitored separately, measuring the temperature may give useful information about the environment conditions.

Each ROC has three different types of temperature sensors. The first one is a dedicated temperature sensor connected to the DCS board. It is placed about 2 cm away from the DCS board. Therefore it measures the air temperature in the surroundings of the DCS board. The sensor is a 10 MΩ NTC resistor. The voltage drop in the resistor is digitized by an ADC on the DCS board. The FEEServer running on the DCS board reads out the sensor in regular intervals, converts the sensor reading to a temperature measured in degree Celsius and publishes the value via DIM.

The DCS board has a second temperature sensor directly mounted on the board. However, this second sensor is not calibrated and measures the local DCS board temperature only. Therefore this sensor is not used.

Finally, each MCM has an embedded temperature sensor which measures the temperature inside the MCM. Therefore, these sensors give a very localized temperature, in contrast to the temperature sensor connected to the DCS board which measures the global air temperature. Since the MCM has six sensors in total (the temperature sensor and five voltage sensors to measure different voltages in the MCM) but only one 10 bit, 80 kSamples/s ADC [A^+05] to digitalize sensor readings, an analog multiplexer is used to connect one of the sensors with the ADC. Which sensor is connected with the ADC can be selected via an SCSN command. The sensor readout is done via the SCSN bus, too.

The MCM temperature sensors are ideal to identify individual uncooled MCMs or blocked cooling lines. But using these temperature sensors contains some challenges.

- Reliability
  Each time an MCM is powered up there is a certain chance that the embedded temperature sensor will not work. Reading out such a sensor results in useless values. Therefore a runtime identification of non-working temperature sensors is required. Since working temperature sensors remain working as long as the MCM has power the runtime identification has to be performed only after each power up of the MCMs. The procedure to identify non-working temperature sensors is described in section 6.2.1.

- Cooling overview
  Each layer in the supermodule contains 656 MCMs. After a layer is installed and connected to the cooling system a quick overview about cooled and not cooled MCMs is required. But the conversion from the sensor readings to temperatures cannot be done very precisely due to calibration problems. Therefore a method was developed to generate a cooling overview of a layer without relying on calibrated temperature sensors (section 6.2.2).

- Calibration
  The MCM temperature sensors are not calibrated. In principle each MCM requires an individual calibration to convert the ADC values to temperatures. As we will see, such a calibration is not feasible. Therefore only a global calibration was performed to get at least a rough estimate of the temperatures. Section 6.2.3 gives more details.

6.2 The MCM Temperature Sensors

6.2.1 Runtime Identification of Working Temperature Sensors

The MCM temperature sensors do not work very reliably. Each time the MCMs are powered up some temperature sensors do not work.

Detailed tests show that working temperature sensors remain working as long as the MCM has power. But if the power is switched off and on again later a previously working temperature sensor may not work afterwards and a sensor which did not work may deliver correct values now. Using a special power-off / power-on sequence it is even possible to increase the average number of working temperature sensors on a readout chamber (ROC): First the MCMs on the ROC are powered for more than two seconds. Then the power is switched off. If the power returns after about 0.5 seconds the number of working temperature sensors increased significantly. Without using this power sequence a ROC has 30 to 40 percent working temperature sensors and afterwards 60 to 70 percent of the temperature sensors do work. This power sequence is implemented in class TempControl and is executed each time the MCMs are powered on.

The MCMs have a design error which causes timing problems in parts of the MCM. The exact mechanism is not known but probably the design error cause the observed behavior.

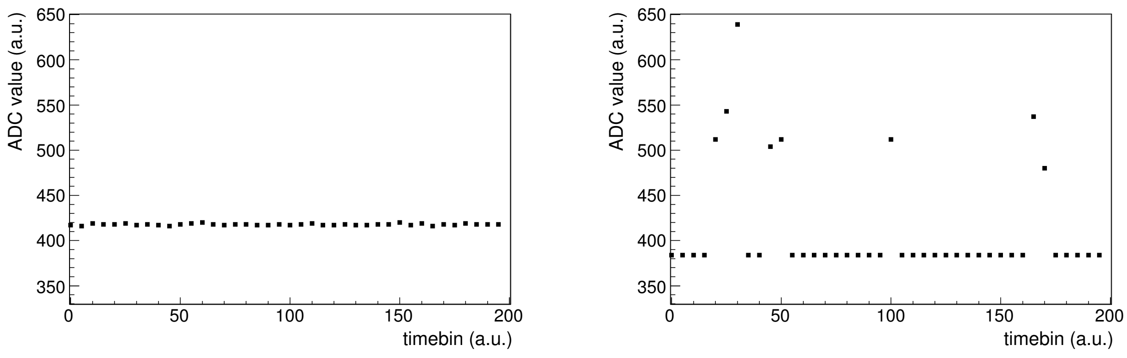


Figure 6.1: Sensor readings of a working (left) and a non-working (right) MCM temperature sensor

Figure 6.1 shows the readings from a working and a non-working temperature sensor. The data were measured in less than two seconds and therefore the temperature change is negligible. Consequently, the working sensor measured almost the same value each time. The small variations are caused by noise. In contrast the non-working sensor shows huge variations in the data. An analysis of the readings from working and non-working sensors revealed some characteristics of non-working temperature sensors.

- The ADC values 361, 384 and 639 are measured frequently. Additionally, in consecutive measurements the reading often jumps between these values.
- ADC readings are below 5 or above 1020. The ADC is an 10 bit ADC and therefor the possible values are between 0 and 1023. Extreme readings indicate a non-working sensor.
- Consecutive readings have exactly the same value. Usually the readings of two consecutive measurements varies by ±1 or ±2.
- Large deviation of single readings from the average value.

Based on these results criteria were defined to distinguish between working and non-working temperature sensors. If one of the following criteria is fulfilled the temperature sensor is considered as non-working:

- Equal or more than 20% of the readings are 384, and equal or more than 20% of the readings are 639.
- The sum of number of samples with values 384, 639, 361, below 5, or above 1020 is more then 60% of the number of total samples.
- More than 80% of the readings have exactly the same value.
- At least one reading deviates by more than 10 from the average value of this MCM.

The thresholds were adjusted iteratively to get no non-working temperature sensors marked as working and as few as possible working temperature sensors marked as non-working.

The above criteria were implemented in an algorithm in the class TempControl for runtime identification of non-working temperature sensors. In the final test a complete layer with 656 MCMs was read out. For each MCM a plot like the one in figure 6.1 was created to identify non-working temperature sensors manually. In parallel the algorithm was used to identify working and non-working sensors. The algorithm found all non-working sensors and only in one case it marked a sensor as non-working which seemed to be working.

6.2.2 Cooling Overview for a Supermodule

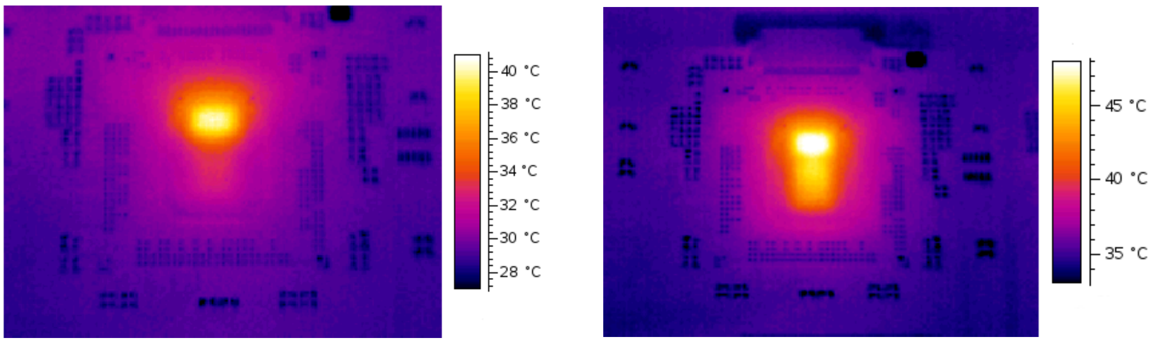


Figure 6.2: IR image of an switched on MCM in idle state (left) and during data taking (right). The PASA part of the MCM is active in both cases but the TRAP chip of the MCM produces much more heat during data taking compared to idle state. (Compared to figure 4.8 the images are rotated 90 ° clockwise.)

Figure 6.2 shows two IR pictures of an MCM. On the left image the MCM is switched on but in idle state and on the right image the MCM is configured and takes data. The PASA chip of the MCM is active in both situations but most parts of the TRAP chip are switched off if the MCM is not configured. Since the temperature sensor is located in the TRAP chip one would expect significant temperature differences between the two states and active or not active cooling should influence the level of temperature change between the two states.

In this thesis a measurement process was developed which uses these temperature differences to identify uncooled MCMs even without having calibration data for the temperature sensors. The only drawback of this method is that the procedure takes about two hours in which the supermodule cannot be used for other purposes like data taking. However, identifying uncooled MCMs is mainly important during the supermodule integration. When the supermodule is assembled, the DCS board temperature sensors are sufficient to monitor the global temperature in the supermodule. Using the MCM temperature sensors during data taking may even increase the noise level in the data: Each time an MCM temperature is measured, first one SCSN command has to be send to trigger the measurement and then a separate SCSN read frame for each MCM is required to read out the result. These activities on the SCSN bus and in the MCMs can increase the noise level.

Measurement Process

To measure the MCM temperature as a function of the two different MCM states the small program mcm_temps was developed. It gets most of its functionality from classes in libTRD and trdCE, including SCSN communication, the function to identify working and non-working MCM temperature sensors and the function to read out the temperature sensors.

The program needs three configuration files: for the transitions from STDBY to STDBY_INIT, from STDBY_INIT to CONFIGURED, and from CONFIGURED back to STDBY_INIT.[1] The configuration files contain the same data that are transferred from the wingDB to the FEEServer during the corresponding transitions of the finite state machine. By using configuration files the program becomes completely independent from the software infrastructure and the configurations in the wingDB. Therefore it is guaranteed that always exactly the same configurations are used and the results of the temperature measurements remain comparable.

Before a measurement is started the supermodule should be switched off for 30 to 60 minutes. This time is required to make sure all MCMs are cooled down and have approximately the same temperature. Afterwards the DCS boards have to be switched on and the program with the three configuration files has to be copied to each DCS board. The program requires four parameters. The first parameter is the chamber type (0 for a six ROB chamber or 1 for a eight ROB chamber) and the other three parameters are the filenames of the configuration files. When the program is started it processes the data in the first configuration file (transition from STDBY to STDBY_INIT). When the configuration is finished it measures the MCM temperatures at regular intervals. The interval can be configured in the first configuration file and is set to 5 seconds by default. After some time (usually 20 or 40 minutes) the second configuration and the same time later the third configuration is sent. These intervals are set in the source code of mcm_temps.

It requires some time and much typing to copy the necessary files to all DCS boards of a layer or even a supermodule, starting the program, and collecting the results afterwards. Therefore three shell scripts were written which reduce the necessary effort. The script dcs_script.sh runs on the DCS board and controls mcm_temps. Usually this script does not need to be called by a user. The script do_test.sh copies all necessary files to the DCS board and performs the temperature measurement for one chamber. It expects the chamber name, the chamber type, and an output directory as parameters. The chamber name is the network name of the target DCS board. The name of the output file where the recorded data are written to is the chamber name plus the extension dat. The data format is shown in listing 6.1. Each number represents one MCM temperature sensor. Non-working sensors are indicated by -1.

```
Debug (\CE): DCS–Temperature: #### 23.3049
Debug (\CE): ROB: 0: ### 360 416 467 431 461  -1  -1 400 372 360 453 379  -1  -1  -1 298  -1
Debug (\CE): ROB: 1: ### 404 421 420 388 358  -1 436 427 427 439  -1 362 462  -1  -1  -1  -1
Debug (\CE): ROB: 2: ### 431  -1 458 376  -1 362 430 409 420 384 408 442  -1 441 373 375  -1
Debug (\CE): ROB: 3: ### 401 457  -1 281 339 396 411 406  -1 372 349  -1  -1  -1 385 460  -1
Debug (\CE): ROB: 4: ### 408  -1 402 347  -1 432 389  -1 386 455  -1 370 380 398 394 424  -1  -1
Debug (\CE): ROB: 5: ### 443  -1 377 358  -1 358  -1 449  -1  -1 367  -1  -1  -1 309  -1  -1  -1
Debug (\CE): ROB: 6: ###  -1  -1 448 405  -1 360  -1 363 404  -1 406 395  -1  -1  -1  -1  -1
Debug (\CE): ROB: 7: ###  -1 402 413 355  -1 351 374 386  -1  -1 442 422 405  -1  -1 412  -1
```

Listing 6.1: Data format used by the temperature measurement program. Non-working temperature sensors are represented by -1 since the possible range of the ADC values is 0 to 1023.

The script temp_layer.sh calls do_test.sh for each chamber of a layer. It requires only the list of layers as parameters. To measure a complete supermodule one has to call just temp_layer.sh 0 1 2 3 4 5.

The data format used by the temperature measurement program is human readable.

[1]Even if the program does not contain a finite state machine, in this section the same names as in section 5.2.2 are used since the configurations are the same.

One needs no conversion programs to check if the data are recorded correctly. The further data analysis is done with the ROOT framework developed by CERN [B$^+$]. The ROOT framework uses a special data format for input and output which stores data very efficiently. Therefore the conversion program temp_converter was developed which converts the output of the temperature measurement program to the ROOT format. The program expects as start parameters either layer, stack, the time difference between to measurements, name of the input file, and name of the file where the converted values should be written to or just the time difference between to measurements and the name of two directories. In the second case all files in the first directory with the extension dat are converted to ROOT files of the same name and written to the second directory. If the program is called without any parameters it prints detailed information about its usage.

Table 6.1 gives an overview of all files required for the data recording and the conversion of the recorded data to ROOT files.

file name	description
mcm_temps	Program to measure the MCM temperatures
main.cc, mcmconfig.cc, mcmconfig.hh	Source code files for mcm_temps
temp_converter	Program to convert the recorded data to ROOT files
temp_converter.cc	Source code of the data conversion program temp_converter
Makefile	Makefile to create mcm_temps and temp_converter
config_201.txt, config_701.txt, config_351.txt	Configuration files containing the transitions for STDBY to STDBY_INIT, STDBY_INIT to CONFIGURED and CONFIGURED to STDBY_INIT.
dcs_script.sh	Control script for mcm_temps, runs on the DCS board
do_test.sh	Control script to measure the MCM temperatures of one chamber
temp_layer.sh	Control script to measure the MCM temperatures of one layer or a supermodule

Table 6.1: Overview of the required files to perform an MCM temperature measurement cycle.

Data Analysis

Figure 6.3 shows the temperature sensor readings of two MCMs as a function of time, recorded with the measurement process described in the previous section. At $t = 0$ min the first configuration for the transition from STDBY to STDBY_INIT is processed. The PASA part of the MCMs and some parts of the TRAP are powered on. This results in a first temperature increase. After 40 minutes the commands for the transition from STDBY_INIT to CONFIGURED are send. All parts of the TRAP are active now. This results in an increased power consumption of the TRAP chip and causes a second temperature increase. At $t = 80$ min the third configuration is sent. The TRAP is reset and

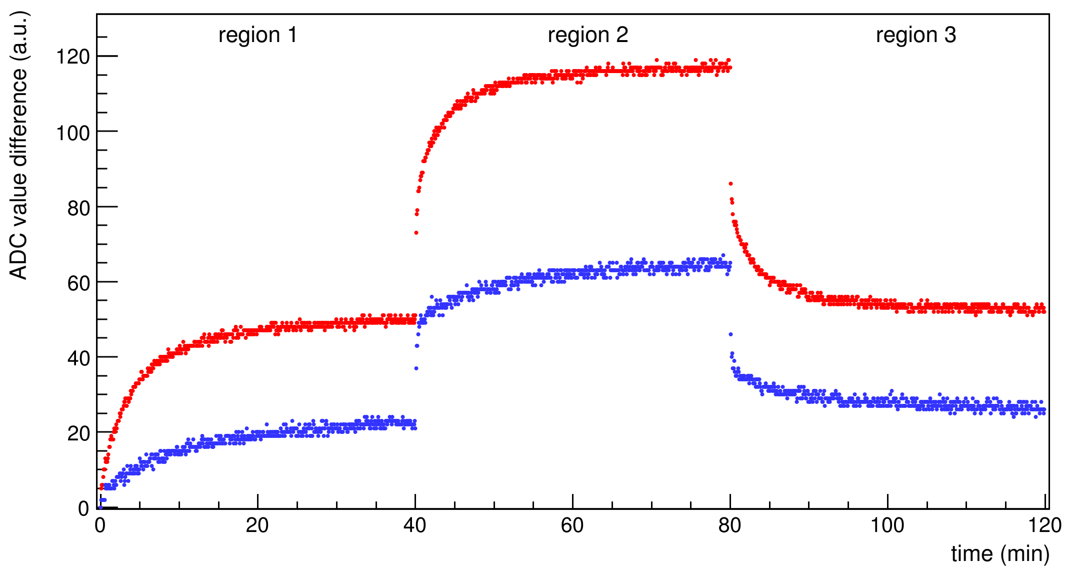


Figure 6.3: MCM temperature sensor readings as a function of time. The three regions corresponds to the configurations STDBY_CONFIGURED, CONFIGURED and STDBY_CONFIGURED as described in the last section. The red points are the data from an uncooled and the blue points are the data from a cooled MCM.

stops data processing. As a result the temperature decreases and reaches nearly the previous level after some time. The small fluctuations in the data are caused by measurement uncertainties and can be estimated to be ± 1 ADC value. Since the temperature sensors are not calibrated the absolute ADC value of two MCMs with the same temperature can differ by more then 100. To compare this, for all analyses in this chapter the first value of an MCM is set to zero and only the difference between the first value and consecutive values is shown. Section 6.2.3 gives more information about problems with the MCM temperature calibration.

It is obvious that the three regions in the plot have to be processed separately. Therefore, first an algorithm is required which can separate the three regions automatically.

The runtime identification of working temperature sensors uses a difference of more than ten in two consecutive values as one criterium to mark an MCM temperature sensor as non-working. Therefore jumps of more than ten should not occur in the data except at the points were the configurations are applied. The algorithm checks the data of each MCM separately and stores the points in time where jumps of more than ten between two consecutive readings occur. Afterwards the points of time of all located jumps are collected and the moments where most of the MCMs have a jump in their readings are selected as the common boundaries between the sections.

Figure 6.3 shows some differences between cooled and uncooled MCMs which can be used for the identification of uncooled MCMs. Uncooled MCMs have a larger and steeper temperature increase in region one and two, resulting in a higher end temperature and a slower temperature decrease in region three. For further analysis and to develop an algorithm which can identify uncooled MCMs it was tried to parameterize the curves.

In figure 6.4 for each section the absolute value of the difference between the current

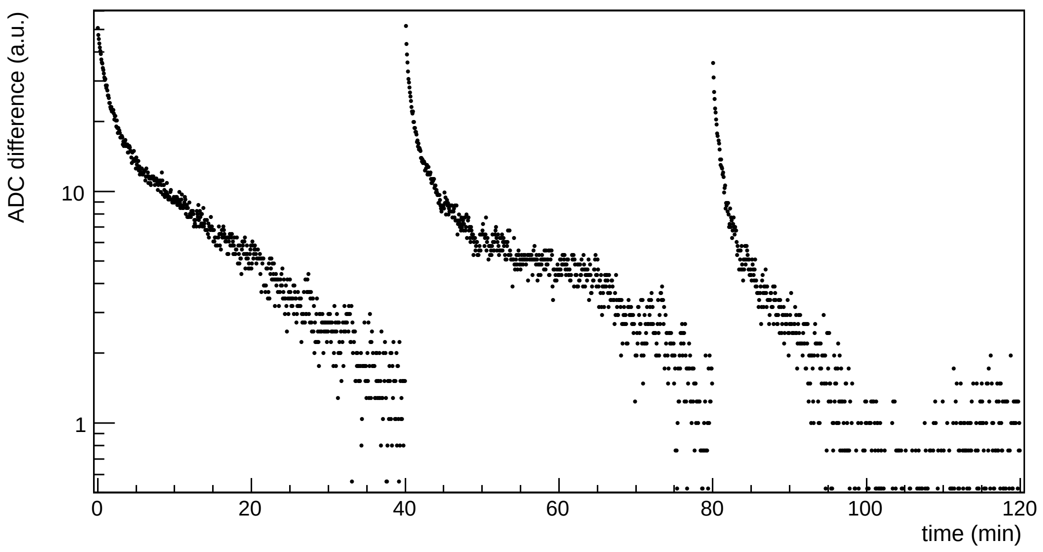


Figure 6.4: Difference between current and last MCM temperature value, calculated separately for each section.

value and the last value of the section is shown[2] If the curves can be parameterized by a simple exponential function the points should be on a straight line in each region. This is not the case. But the points can be approximately described by a sum of two exponential functions. Such a fit is shown in figure 6.5 for a cooled and an uncooled MCM. The error bars are ± 1 ADC value. The formula used for the fit is

$$f(x) = a_1 \cdot e^{b_1 \cdot (x - x_0)} + a_2 \cdot e^{b_2 \cdot (x - x_0)} + c. \tag{6.1}$$

Fitting a function with many parameters has some disadvantages. For each MCM the fitting algorithm requires adequately chosen start values for the fit parameters. Otherwise the fit fails. Second, it is difficult to extract information about the cooling status of the MCM when the number of fit parameters is that high. Third the fitting process takes some time. A complete layer needed more than 30 minutes for the fitting.

Therefore the data shown in the plots 6.4 and 6.5 were analyzed again. One can see that expecially the temperature change at the beginning of each section requires to use a sum of two exponential functions. If one excludes the first minute of each section from the fit and accepts a slightly worse parametrisation of the data a single exponential function can be used for fitting each region.

$$f(x) = a \cdot e^{b \cdot (x - x_0)} + c \tag{6.2}$$

The parameter x_0 shifts the curve parallel to the x axis. This shift parameter is used to achieve $x - x_0 = 0$ at the beginning of each section. Therefore x_0 is fixed to the difference between the begin of the measurement and the begin of each section. The other three parameters are fit parameters. Parameter a is the magnitude and parameter b determines

[2]The data were recorded separately with an uncooled MCM. Every 5 seconds the temperature was read out eight times and the eight values were averaged. Otherwise the noise would have been to high for this plot.

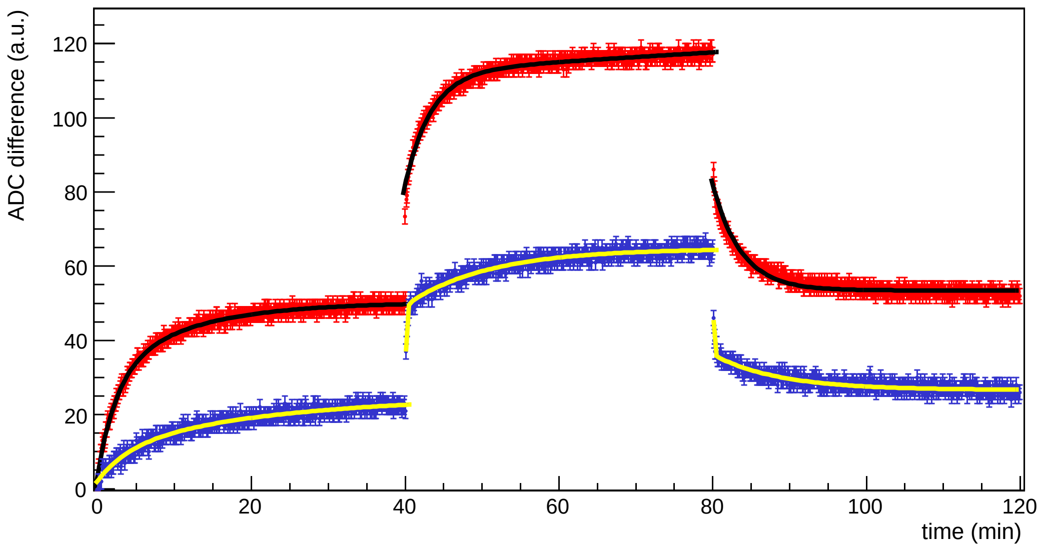


Figure 6.5: MCM temperature sensor data with fitted functions of type $f(x) = a_1 \cdot e^{b_1 \cdot (x - x_0)} + a_2 \cdot e^{b_2 \cdot (x - x_0)} + d$ (solid lines). Each section was fitted separately. The data are the same already shown in figure 6.3.

the slope of the curve. Parameter c determines the final value of the function for $x \rightarrow \infty$ since $b < 0$ for all fits in this chapter.

Figure 6.6 shows the fitted curves. But for a cooling overview three fit parameters for each MCM and each section are still too many. A formula is required which takes the nine fit parameters as inputs and gives just one number ('score') which indicate how well cooled an MCM is.

To deduce such a formula, the temperature curves of each MCM in two complete layers were fitted. One layer had active cooling and the other one was not cooled. The distribution of the obtained fit parameters are shown in figure 6.7. In the left column the magnitude is shown as a function of the slope and in the right column the magnitude is shown as a function of the limit. The linear dependence between the magnitude (parameter a) and the limit (parameter d) is expected.

As described above, x_0 is chosen to fulfill $x - x_0 = 0$ for the first x of each section. For this value equation 6.2 simplifies to $f(x) = a + c$. Therefore a linear dependence between a and c just means that $f(x)$ has the same value for all MCMs at the beginning of the region. In particular the fit parameters for region one have to show this linear dependency since the first MCM temperature sensor reading of all MCMs was set to zero (see begin of this section).

The plots in figure 6.7 show that uncooled MCMs always have a larger absolute value of $a \cdot b$. After some tests it was decided to use the following formula to calculate the 'score':

$$\text{score} = 50 \cdot a_1 \cdot b_1 + 10 \cdot a_2 \cdot b_2 - 20 \cdot a_3 \cdot b_3. \tag{6.3}$$

a_1 and b_1 are the fit parameters obtained by fitting the function 6.2 in region 1, a_2 and b_2 are the fit parameters from region 2 and a_3 and b_3 are the fit parameters from region

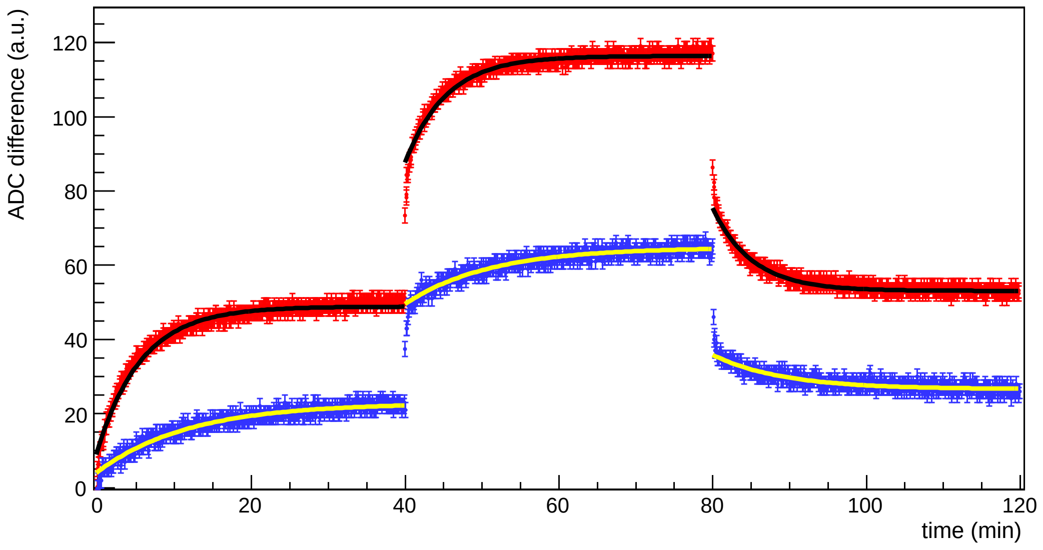


Figure 6.6: MCM temperature sensor data with fitted simple exponential functions (solid lines). The data are the same already shown in figure 6.3.

3. The factors 50, 10 and 20 in formula 6.3 are weighting coefficients. It turned out that the differences in the fit parameters between cooled and uncooled MCMs are bigger in region 1 than in the other two regions. Therefore this part is given a larger weight.

Two examples of the final cooling overview showing the score values of all MCMs in a layer are given in figure 6.8 and 6.9. The two plots show the score value of the same layer obtained in different measurements. The measurement for plot 6.8 was done about half an hour after an electronics stress test. Therefore the layer was not completely cooled down at the beginning of the measurement. The other measurement was done after the layer had been switched off for more than one day. As a result the average score value in plot 6.8 is higher than in plot 6.9. However, the bad cooled MCMs are clearly identifiable in both plots due to their higher score value compared to the other MCMs.

Furthermore one can see the effect of the MCM temperature readout unreliability. All MCMs with no score value in the plots had a non-working temperature sensor during the measurement (except the MCMs in columns 60 – 75 in plot 6.8). Some MCMs have a score value only in the first and other only in the second plot.

The analysis program

All the analysis functions described in the last section have been compiled to one ROOT library named libTempAlys.so. The library contains the class LayerTemp which provides the commands necessary to analyze the recorded temperature data.

When ROOT is running, first the library has to be loaded with the command gSystem −>Load("libTempAlys.so"). Afterwards one has to create a instance of the contained class via the command LayerTemp *layer= **new** LayerTemp. Now ROOT provides the following commands for the data analysis:

- layer−>loadRootFile("filename"): Add a data file. All files have to belong to the same layer.

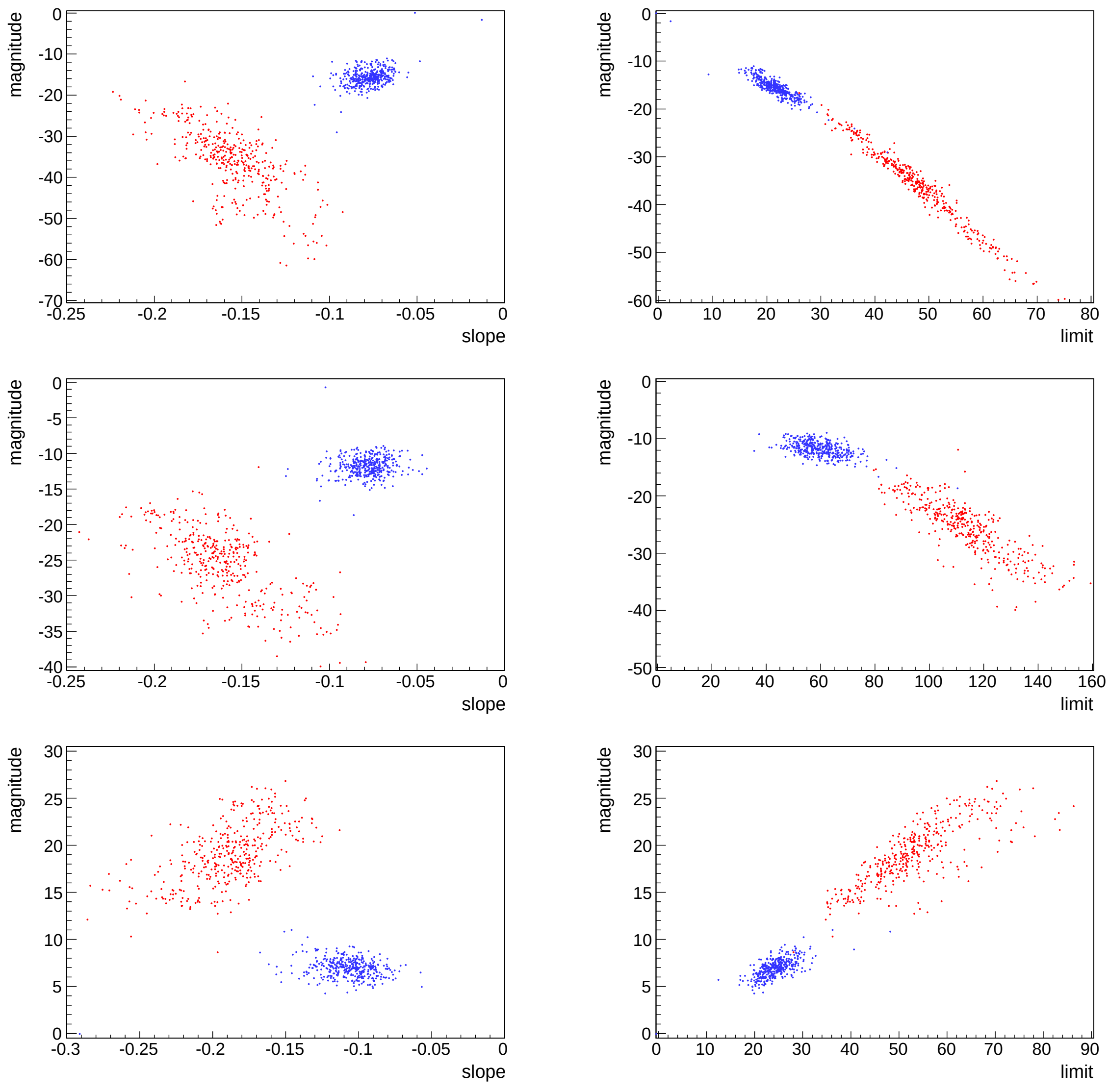


Figure 6.7: Scatter plots of the magnitude as a function of slope (left column) and magnitude as a function of limit (right column) from equation 6.2. Blue points are the fit parameters from cooled MCMs and red points are the fit parameters from uncooled MCMs. The two uppermost plots show the fit parameters from fitting the data in region 1, the two plots in the middle show the fit parameters for region 2 and the remaining two plots show the fit parameters from region 3. The positions of the three regions were shown in figure 6.3.

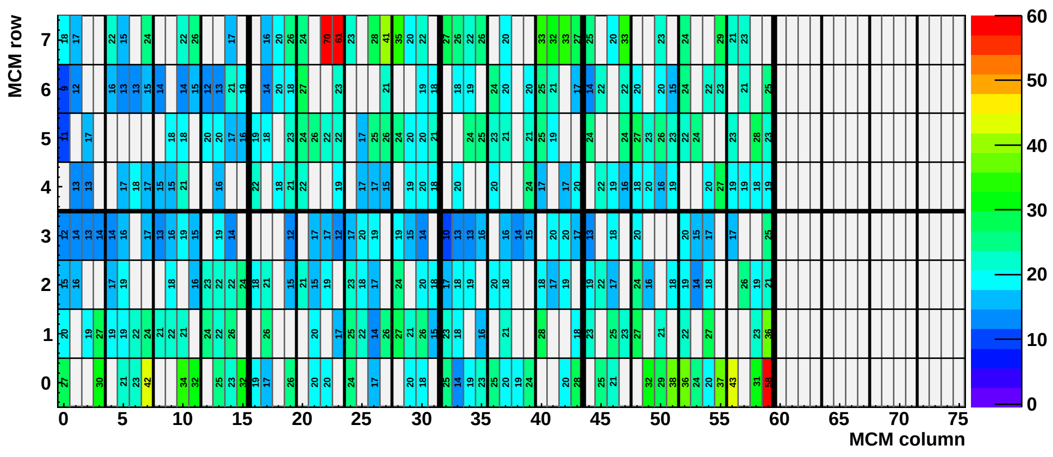


Figure 6.8: Cooling overview of a complete layer with active cooling. One can see that some MCMs have a worse cooling than other MCMs. This is caused by a bad thermal coupling between the MCMs and the cooling plates glued on top. As long as only a few MCMs are affected no further action is required since the MCMs get sufficiently cooled by the thermal coupling between the MCM and the readout board and by air cooling. The readout chamber in the last stack (MCM column 60 – 75) was switched off. The MCMs without a score value in the other stacks had a non-working temperature sensor during this measurement and were automatically excluded. Figure 6.9 shows a second measurement of the same layer.

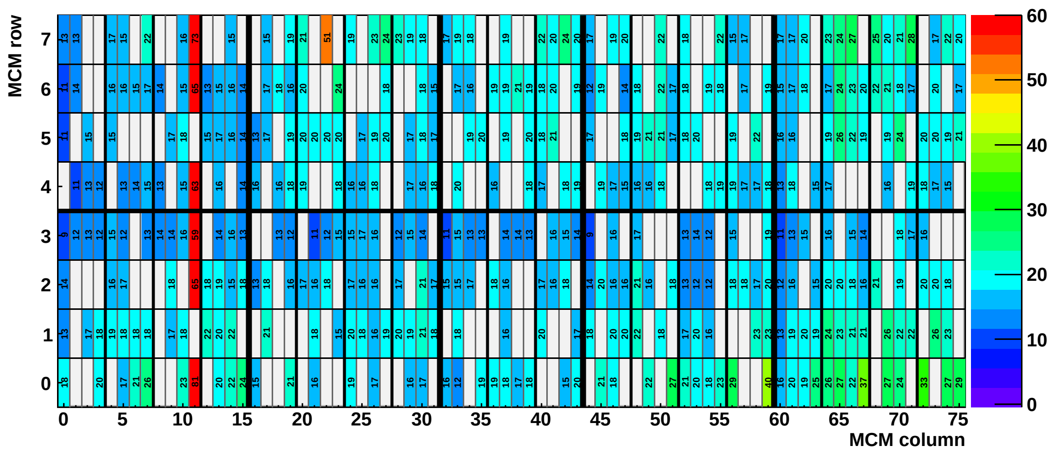


Figure 6.9: Cooling overview of a complete layer with active cooling. The cooling line for the MCMs in column 11 was blocked manually. The affected MCMs are clearly identifiable as uncooled. If such an observation is done after the installation of a layer the problem has to be fixed. Figure 6.8 shows a different plot of the same layer.

- layer−>loadLayer("dir", layer): Load all files from layer layer in directory dir.
- layer−>drawLayer(): Draws a cooling overview for a complete layer. Two plots created with this command are shown in figures 6.8 and 6.9.
- layer−>drawRoc(stack): Draws a cooling overview for one stack.
- layer−>drawCol(col): Draws the temperature sensor readings as function of time for all MCMs in one cooling line (one curve per MCM).
- layer−>reset(): Reset all information. Necessary to switch to another layer.

6.2.3 Calibration of the MCM Temperature Sensors

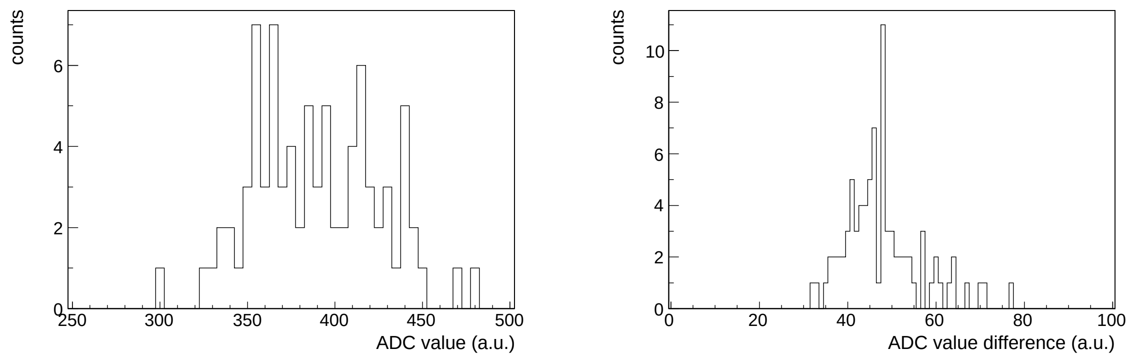


Figure 6.10: Distribution of the absolute MCM temperature sensor readings at a given point in time (left) and the distribution of the differences of the MCM temperature sensor readings measured at two points in time.

The MCM temperature sensors are not calibrated. Figure 6.10 shows the absolute ADC values within one chamber just after switching on the MCMs. In this case one can assume all MCMs have the same temperature and therefore all sensors should report the same value. In contrast to expectation the plot shows a broad distribution of the ADC values. In the second plot of figure 6.10 the difference between the ADC value at two points in time of an uncooled readout chamber is shown. Because the chamber is uncooled one can assume that the temperature difference for all MCMs is the same. Calibrated sensors should give all the same value which corresponds to the temperature change between the two points in time but the plot shows a broad distribution of the values.

The plots in figure 6.10 illustrate that for calibration at least two calibration factors are required:

$$\text{temperature in °C} = a \cdot \text{ADC value} + b \tag{6.4}$$

$$\text{with} \qquad a = \frac{T_1 - T_2}{ADC_1 - ADC_2} \qquad \text{and} \qquad b = T_1 - a_1 \cdot ADC_1 \tag{6.5}$$

Therefore two ADC temperature values (ADC_1, ADC_2) and the corresponding temperatures (T_1, T_2) are required for a calibration. The main challenge is to determine T_1 and T_2.

The temperature inside the chip cannot be measured. Instead one can use the surface temperature of the MCMs. Using the surface temperature has the disadvantage that the temperature gradient in the MCM is unknown. The surface temperature was measured with an infrared camera. The infrared camera registers the infrared radiation emitted by the MCM surface and converts it to a temperature. This measurement principle adds a second uncertainty. The exact infrared emission of a body with fixed temperature depends on its infrared emission coefficient. The emission coefficient of the MCM surface is not known. According to the manual of the infrared camera the emission coefficient for black plastic surfaces is between 0.85 and 0.95 (a ideal black body has an emission coefficient of 1). If the MCMs have the cooling tubes glued on top already measurement the surface temperature is not possible. Aluminum has an emission coefficient of 0.4 only and reflects IR radiation quite well. Third the absolute scale of the uncertainty for the used infrared camera is $\pm 2\,\mathrm{K}$ according to its manual [Gua06]. For these reasons a precise (absolute) calibration of the temperature sensors is not possible.

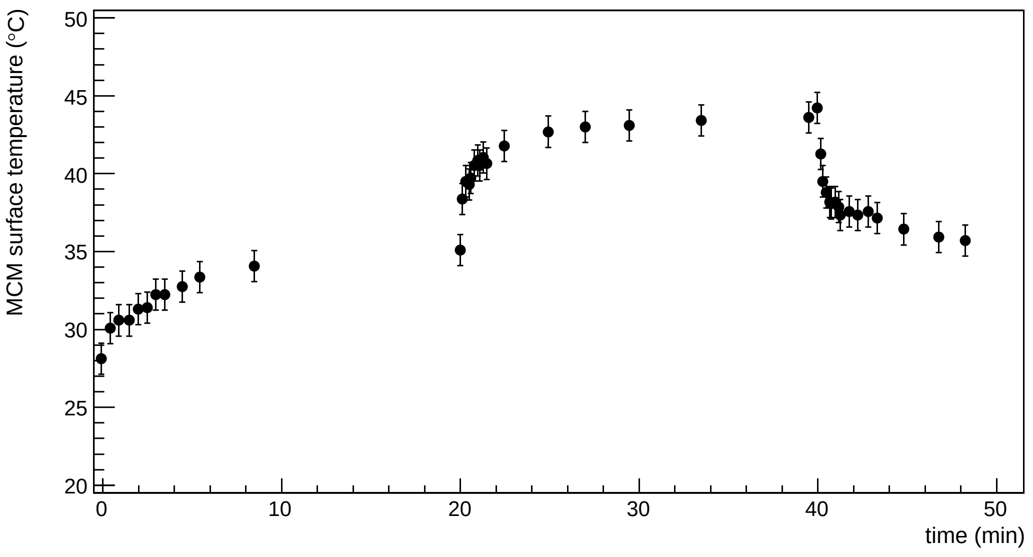


Figure 6.11: MCM surface temperature measured with an IR camera as a function of time.

To get at least an estimate of the calibration parameters the surface temperature at the center of the TRAP chip of an uncooled MCM was monitored during a measurement cycle described in section 6.2.2. The resulting temperature curve is shown in figure 6.11. The temperature can be plotted as a function of the corresponding ADC values, too. In this case one gets the plot shown in figure 6.12. The line in the plot is a fit:

$$T_{IR} = a \cdot \mathrm{ADC} + b, \tag{6.6}$$

with T_{IR} as the temperature measured with the infrared camera and ADC as the corresponding MCM temperature sensor reading.

The values obtained are $a = 0.1377 \pm 0.005\,^\circ\mathrm{C}$ and $b = -27.1 \pm 0.4\,^\circ\mathrm{C}$. The two fit parameters correspond to parameters a and b in equation 6.4. By using these values one has to keep in mind that the determined conversion is only valid for the analyzed MCM and relates the surface temperature with the ADC values. The real temperature measured by the MCM temperature sensor is higher. The result of this analysis is that an MCM temperature sensor value of 400 corresponds to a surface temperature of $28.0 \pm 0.4\,^\circ\mathrm{C}$,

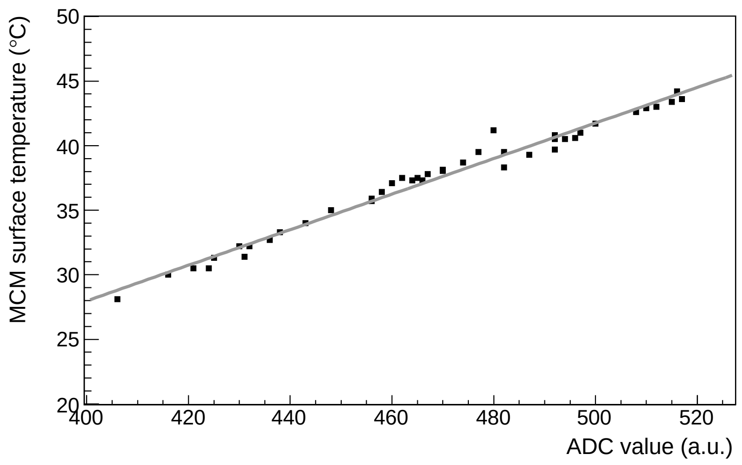


Figure 6.12: MCM surface temperatures measured with an IR camera as a function of the corresponding MCM temperature sensor readings. The straight line is a fit.

a value of 500 corresponds to a surface temperature of 41.8 ± 0.4 °C and a change in the MCM temperature sensor reading of 100 corresponds to a temperature change of 13.8 ± 0.5 °C.

Due to the variation in the gain factor of the temperature sensors a generalization of the just obtained result has to be done very carefully. Figure 6.13 shows the averaged MCM sensor readings of the chamber used for the IR temperature measurement. The first values of all ADCs were set to zero to eliminate constant offset. Additionally, the sensor readings of the two MCMs with the highest and lowest values are shown. The error bars were determined by fitting the distribution of the MCM sensor readings in each time-bin with a Gaussian function and using the σ of the fit as the error of the corresponding data point. Therefore all curves show relative changes. The solid black lines are the scaled temperatures measured with the infrared camera (plot 6.11). The scaling was done with the linear function

$$\mathrm{ADC} = a \cdot T_{IR} + b. \tag{6.7}$$

T_{IR} is the temperature measured with the infrared camera and ADC is the scaled value shown in figure 6.13. The parameters a and b in equation are determined by plotting the measured temperature as a function of the corresponding MCM sensor readings and fitting the data like shown in figure 6.4 already. Table 6.2 shows the obtained values for a and b.

Formula 6.7 can be transposed to T_{IR}. Using the values shown in table 6.2 for a and b, a change of 100 in the MCM temperature sensor readings corresponds to an surface temperature change of about 13.3 °C for the averaged plot, to a temperature change of 9.2 °C for the low MCM and to an temperature change of 15.3 °C for the high MCM.

Taking all obtained results into account, the final result is that a change in the MCM temperature sensor readings of 100 corresponds to an surface temperature change of about 13 ± 2 °C. An absolute calibration of the MCM sensors remains not feasible.

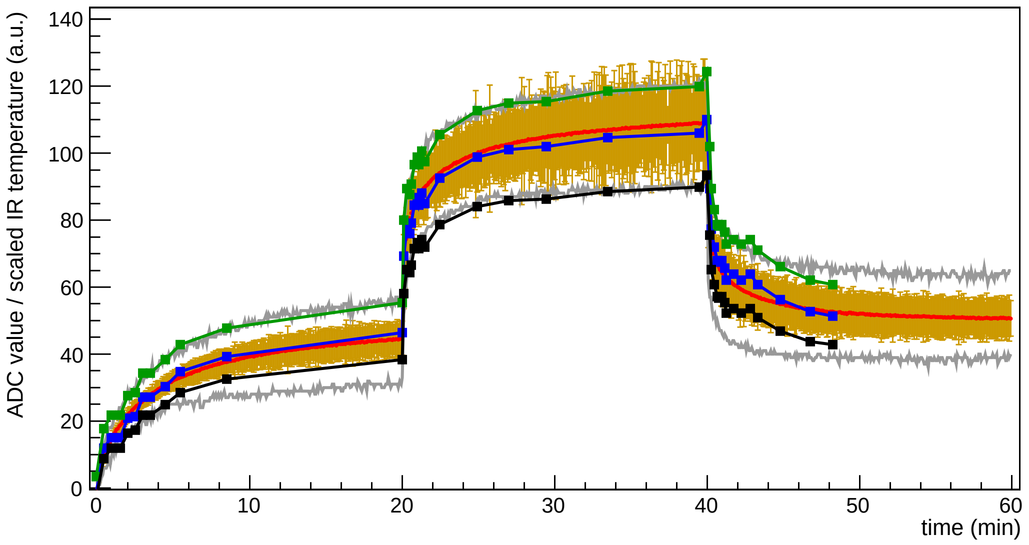
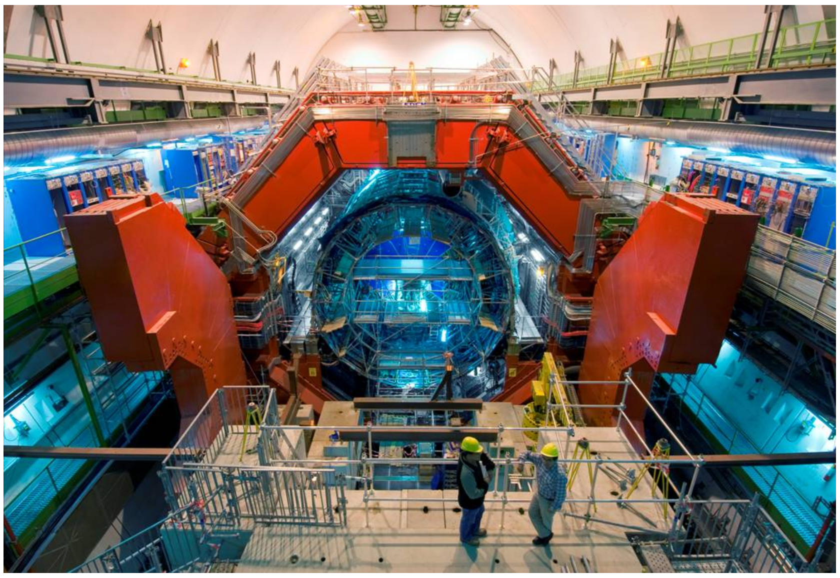


Figure 6.13: The red curve shows the averaged MCM temperature sensor readings of a complete chamber. The corresponding error bars are plotted in orange. Before averaging the first value of all MCMs were set to zero and and only the differences between the first value and the current value of each MCM were averaged. The gray curves are the MCM temperature sensor readings of the MCMs with the highest and lowest changes. The black, blue and red lines / points are the scaled surface temperatures showed in plot 6.11.

MCM curve	fit parameter	value (°C)	uncertainty (°C)
high	*a*	6.9	±0.51
high	*b*	-196	±19
averaged	*a*	7.5	±0.45
averaged	*b*	-207	±17
low	*a*	5.9	±1.2
low	*b*	-173	±45

Table 6.2: IR temperature scaling factors

7 Summary



The ALICE detector.

The ALICE experiment at the Large Hadron Collider at CERN is a dedicated experiment to study the properties of the QGP created in heavy ion collisions. The TRD is one of the detectors in ALICE. Its main tasks are to work as a fast tracker and to provide electron-pion separation at momenta in excess of 1 $^{\mathrm{GeV}}/_{\mathrm{c}}$. The TRD has complex front end readout electronics directly embedded in the detector. The electronics needs to be configured and monitored. One part of the front end readout electronics are the DCS boards, small embedded computers running a Linux operation system. On the Linux system runs the FEEServer and the control engine which control and configure the TRD.

In this thesis the control engine for the transition radiation detector of the ALICE experiment was redesigned and implemented in C++. The original version was written in C and had both a lack of extensibility and features. Therefore the source code was completely rewritten.

A finite state machine and a consistent reporting and error handling system were added. The handling of incoming configurations and the hardware test procedures were improved. Furthermore, the complete SCSN system was rewritten to add the ability to bridge linkpairs and to handle bridged linkpairs. Finally, a class to read out the internal temperature sensors of the MCMs was added.

On the basis of the MCM temperature readout a procedure was developed to identify badly cooled MCMs and blocked cooling lines during supermodule assembly. The identification during the assembly is vital because afterwards it is difficult if not even impossible to fix cooling problems inside a supermodule.

Appendices

A Using the libTRD

The C++ library libTRD provides comfortable interface classes to control the readout chamber. Furthermore it provides some general purpose classes like the logging system. In this appendix an example program is described which illustrates how the libTRD can be used by stand alone programs. The source code is shown in listing A.1.

In the lines 1 to 14 the necessary header files are included. Apart from iostream and vector all other header files belongs to libTRD. First the logging system is initialized (lines 20 to 27). In this example two logging output channels are created. One channel writes all log messages with a log level equal or greater than einfo to standard out (line 21) and the other channel writes all messages, including debug messages, to a file (line 22). After the logging system is initialized, a corresponding debug message is printed (lines 29-30). The message will only show up in the log file but not on the screen because its log level is smaller than einfo.

Then two SCSN commands are created. In line 32 the array which will hold the commands is created and filled afterwards (lines 35 to 43). The vector which will hold the results of the SCSN commands is created in line 46. Next, the SCSN bus is opened, the two frames are send, the results are stored in the created vector and the bus is closed. In case of an error, an exception is thrown by the class SCSNBus. To prevent the program from just quitting in such a case, the exception is caught and printed to tlog (lines 55 to 59). For that output the second highest log-level (eerror) is chosen, because exceptions thrown by class SCSNBus usually indicate serious problems. The far most common cause for such an exception are not powered MCMs. If no error occurred, the if-clause in lines 61 to 72 checks if the value was written correctly and prints out a suitable message. Afterwards the program is finished.

```
#include <iostream>
#include <vector>

#include "scsnbus.hh"
#include "trd_factory.hh"
#include "scsn_ids.h"

#include "errorinfo.hh"

#include "logging.hh"
#include "logoperators.hh"
#include "stream_loglevels.hh"

#include "trd_factory.hh"

using namespace std;

int main(void)
{
    try { // create two output streams
      tlog.addStream("coutlog", new CoutLogBuffer(einfo));
      tlog.addStream("filelog", new FileLogBuffer("/tmp/logfile",
          edebug));
      tlog.setSource("MyProgram");
    }
    catch(...) {
      cerr << "Initialisation of tlog faild" << endl;
    }

    tlog << loglev(edebug) << "Logging system is initialized."
         << endl;

    cfdat_command cmds[2];

    // prepare the SCSN frames
    cmds[0].cmd  = SCSN_CMD_WRITE;  // write to the target MCM(s)
    cmds[0].dest = 127;             // target: all MCMs (broadcast)
    cmds[0].addr = 0x0D46;      // write to the MCM register 0x0D46
    cmds[0].data = 0xACACACAC; // the value which should be written

    cmds[1].cmd  = SCSN_CMD_READ;   // read the target MCM
    cmds[1].dest = scsnids_ali_to_extali(3, 9); // read MCM with
        Ali ID 9 on ROB 3
    cmds[1].addr = 0x0D46;         // read the MCM register 0x0D46
    cmds[1].data = 0;              // no meaning
// preparation finished

    std::vector<FramesResult> result;
    try {
      SCSNBus *scsnbus = trd_factory::instance()->scsnbus();
              // create an instance of class SCSNBus
      scsnbus->openDevice();        // open the SCSN Bus interface
      scsnbus->execute(&cmds, 2); // send the two SCSN frames
    result = scsnbus->getResults(); // get the resulting frames
    scsnbus->closeDevice();      // close the SCSN Bus interface
  }
  catch (int x) {
    tlog << loglev(eerror) << ErrorInfo::getInstance()->getMsg()
       << endl;
    return -1;
  }

  // check the results and print an appropriate message
  if(result[0].data == result[1].data) {
    tlog << loglev(einfo) << "MCM " << result[1].mcm
         << "on ROB " << result[1].rob
         << "has stored the value successfully." << endl;

  }
  else {
    tlog << loglev(eerror) << "Write Error: MCM "
         << result[1].mcm << "on ROB " << result[1].rob
         << "failed to stored the value." << endl;

  }

  return 0;
}
```

Listing A.1: Example code for using the libTRD to access MCMs via SCSN. The program registers two logging streams for output, writes an arbitrary number to an unused memory register of all MCMs, and reads out this memory register afterwards. Finally it is checked if the operation was successful.

B The SCSN Commands and Extended SCSN Commands

SCSN command	command number	description
SCSN_CMD_NETERR	0	Network error reply (invalid checksum, bitstuff error, timeout). The command field of the SCSN frame is set to this number if an error occurred.
SCSN_CMD_READNRQ	1	not used any more
SCSN_CMD_WRITENRQ	2	not used any more
SCSN_CMD_READBNRQ	3	not used any more
SCSN_CMD_BRIDGE	4	activates (data field of the frame = 1) or resets (data field = 0) the bridging mode of the target MCM
SCSN_CMD_CYCLE	5	Bus timing set to fast when data(31)='0' and set to slow when data(31)='1'
SCSN_CMD_RESET	6	reset all settings of the target MCM to default values
SCSN_CMD_NOP	7	No operation, used for ping
SCSN_CMD_PAUSE	8	This command is not send via the SCSN bus but handled by the class SCSNBus. If the data field is greater than 0 the complete control engine halts for the number of micro seconds given in the data field. Otherwise the class SCSNBus creates and send a READ frame which reads out the state machine of the MCM. The second behaviour has historical reasons.
SCSN_CMD_READ	9	Reads out the configuration register given in the address field of the target MCM. The result is stored in the data field of the frame
SCSN_CMD_WRITE	10	Writes the value given in the data field of the frame to the configuration register given in the address field of the target MCM.
SCSN_CMD_UNDEF11	11	has been never used
SCSN_CMD_PRETRIGGER	12	not used any more
SCSN_CMD_POWER	13	not used any more
SCSN_CMD_PREPLY	14	unknown command error reply
SCSN_CMD_SHOWCLK	15	not used any more

Table B.1: List of all SCSN commands. Except for SCSN_CMD_PAUSE all commands which are not marked as outdated are processed by the MCMs.

extended SCSN command	command number	description
SCSN_CMD_ROB_POWER	16	Control ROB power dest = bit 0-7: ROBs (0 means switch off, 1 means switch on) bit 8-11: 0x0: all ROBs 0x1: ROBs on a-side 0x2: ROBs on b-side 0x4: both ROBs with the ORI and HCMs (ROB 4 and 5) 0x8: uses the informations within bis 0-7 addr = delay[0:0xFFFF](ms) data= 0 : ROB is switched off 1 : Only lower MCMs (0...7) are switched on 2 : Keep digital part on 3 : lower MCMs and whole digital part is switched on 4 : Only upper MCMs (8...15) are switched on 6 : upper MCMs and whole digital part is switched on 5,7 : fully functional, all on

SCSN_CMD_ROB_RESET	17	Control ROB reset
		dest=0: addr = delay (ms), data = number of resets
		dest=1: roc_excutor handles reset

SCSN_CMD_UNDEF18	18	not used
SCSN_CMD_TTCRX	19	ttcrx_regs, processed by ttcrx_stat
SCSN_CMD_HW_PTRIGGER	20	send pretrigger <data> (via DCS board)

SCSN_CMD_RUN_TEST	21	Run the test
		data = 0: ni_scsn
		data = 1: ori
		data = 2: bridge
		data = 3: reset
		data = 4: shutdown
		data = 5: read_laser_id
		data = 6: ni_scsn_fast
		data = 7: dmm_test
		data = 8: ddd_test
		data = 9: imm_test
		data = 10: ni_scsn_singlemcm
		data = 11: ping

SCSN_CMD_SETHCID	22	Set half chamber ID
SCSN_CMD_QUERY	23	commands to query state of ROC cf: scsn_query_subcommand

SCSN_CMD_MCMTEMP	24	configure MCM temperature sensor readout addr = 1: sets the type of output format for the MCM temperature sensor readings according to the data field. data= 1: no output but measurement is still active 2: current value 3: baseline 4: diff. between baseline and current value 5: curr. value and diff. between curr. and baseline addr=2: Sets the measurement interval to the value in the data field (in seconds) addr=3: Sets the minimal number of working MCM temperature sensors in a cooling column according to the data field. addr=4: Sets the maximal number of tries to increase the number of working temperature sensors to the value in the data field. addr=5: not used anymore addr=6: Enables (data=0) or disables (data!=0) the MCM temperature readout addr=10: Read out the MCM temperature sensors one time and print the results. The data field determins the output format (see addr=1)
SCSN_CMD_PM_CTRL	25	Patch Maker control commands see patch_maker.hh for definitions
SCSN_CMD_ORI_EEPROM	26	ORI EEPROM Programming, no details defined yet
SCSN_CMD_UNDEF27	27	not used
SCSN_CMD_PT_CTRL	28	Pretrigger Control
SCSN_CMD_JTAG	29	JTAG control (for pretrigger use, and more?)
SCSN_CMD_UNDEF30	30	not used
SCSN_CMD_NV_WRITE	31	NV-write (non-volatile tag)

Table B.2: List of all defined extended SCSN commands. Unused commands are reserved for future extensions

C The Classes of libTRD and trdCE

Block	Class	task
Configuration handling	ROCInfo	Stores general information about the ROC
	trd_factory	Singleton class storing the pointers to SCSNBus, ROCInfo, rstate_reader, patch_maker and ttcrx_stat
Patch maker	patch_maker	Main class of the patch maker
	rob_cnf	Administrates the patches for complete readout boards
	mcm_cnf	Administrates the patches for single MCMs
	patch_maker_gpart	Stores global parameters for the patch-maker system
SCSN bus system	SCSNBus	Main class of the SCSN bus system
	SCSNBusBridge	
	SCSNGuardian	Synchronizes the SCSN bus access using the mutex mechanism
Error handling	ErrorInfo	Singleton class - An error message is streamed to this class each time an error occurred
Logging System	LoggingStream	Main class of the logging system
	LogBuffer	Base class for the different logging output channels
	CoutLogBuffer	Logging output channel which writes to STDOUT
	SyslogLogbuffer	Logging output channel which writes to syslog of Linux
	FileLogBuffer	Logging output channel which writes to arbitrary log files
	LogBufferManager	Administrates the different logging output channels and distributes the logging messages to the channels
	loglev	Provides the stream operator to set the loglevel of a log message
Unassigned	adc_device	Base class for temp_sensor and voltage_sensor
	temp_sensor	Provides access to the DCS board temperature sensor
	voltage_sensor	Provides access to the voltage sensor readings

Table C.1: Classes in the library libTRD

Block	Class	task
Finite state machine	FiniteStateMachine	Base class for finite state machines
	CEStateMachine	Contains the FSM logic and the main control functions
Configuration Handling	ROCControl	Splits the incoming configuration and distributes the parts to the appropriate classes
	TRDCEFactory	Singleton class - provides pointers to the instances of ROCExecutor and TempControl
command execution	ROCExecutor	Executes the extended SCSN commands and forwards normal SCSN commands to class SCSNBus
Temperature Monitoring	TempControl	Main class for the MCM temperature sensor monitoring
	MCMChip	Class to store information about and readings of each MCM temperature sensor - only used by TempControl
	RingBuf	Class implementing a ringbuffer - only used by MCMChip
Test System	TestClass	The base class for all tests
	TestFactory	Factory class to create an instance of the requested test
	TestBridge	Class for the bridge test
	TestLaserID	Class for the laser ID test
	TestMem	Class for the three memory tests
	TestNI	Class for the network interface tests
	TestORI	Class to readout and check the ORI configuration
	TestReset	Class to check if the SCSN reset works correctly
	TestShutdown	Class to check the power regulators for the MCMs
	SCSNCommandStack	Provides functions to generate SCSN commands and stores them in a buffer to send all stored commands at once. Used only by the test functions.
ORI System	I2C	Contains functions to access the ORIs via I2C bus
	J2C	Contains functions to access the ORIs via the J2C bus
	ORIControl	Provides functions to communicate with the ORIs; the only class which uses I2C and J2C
	ORIResult	Data class to store and convert values read out from the ORIs. Used by the ORI test
Logging system	ODimLogBuffer	Class to write log messages to DIM channels

Table C.2: Classes in the library trdCE

Bibliography

[A$^+$05] V. Angelov et al. ALICE TRAP User Manual. http://www.kip.uni-heidelberg.de/ti/TRD/doc/trap/TRAP-UserManual.pdf, 2005. Revision 1.1.

[ABMS06] A. Andronic, P. Braun-Munzinger, and J. Stachel. Hadron production in central nucleus-nucleus collisions at chemical freeze-out. *Nucl. Phys. A*, pages 167–199, 2006.

[ALI] ALICE homepage: The ALICE experiment. http://aliceinfo.cern.ch/Public/en/Chapter2/Chap2Experiment-en.html. Dec. 2008.

[And04] A. Andronic. Electron identification performance with ALICE TRD prototypes. *Nuclear Instruments and Methods in Physics Research Section A*, 522:40–44, 2004.

[Ang06] V. Angelov. Design and performance of the ALICE TRD front-end electronics. *Nuclear Instruments and Methods in Physics Research Section A*, 563:317–320, 2006.

[AoG08] C. Amsler, others, and (Particle Data Group). PL B667, 1, 2008.

[ATT99] ATLAS detector and physics performance. Technical design report. Vol. 2. 1999. CERN-LHCC-99-15.

[B$^+$] R. Brun et al. The ROOT system. http://root.cern.ch.

[BBO] Homepage of the BusyBox project. http://www.busybox.net. Jan. 2009.

[BDNFP06] Austin Ball, Michel Della Negra, L Foà, and Achille Petrilli. *CMS physics: Technical Design Report*. Technical Design Report CMS. CERN, Geneva, 2006.

[BMS07] P. Braun-Munzinger and J. Stachel. The quest for the quark-gluon plasma. *Nature*, pages 302–309, 2007.

[Car97] Jean-Luc Caron. Layout of the LEP tunnel including future LHC infrastructures. AC Collection. Pictures from 1992 to 2002., 1997.

[CCC$^+$04] F. Carena, W. Carena, S. Chapeland, R. Divià, J-C. Marin, K. Schossmaier, C. Soós, P. Vande Vyvre, and A. Vascotto. THE ALICE EXPERIMENT CONTROL SYSTEM, 2004. CHEP, Interlaken.

[CER00] CERN. New State of Matter created at CERN. *CERN press release PR01.00*, 2000.

[CER08] CERN Communication Group. CERN faq – LHC the guide. http://cdsmedia.cern.ch/img/CERN-Brochure-2008-001-Eng.pdf, 2008.

[DA07] D. Wegerle D. Antonczyk, H. Appelshäuser. A gas monitor for the ALICE TRD, 2007.

[Dol93] B. Dolgoshein. Hadron production in central nucleus-nucleus collisions at chemical freeze-out. *Nucl. Phys. A*, pages 434–469, 1993.

[ECT08] ALICE Electromagnetic Calorimeter Technical Design Report. 2008. CERN-LHCC-2008-014.

[EMT] Homepage of the company EMT. http://www.etm.at/. Jan. 2009.

[G$^+$] D. Gottschalk et al. DCS Board Schematics Version 1.64. http://www.kip.uni-heidelberg.de/ti/DCS-Board/current/schematics/pdf/ver164/DCS164.pdf.

[GA03] The ALICE-TOF Group and P. Antonioli. The ALICE Time of Flight System. *Nuc. Phys. B*, 125:193–197, 2003.

[Gar02] R. Gareus. Slow Control - Serial Network and its implementation for the Transition Radiation Detector, 2002.

[GD93] C. Gaspar and M. Dönszelmann. DIM – A Distributed Information Management System for the Delphi experiment at CERN. In *Proceedings of the IEEE Eight Conference REAL TIME '93 on Computer Applications in Nuclear, Particle and Plasma Physics*, Vancouver, Canada, 1993.

[GF97] C. Gaspar and B. Franck. SMI++ - Object Oriented Framework for Designing Control Systems for HEP Experiments. In *Presented at: CHEP 97 - International Conference on Computing for High Energy Physics*, Berlin, Germany, 1997.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Longman, Amsterdam, 1994.

[GKLT05] D. Gottschalk, T. Krawutschke, V. Lindenstruth, and H. Tilsner. Single Board Computer for the ALICE DCS. *GSI Scientific Report 2004*, page 361, 2005.

[Gua06] Guangzhou SAT Infrared Technology Co., LTD. SATIR HotFind Infrarotkamera Bedienungsanleitung, 2006.

[ISE] Homepage of the company ISEG. http://www.iseg-hv.de. Jan. 2009.

[Jac75] J. D. Jackson. *Classical Electrodynamics*. John Wiley & Sons, Inc., New York, 1975.

[KB04] Ch. Klein-Bösing. *Production of Neutral Pions and Direct Photons in Ultra-Relativistic Au + Au Collisions*. PhD thesis, Universität Münster, 2004.

[LARP06] V. Lindenstruth, V. Angelov, F. Rettig, and P. Popov. Optical Readout Interface for the ALICE TRD. *GSI Scientific Report 2005*, page 289, 2006.

[Lip06] C. Lippman. The ALICE transition radiator. In *SNIC symposium*, Stanford, California, 2006.

[Mah04] T. Mahmoud. *Development of the Readout Chamber of the ALICE Transition Radiation Detector and Evaluation of its Physics Performance in the Quarkonium Sector*. PhD thesis, Universität Heidelberg, 2004.

[MS86] T. Matsui and H. Satz. J/ψ suppression by quark-gluon plasma formation. *Physics Letters B*, 178(4):416 – 422, 1986.

[NA500] NA50 collaboration. Evidence for deconfinement of quarks and gluons from the J/ψ suppression pattern measured in Pb-Pb collisions at the CERN-SPS. *Physics Letters B*, 477:28 – 36, 2000.

[RS06] I. Rusanov and J. Stachel. The Readout Boards for the ALICE TRD. *GSI Scientific Report 2005*, page 287, 2006.

[Sat90] H. Satz. Color screening and quark deconfinement in nuclear collisions. *Adv. Ser. Direct. High Energy Phys.*, 6:593–630, 1990.

[ST01] F. D. Steffen and M. H. Thoma. Hard thermal photon production in relativistic heavy ion collisions. *Phys. Lett. B*, page 98, 2001.

[TPC00] TPC Technical Design Report. 2000. CERN/LHCC 2000-001.

[TRD01] ALICE TRD Technical Design Report. 2001. CERN-LHCC-2001-21.

[UCL] Homepage of the uClibc project. http://www.uclibc.org. Jan. 2009.

[Web07] C. Weber. UrQMD Animations. http://th.physik.uni-frankfurt.de/~weber/CERNmovies/index.html, 2007. Universität Frankfurt.

[YHM05] K. Yagi, T. Hatsuda, and Y. Miake. *Quark-Gluon Plasma*. Cambridge University Press, Cambridge, 2005.