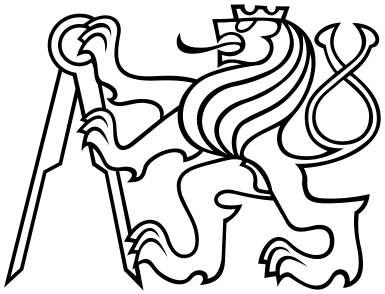
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science



Master's thesis

Remote user interface for the control system of the COMPASS experiment at CERN

Bc. Antonín Květoň

Supervisor: Ing. Tomáš Černý, Ph.D.

Study programme: Open Informatics

Specialization: Software Engineering

2 May 2017

Acknowledgment:
I would like to thank Ing. Tomáš Černý, Ph.D. for supervising my master's thesis, prof. Ing. Miroslav Finger, DrSc. for making my journeys to CERN possible and all my CERN colleagues for always being so willing and enthusiastic to explain any complex subject to me.

Prohlášení:
Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 2. května 2017 Antonín Květoň

Declaration:
I hereby declare that I have written the submitted thesis myself and quoted all sources used in accord with the Methodical directive concerning ethical principles regarding academic theses.

Prague, May 2, 2017 Antonín Květoň

Název práce česky:

Vzdálené uživatelské rozhraní pro řídící systém experimentu COMPASS v CERN

Autor: Bc. Antonín Květoň

Program: Otevřená Informatika

Obor: Softwarové Inženýrství

Druh práce: Diplomová práce

Vedoucí práce: Ing. Tomáš Černý, Ph.D., FEL ČVUT v Praze

Abstrakt: Částí řídícího systému sběru dat experimentu COMPASS v CERN je grafické uživatelské rozhraní, které je navrženo s ohledem na lokální přístup – jedinou možnou metodou vzdáleného přístupu je protokol Secure Shell s X11 přesměrováním. Nicméně, k tomuto systému je ve skutečnosti často přistupováno vzdáleně, což tento způsob přístupu činí nevhodným pro uživatele nacházející se v síťovém prostředí s nízkou šířkou pásma a vysokou latencí. Tato práce se zabývá analýzou funkcionality tohoto grafického uživatelského rozhraní a zhodnocením nejvhodnější metody vzdáleného přístupu. Nejvhodnějším řešením je zvoleno konzolové uživatelské rozhraní a zbytek této práce se zabývá jeho návrhem, implementací a testováním.

Klíčová slova: CERN, COMPASS, DAQ, GUI, CLI

Title:

Remote user interface for the control system of the COMPASS experiment at CERN

Author: Bc. Antonín Květoň

Abstract: The data acquisition control system of the COMPASS experiment at CERN includes a graphical user interface which is designed to be accessed locally, the only method of accessing it remotely being the Secure Shell protocol with X11 forwarding. However, the system is in fact often accessed remotely and this approach is unsuitable for users in a low-bandwidth, high-latency network environment. This work is concerned with analysis of the functionality of the graphical user interface as well as evaluation of the most suitable approach to remote access. A command-line interface is chosen as the most suitable solution and the rest of this work deals with its design, implementation, and testing.

Key words: CERN, COMPASS, DAQ, GUI, CLI

Contents

Introduction

In Chapter 1, this work gives a short account of the COMPASS experiment at CERN, its data acquisition system and the related control system, including an overview of the dependencies used.

The graphical user interface of the control system is thoroughly analyzed in Chapter 2, providing an overview of its functionality as well as descriptions of related concepts and systems as the Chapter proceeds. A complete pictorial description of the interface is shown in Chapter 3.

Chapter 4 deals with analysis of the internal implementation of the graphical user interface, general requirements for remote access, and CERN security restrictions. It then provides a list of possible solutions to the problem and presents arguments as to why a command-line interface can be considered the most suitable solution.

The aim of Chapter 5 is to provide a short summary of the requirements for the command-line interface as well as a complete overview of the functionality of the related application developed as a part of this Master's thesis. Chapter 6 describes its internal implementation and Chapter 7 deals with its testing.

The design, implementation, deployment, and testing of the command-line interface comprise the practical part of this work and were carried out solely by its author.

Chapter 1

COMPASS experiment

COMPASS (Common Muon and Proton Apparatus for Structure and Spectroscopy) is a fixed-target experiment at the Super Proton Synchrotron (SPS) accelerator at CERN near Geneva, Switzerland. It was proposed and approved in 1996, commissioned in 2001, and started taking physics data in 2002. The main objective of the experiment is study of hadron structure and spectroscopy using high intensity muon and hadron beams [1].

The lifetime of the COMPASS program was extended in 2012, and the experiment has been operating under the name COMPASS II since, utilizing phenomena such as Deep Virtual Compton Scattering, Hard Exclusive Meson Production, Semi-inclusive Deeply Inelastic Scattering, Drell-Yan process, and Primakoff reactions in order to make progress in carrying out its objective [2].

The spectrometer of the experiment consists of a number of detectors used for particle identification, tracking and energy measurements. Particle detectors used at COMPASS include, but are not limited to: electro-magnetic calorimeters, hadron calorimeters, gaseous electron multipliers, micro mesh gas detectors, drift chambers, multi-wire proportional chambers, scintillating fiber stations and a ring-imaging cherenkov detector [1, 2].

The experiment is currently planned to operate until the end of 2018 [CERN research board meeting, 7 December 2016] and the facilities are subsequently expected to undergo another lifetime extension, operating under a completely new name and classification [3].

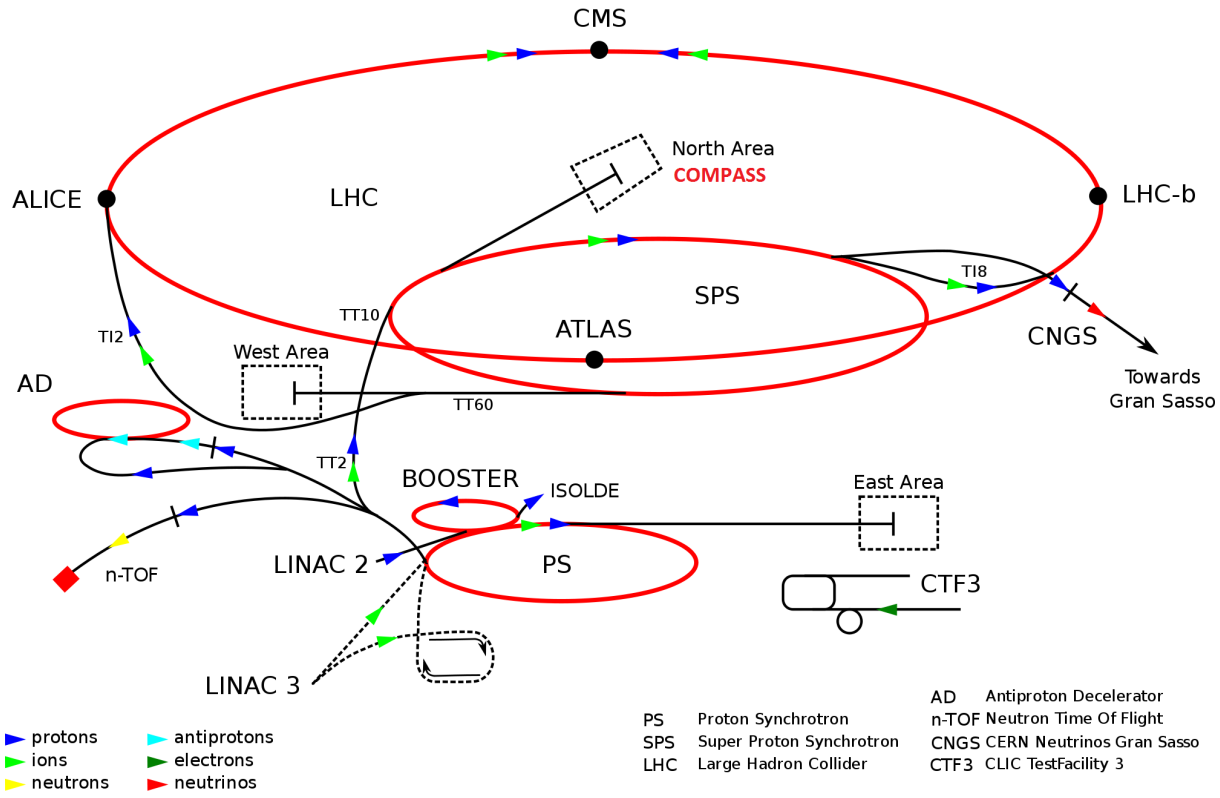


Figure 1.1: COMPASS location within the CERN accelerator complex [4]

1.1 COMPASS data acquisition system

The current COMPASS data acquisition system (DAQ) has been in use since 2014, replacing the old DAQ, which had been in use since the conception of the experiment. Its objective is to read out raw data concerning physics events[1] from detectors and store them on hard drives. These data are subsequently sent to the CASTOR (CERN Advanced STORage manager) system – a hierarchical storage management system for managing physics data files in the order of petabytes, using both tape and hard drive storage [5].

Since the number of physics events occurring in collider and fixed-target experiments is generally far too large to store and subsequently analyze, systems referred to as trigger systems (TS) are utilized in such experiments, including COMPASS. When a physics event is captured by the detectors, the TS makes a decision whether the event is to be read out from the detectors or discarded. The decision is made in real time based on several criteria which ensure that only events relevant to physics analysis are captured, decreasing the amount of recorded data by several orders of magnitude. A typical example is filtering of physics events caused by cosmic rays [6].

[1] A physics event is the outcome of a fundamental interaction between subatomic particles

A single physics event is almost always described by readings from multiple detectors, whose number of output channels is in the order of thousands – in total, there are about 300 000 channels from which the DAQ receives data. The channels are then multiplexed, using three layers of multiplexers:

1. HGeSiCA, CATCH and Gandalf modules [7, 8, 9]
2. Slink multiplexers and TIGER VXS data concentrators
3. FPGA multiplexers [10]

After the multiplexing process is finished, the data are sent into an FPGA switch which handles event building. The events are recombined during the event building process – data concerning each event, despite originating in multiple detectors counting thousands of channels, are unified into a single continuous stream in a single channel.

Finally, the data are read out by readout computers using Spillbuffers and stored on hard drives. A Spillbuffer is a PCI express card with an FPGA chip which handles the connection of an S-Link (a CERN standard for connection of multiple layers of front-end and readout electronics) [23] and the RAM of a computer.

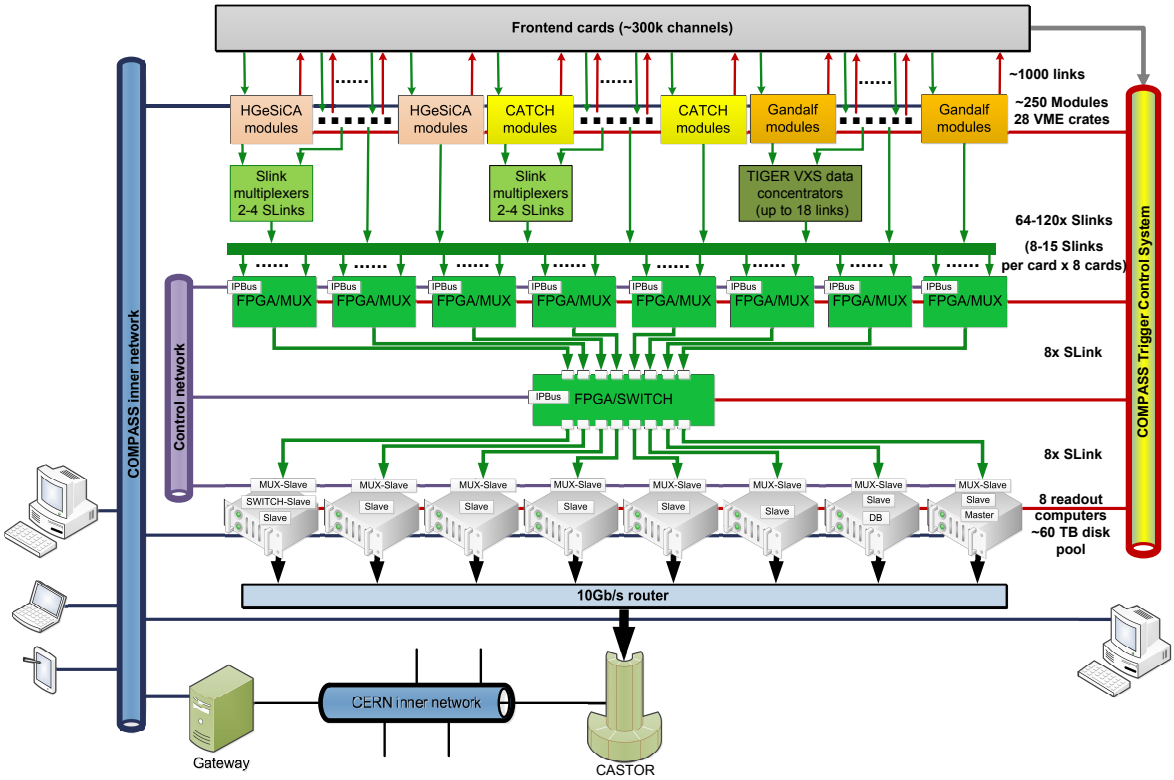


Figure 1.2: Hardware structure of the DAQ [11]

1.1.1 User profile

There exist numerous software systems which are used to control and/or monitor various parts of the COMPASS experiment. For example, the CESAR (CERN Experimental areas SoftwAre Renovation) system is used to control aspects pertaining to the beam line, allowing the user to alter the character of the particle beam up to a certain degree [12]. Another example is the DCS (Detector Control System) [13], which provides detailed real-time information concerning the numerous detectors of the experiment, as well as the capability to adjust their low-level hardware settings, such as various voltages.

When the experiment is in the phase of data taking, it is necessary that operators are present in the control room in order to oversee the process of data taking as well as the condition of the experiment. Since the experiment takes data 24 hours a day, 7 days a week during a physics data-taking period (with the exception of emergency and machine development stops), continuous presence of staff at the site is ensured using a shift-work model. A shift lasts 8 hours and two staff members are present during it. There are 3 shifts per day throughout all days of the week. The staff are normally members of the COMPASS collaboration (each member institution of the COMPASS collaboration is obligated to carry out a given number of shifts).

As the COMPASS experiment is vastly complex, the staff members are normally experts in a single area of the experiment – their knowledge of the other areas might be limited or very basic. According to [14], over 100 different collaboration members partook in data-taking shifts in 2016 alone. The training to become an operator consists of a single 8-hour training shift and cannot possibly explain all the intricacies behind the experiment – most staff learn "on-the-go".

The COMPASS DAQ control system is one of the previously mentioned software systems – it is used to oversee the data taking process. Being central to data taking, it is the system the staff come into contact with most frequently – for this reason, it is necessary for its usability to be held up to a certain standard.

1.1.2 The COMPASS DAQ control system

The DAQ control system is a software system used for control and monitoring of the final three layers of the DAQ hardware (FPGA multiplexers, FPGA switch, and readout engines). It consists of several processes:

- **Slave readout** – a process which runs on the readout computers and handles processing of physics data
- **Slave control** – a process which runs on the readout computers and handles monitoring and configuration of the FPGA cards (including the spillbuffers), using direct communication through IPbus (an interface for UDP-based control of Ethernet-attached hardware devices) [15]
- **Run control GUI** – A graphical user interface used to control the DAQ, described in Chapters 2 and 3
- **Master** – a process which runs on a dedicated computer and acts as the mediator of communication between the DAQ processes. It incorporates state control of the DAQ control system and most of the error handling [16].

- **Message logger** – a process which collects messages from the slave and master processes and stores them in a database (the messages are divided into four types: information, warning, error, and fatal error)
- **Message browser** – a graphical user interface which allows the user to view messages from the slave and master processes in real time or retroactively browse the messages stored in a database

1.1.2.1 State machines

The Master and each of the slave processes all have a state machine associated with them, serving as an indicator of the nature of the process' activity at a given time. This Chapter only provides a diagram of the Master process state machine for informative purposes, as the subject is rather complex and has already been described to great detail in [16].

1.1.2.2 Dependencies

The DIALOG communication library

The DAQ control system uses a custom-built communication library for network communication – DIALOG. It is implemented in C++, using the Qt framework and offering a C++ API. This library replaced the DIM (Distributed Information Management) library [17, 18, 16], which had been in use until April 2016. While all the processes of the DAQ use DIALOG for communication with one another, some of the DAQ processes are still dependent on the DIM library, as several other COMPASS systems, such as the Trigger Control system, still utilize it for network communication.

Similarly to the DIM library, the DIALOG library is based on the philosophy of services and commands and a control server which functions as the mediator of initial connection information.

Services serve as a means of 1 to n communication. Once a process registers a service on the control server, it becomes a "publisher" – any data published using this service will be sent directly to all processes subscribed to the service. Other processes can subscribe to the service by requesting network information concerning it from the control server and connecting directly to the publisher after having received it.

Contrary to DIM commands, which serve as a means of 1 to 1 communication, DIALOG commands serve as a means of 1 to n communication. Once a process registers a command on the control server, it will receive data sent by any process to the control server using that given command – the data is forwarded to it by the control server. Commands are different from services in that the sender is not required to possess a list of receivers, reducing overhead resource costs at the expense of higher strain on network resources of the control server [O. Šubrt (CERN), personal communication, August 2016].

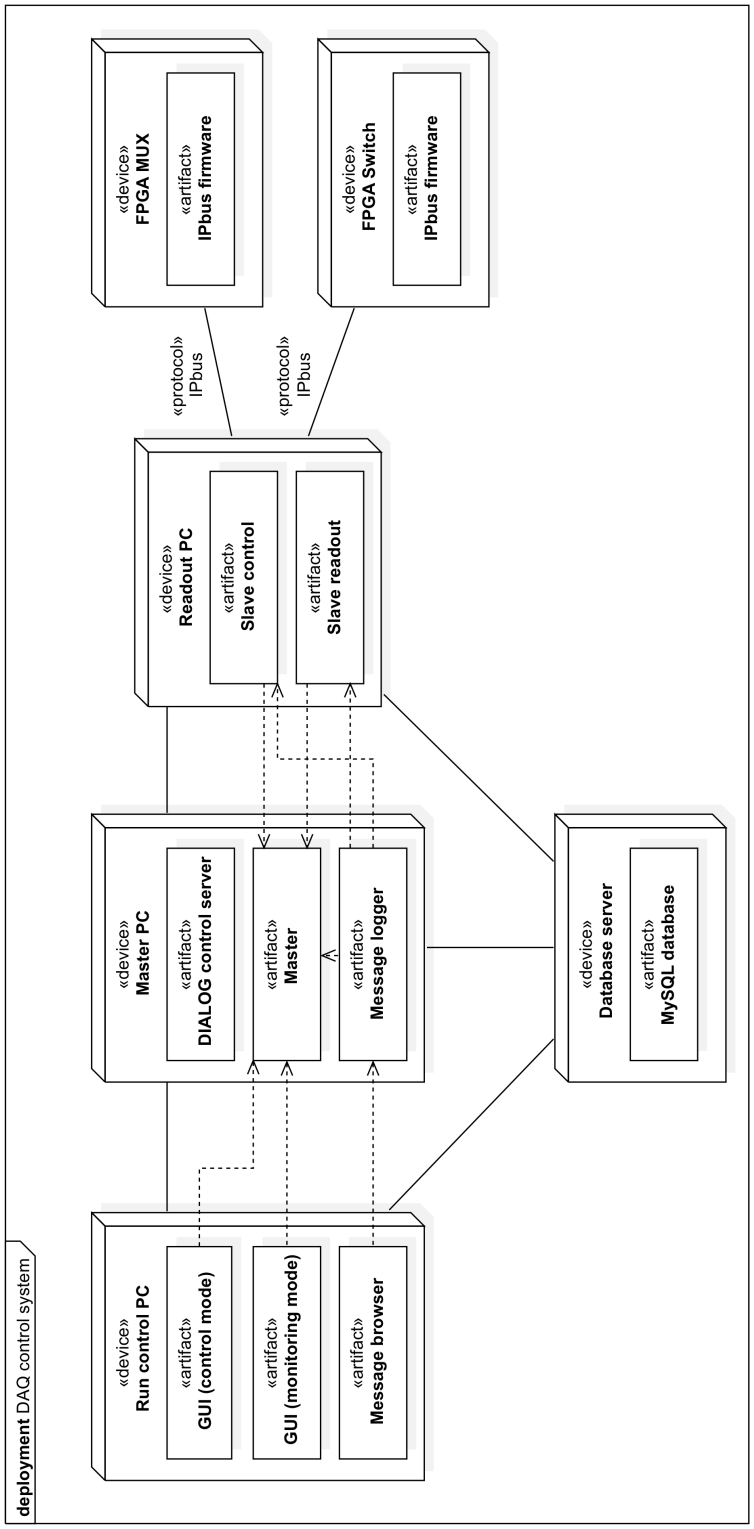


Figure 1.3: The deployment diagram of the DAQ control system. Dependencies related to the DIALOG control server have been omitted from this figure

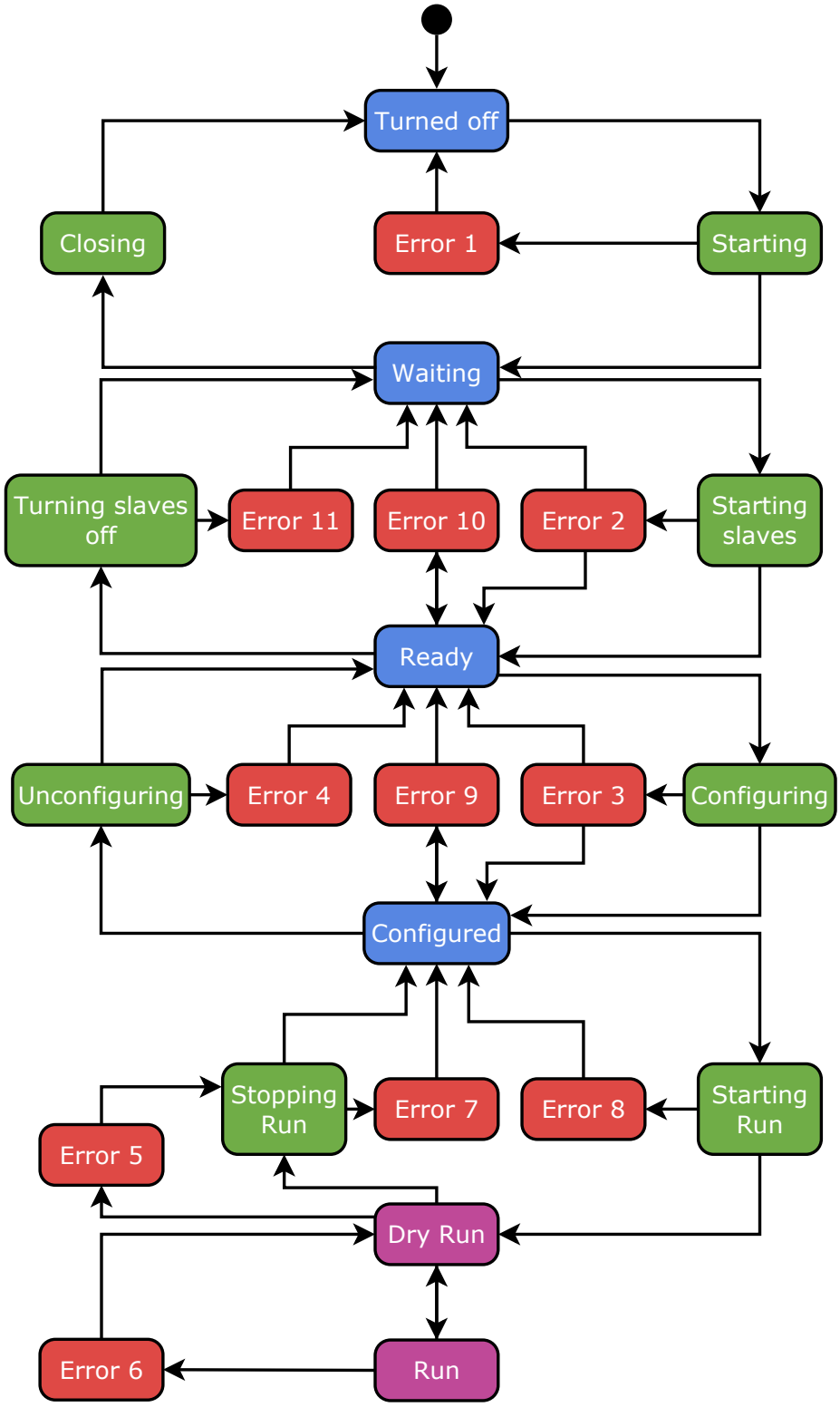


Figure 1.4: The state machine diagram of the Master process [16]

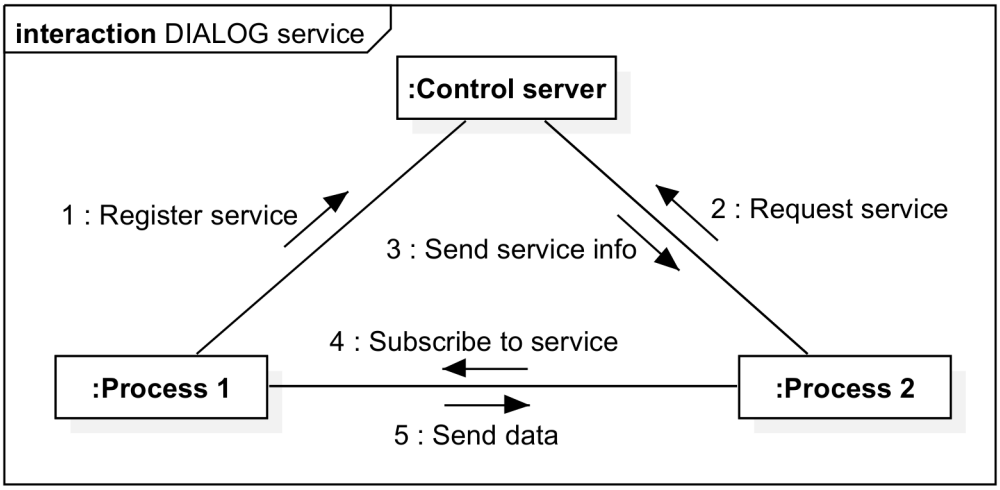


Figure 1.5: The concept of network communication using services in DIALOG

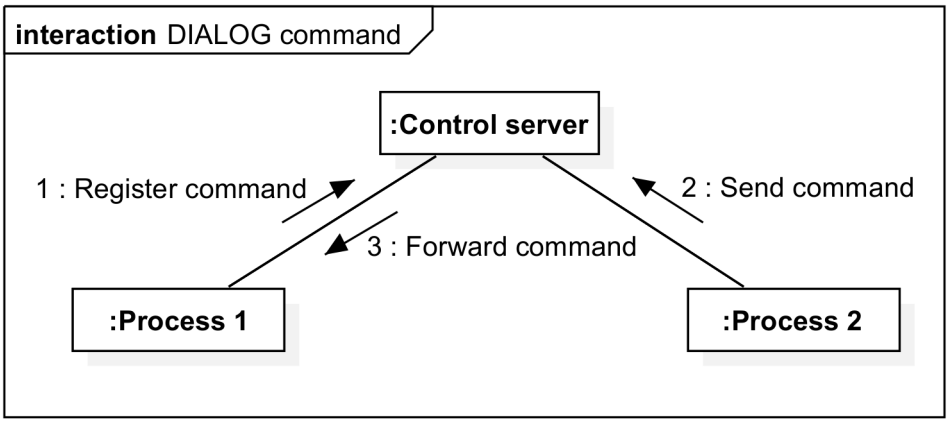


Figure 1.6: The concept of network communication using commands in DIALOG

Qt framework

The Qt framework is a cross-platform application framework originally conceived by Trolltech in 1991. As of 2017, the development is supervised by The Qt Company. While the main purpose of the framework is development of applications with graphical user interfaces, it also provides tools for development

of applications without graphical user interfaces [19]. The Qt framework is utilized in all processes of the DAQ control system [4].

Graphical user interaces in the Qt framework

The Qt framework uses the *ui* format to describe structure of graphical user interfaces. Internally, the *ui* format is an XML file specifying the types and properties of individual Qt widgets present within a given window. A Qt widget is a single user interface element which can display data and status information, receive user input, and provide a container for other widgets that should be grouped together. The Qt framework provides a basic set of Qt widgets to create classic desktop-style graphical user interfaces. The common base class for all Qt widgets is *QtWidgets*. It provides capability to render to the screen and handle user input events. A *ui* file can be either written manually, or automatically generated using the Qt Creator IDE.

The Qt framework uses a preprocessor tool called the User Interface Compiler (uic) to generate C++ header files associated with given *ui* files. Each window has a class associated with it, creating a framework hot spot for implementation of graphical user interface application logic. Individual widgets can be accessed through a private pointer attribute of the class. User input events are represented as signals whose corresponding private slots are situated within the given window class [19].

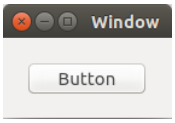


Figure 1.7: A simple window created in Qt (Operating system: Ubuntu 14.10)

```
class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private slots:
    void on_pushButton_clicked();

private:
    Ui::MainWindow *ui;
};
```

Listing 1.1: The class associated with the window in Figure 1.7

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>MainWindow</class>
 <widget class="QMainWindow" name="MainWindow">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>140</width>
    <height>66</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>Window</string>
  </property>
  <widget class="QWidget" name="centralWidget">
   <widget class="QPushButton" name="pushButton">
    <property name="geometry">
     <rect>
      <x>20</x>
      <y>20</y>
      <width>99</width>
      <height>27</height>
     </rect>
    </property>
    <property name="text">
     <string>Button</string>
    </property>
   </widget>
  </widget>
 </widget>
 <layoutdefault spacing="6" margin="11"/>
 <resources/>
 <connections/>
</ui>
```

Listing 1.2: The *ui* file associated with the window in Figure 1.7

Chapter 2

Run control GUI

As mentioned previously, a graphical user interface is part of the COMPASS DAQ control system. Being a separate process, the GUI[1] process can be launched or terminated independently of the other DAQ processes. The GUI serves for both control and monitoring purposes. While there can exist at most a single instance of the GUI which is allowed to control the DAQ, the maximum number of instances which provide only monitoring capabilities is not explicitly limited. This is made possible by the two modes the GUI provides – Control and Monitoring. All GUI instances start in the Monitoring mode and the user is able to "lock into" the GUI into the Control mode at any time, without the need for in-application authentication. If there is an instance of the GUI already in the Control mode when a lock-in happens in a different instance of the GUI, the former will transition back into the Monitoring mode. A messaging feature is provided in order that "lock-in wars" are less likely to occur.

2.1 DAQ monitoring functionality

A list containing descriptions of individual DAQ monitoring features of the GUI follows.

2.1.1 State machine monitoring

The GUI provides the user with information concerning states of all the state machines of the DAQ, Master and Slave processes alike.

2.1.2 Trigger control system channel monitoring

The Trigger Control System (TCS) connects the trigger logic with the DAQ [6]. The GUI allows the user to monitor 12 trigger channels of the TCS. In particular, three values are provided for each channel

[1] While this acronym is normally used to represent the generic term "graphical user interface", it is important to note that the name of the application itself is also "GUI". So as to avoid confusion, the acronym GUI, as well as the phrase "the GUI", will be used solely when referring to the application name. Therefore, when referring to the generic meaning of the term "graphical user interface", this acronym will not be used.

– *In*, *Out* and *Div*. *In* represents the number of events read out by a given trigger in the last spill[2], *Div* represents the setting of the prescaler, i.e., the inverse proportion of events to be purposefully discarded, and finally, *Out* represents the number of retained events in the last spill. In an ideal case, these variables will behave according to the following formula:

$$Out = \frac{In}{Div}$$

2.1.2.1 TCS overview

The GUI displays values read out from the TCS, containing the following information for each active DAQ[3]:

- DAQ id
- Whether a run[4] is in process
- Number of physics events recorded

2.1.3 FPGA register monitoring

The GUI allows the user to view the register values of the FPGA cards involved in the new[5] DAQ in real time. A multitude of register values is provided for each FPGA and all its ports, most of which manifest elsewhere in the GUI, taking on a more accessible form. This functionality is therefore mostly used only by DAQ experts for fault-finding.

2.1.4 DAQ hardware link status monitoring

The GUI provides hardware link status monitoring of the new DAQ. The term "link status" represents the status of data flow between individual hardware components of the new DAQ. The three types of hardware components being monitored are listed below:

FPGA multiplexers

The GUI provides the following information concerning the FPGA multiplexers:

[2]A spill is the extraction period of the SPS accelerator cycle, also referred to as on-spill time. During this period, physics data concerning collisions of the extracted particles with the fixed target are produced. The entire SPS accelerator cycle consists of injection, acceleration, and extraction of the particles. Cf. [1], [6], and [21] for more information.

[3]The TCS controller (cf. [4]) can manage more DAQs at once. The id of the new COMPASS DAQ is 0.

[4]A run is the period of time between the start and stop of data taking. Physics data concerning individual spills are taken during a run. While the number of spills per run is arbitrary, a run normally consists of 200 spills. This term can also be used to refer to a period of calendar time between the start and stop of formally declared SPS activity – e.g. the 2014 run, which lasted from October 6 to December 15.

[5]"New DAQ" refers to the DAQ system developed in [4] and [11] and deployed in 2014. When using this expression in relation to the DAQ hardware, this refers to the FPGA multiplexers, FPGA switch and event-builder, and the readout engine.

- **Source id errors** – should an error occur on a port associated with a given source id[6], the GUI will warn the user and provide information concerning the error.
- **Memory usage** – each port of an FPGA multiplexer is assigned a segment of the on-board memory. The occupancy of this segment is shown.
- **Accepted data** – the number of accepted incoming 32-bit words in the last and current spill on a given FPGA card and all its individual ports.

FPGA switch

The information provided concerning the FPGA switch is analogous to the information provided concerning the FPGA multiplexers.

Readout computers

The following information pertaining to the readout computers can be retrieved from the GUI:

- **Port errors** – data concerning port errors as well as propagated port errors (cf. section 2.1.4.2)
- **Spillbuffer usage** – information related to the occupancy of the Spillbuffer associated with a given readout computer.
- **Readout computer hardware monitoring information** – information concering the CPU and RAM loads of the readout computers as well as the availability of HDD storage.
- **Accepted data** – the number of accepted incoming 32-bit words in the last and current spill on a given readout computer.

2.1.4.1 Port errors

There are several types of port errors which are detectable by the system. Each batch of data sent along the DAQ hardware consists of an S-link header and body (cf. [23, 16]). The S-link header contains meta-information concerning the data, as well as 11 error bits. A listing of errors associated with these bits follows:

1. **Timeout on port** – Each port has a timeout value associated with it – should an event not be sent within a given period of time of a trigger activating, this error will occur. The timeout value is carefully tailored to each specific port – values too large could lead to an event being attributed to an incorrect trigger activation, resulting in desynchronization of the DAQ.
2. **Event size mismatch on port** – The size of the event data does not correspond to the size specified in the header. This is normally a result of front-end electronics sending incorrect data, or data being incorrectly trimmed in the communication process.

[6] A source id is a unique integer used to identify detector front-end electronics as well as DAQ electronics.

3. **Event number mismatch on port** – Event number in received data does not correspond to the one specified in the header, indicating DAQ desynchronization.
4. **Spill number mismatch on port** – Spill number in received data does not correspond to the one specified in the header, indicating DAQ desynchronization.
5. **Event type mismatch on port** – Event type in received data does not correspond to the one specified in the header, indicating DAQ desynchronization.
6. **First S-Link event word is not an S-Link header word** – The data lack a header, indicating the data being incorrectly trimmed in the communication process.
7. **Maximum event size exceeded** – The size of the body has exceeded a pre-defined size.
8. **TCS FIFO full** – The TCS FIFO is full, indicating that the new DAQ hardware is receiving events faster than it can process them.
9. **DDR memory full** – This indicates 100% occupancy of the FPGA memory associated with a given port or a given spillbuffer.
10. **Slink CRC error** – The S-link checksum is incorrect, indicating corruption of data during the communication process.
11. **Control bit set but no control word** – This indicates a serious firmware malfunction.
12. **Unexpected FE SourceID** – Source id specified in received data does not correspond to one specified in the header, indicating incorrect DAQ structure type configuration (cf. section 2.2.4) or a critical frontend error (such errors can cause a corrupted source id to be sent).

2.1.4.2 Port error propagation

Whenever an error occurs in the new DAQ hardware, it propagates down the data processing chain. This phenomenon takes place owing to the fact that whenever several channels of data are being merged, the firmware checks whether any of the error bits are set to 1. Should this be true for any error bit, its value in the newly merged stream will also be set to 1.

Errors can also propagate by a cascade effect – errors associated with DAQ desynchronization have a tendency to cause different errors farther down the readout chain.

2.2 DAQ control functionality

A list containing descriptions of individual DAQ control functionalities of the GUI follows.

2.2.1 Run control

The most important control functionality of the GUI is the ability to control the state machines (STMs) of the DAQ. This feature is only available when the Master's STM is in a stable or a working state (i.e., the states denoted by blue and purple color in Fig. 1.4, respectively).

2.2.2 TCS prescaler setup

The possibility of altering the value of the TCS prescalers (i.e., *div*) is not excluded from the GUI. This option is offered for all 12 trigger channels. The values can be either set arbitrarily or to various template presets.

2.2.2.1 Monitoring prescaler setup

Similarly to the TCS prescalers, this prescaler affects the ratio of monitoring data to be purposefully discarded. This comprises information regarding the quality of the physics data, such as hit rates or occupancies per given detector frontend channel. These data are visualised using external tools implemented in the ROOT framework[7]. The value of the prescaler can be set for each readout computer independently.

2.2.3 Calibration trigger setup

The calibration trigger is an artificial trigger which generates special events that serve for calorimeter calibration. The rate (how often the events are generated) can be set to one of three given settings. The generation can also be set to be triggered "on spill" or "off spill" separately, meaning the generation starts with the start of a spill and ends with the end of a spill, and vice versa. The generation can also be set to be triggered both "on spill" and "off spill", or on neither, resulting in continuous or disabled event generation, respectively.

2.2.4 DAQ structure type configuration

The information regarding the hardware configuration (i.e., the relationship between the individual source ids and ports) of the new DAQ is stored in a database. It is necessary for the DAQ control system to retrieve this configuration from the database while transitioning from the *ready* state to the *configured* state. The GUI allows the user to choose between individual configuration profiles which are stored in a database[8] (but does not offer the option to alter them – this can be realized through a web interface).

2.2.5 Run configuration

The GUI allows the user to set the number of taken spills after which the run will terminate, the run type (an id stored in the database, helping analysts identify the type of physics data contained), whether the recorded data is to be stored or not, and spill structure. There exist two types of spill structure: SPS and Artificial. In actual data taking, the spill structure is always set to SPS – it contains information concerning when a spill is to occur, received directly from SPS systems. However, as the period of the

[7] A C++-based data analysis and data mining tool developed by CERN [22]

[8] The COMPASS DAQ database, which stores not only these profiles, but also other data related to data acquisiton, such as logbook events and comments

SPS spill cycle is rather long (typically 30 to 40 seconds), this is suboptimal for diagnostics. For this reason, the option to switch to an artificial spill structure with a much shorter period is present.

2.2.6 DAQ hardware link configuration

The ability to toggle individual ports of the new DAQ electronics is undoubtedly the most important functionality pertaining to the DAQ hardware. This feature is most commonly used in order to disable individual ports of the FPGA multiplexers. This resolves problematic situations which occur when an erroneous data stream coming from a source id associated with a malfunctioning detector "clogs up" a large part of the DAQ. Less frequently, it can also be used for detector diagnostics – by toggling individual source ids sequentially, the user can sometimes identify the problematic one.

2.2.7 Hardware error masking

The GUI allows the user to mask errors on individual ports of the new DAQ electronics. The masking status is stored directly in the electronics' registries and is therefore not dependent on the individual GUI instances.

2.2.8 Front-end electronics loading

Using the GUI, it is possible to send a "LOAD" command to the frontend detector electronics. Sending the LOAD command to a source id is an universal method of solving problems which arose on the hardware associated with the source id. Its internal behavior varies depending on the source id (i.e., the LOAD command could reinstall the firmware of a given control unit, re-initialize detector settings to default values etc.). It is important to note that this command is not only used to solve problems, but also serves as a means to configure and generally remotely interact with frontend detector electronics.

The LOAD command is already available through console commands, so its presence in the GUI is merely a "quality of life" improvement. The possibility of filtering source ids by the name of the detector or by integer range, as well as automation of arguments sent with the LOAD command are the features which provide an advantage over the console environment.

2.2.9 DAQ dead-time configuration

The application also provides the possibility of changing the DAQ dead-time. The DAQ dead-time is a collection of trigger settings which constrain the temporal distance between triggers – specifically, it defines constraints for the following three cases:

- Minimum temporal distance between any two triggers
- Minimum temporal distance between the first and last trigger of any sequence of three triggers
- Minimum temporal distance between the first and last trigger of any sequence of ten triggers

These constraints arise from firmware and hardware limitations of the detectors and their electronics. Too short dead-time can lead to problems such as detector noise, cache overflow, or event discarding. As the detector capabilities vary depending on the type, energy and intensity of the particle beam, there exist several pre-defined configurations which allow the detector experts to fine-tune the dead-time settings in order to maximize performance while keeping the detector hardware and firmware stable. Using the GUI, it is possible to switch between the pre-defined profiles, but it is not possible to edit them.

Chapter 3

Pictorial description of the GUI

This chapter provides labeled pictures of the application's graphical user interface, describing the functionality of its individual elements.

3.1 Main window

The Main window implements the following functionality: state machine monitoring, TCS channel monitoring, FPGA register monitoring, run control, prescaler setup, calibration trigger setup, DAQ structure type configuration, and run configuration.

A convenience function of adding comments to the COMPASS logbook and shift manager systems is also provided by this window. The *add_coment* and *shift manager* are independent applications which serve for the purpose of storing information concerning the data-taking process, general condition of the experiment and technical issues. Internally, the GUI only launches these applications [24].

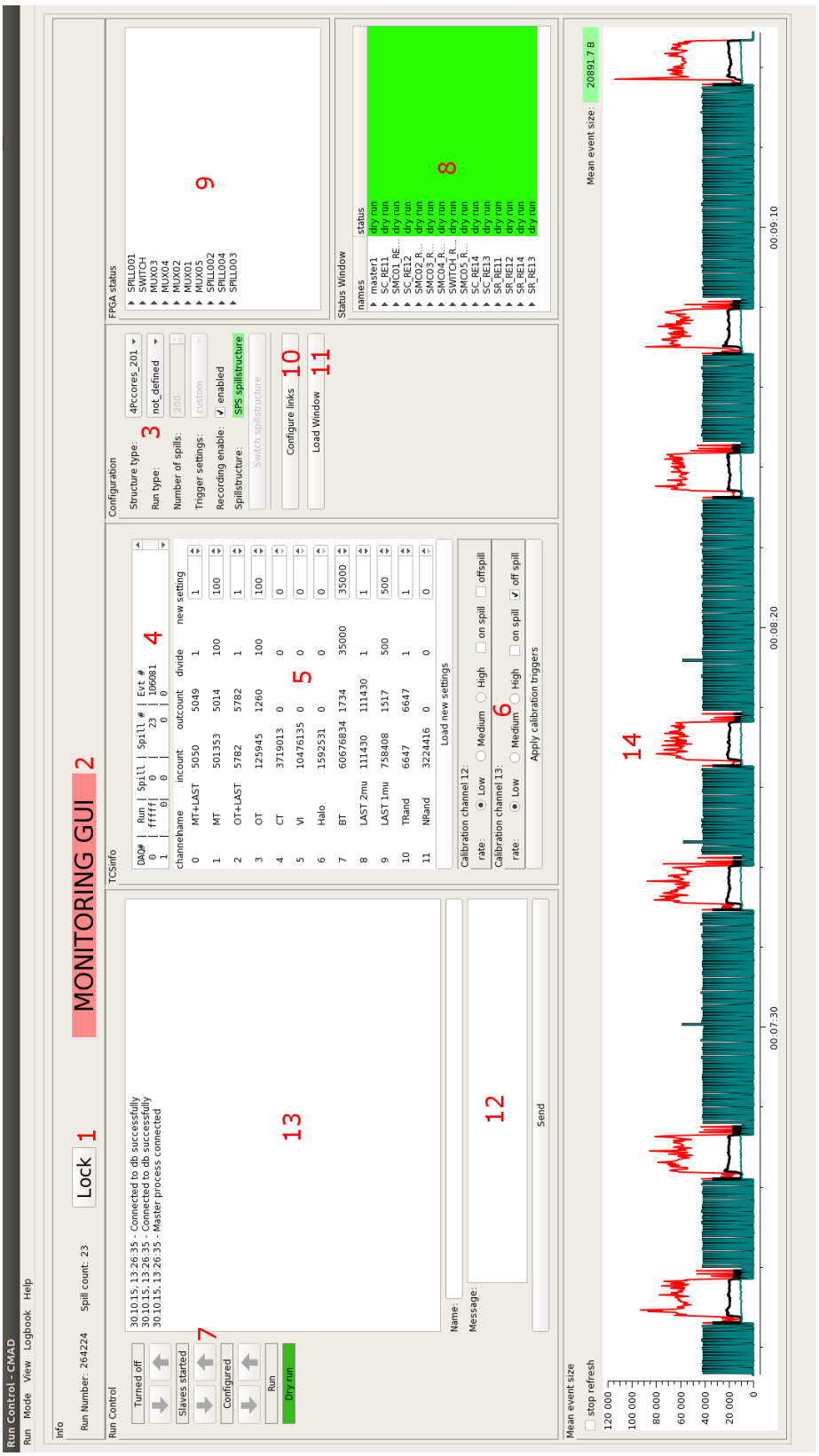


Figure 3.1: The Main window

1. A button which allows the user to lock-in into the control mode
2. A label indicating whether the given GUI instance is in the monitoring mode or the control mode
3. Run configuration area: dropdown menus which allow the user to choose the DAQ structure type, run type, and trigger settings preset for a given run, as well as items which can be used to set the number of spills, spill structure and trigger recording
4. TCS overview
5. The prescaler widget, allowing for monitoring and configuration of individual trigger channels
6. Calibration trigger setup
7. State transition buttons which induce STM state shifts
8. The status widget, displaying states of individual STMs of the DAQ
9. A tree widget allowing for FPGA register monitoring
10. A button which opens the Link status window
11. A button which opens the Load window
12. A textbox allowing for text messages to be sent to other instances of the GUI
13. A message log used to display basic system messages, lock-in messages and messages from users of other instances of the GUI
14. A graph widget showing event size in real time

3.2 Link status window

The Link status window implements the following functionality: DAQ hardware link status monitoring and DAQ hardware link configuration.

Along with the Main window, the Link status window is the most comprehensive monitoring tool in the GUI. The individual Error view buttons light up in red color whenever an error occurs on a given port, attracting the user's attention. If a hardware component is disabled as a result of structure type configuration, the corresponding part of the Link status window is stylized in gray.

The visual structure of the Link status window is split into three distinct parts, each representing a single hardware layer of the new DAQ (described from top to bottom: 8 FPGA multiplexers, 1 FPGA switch, 8 readout computers).

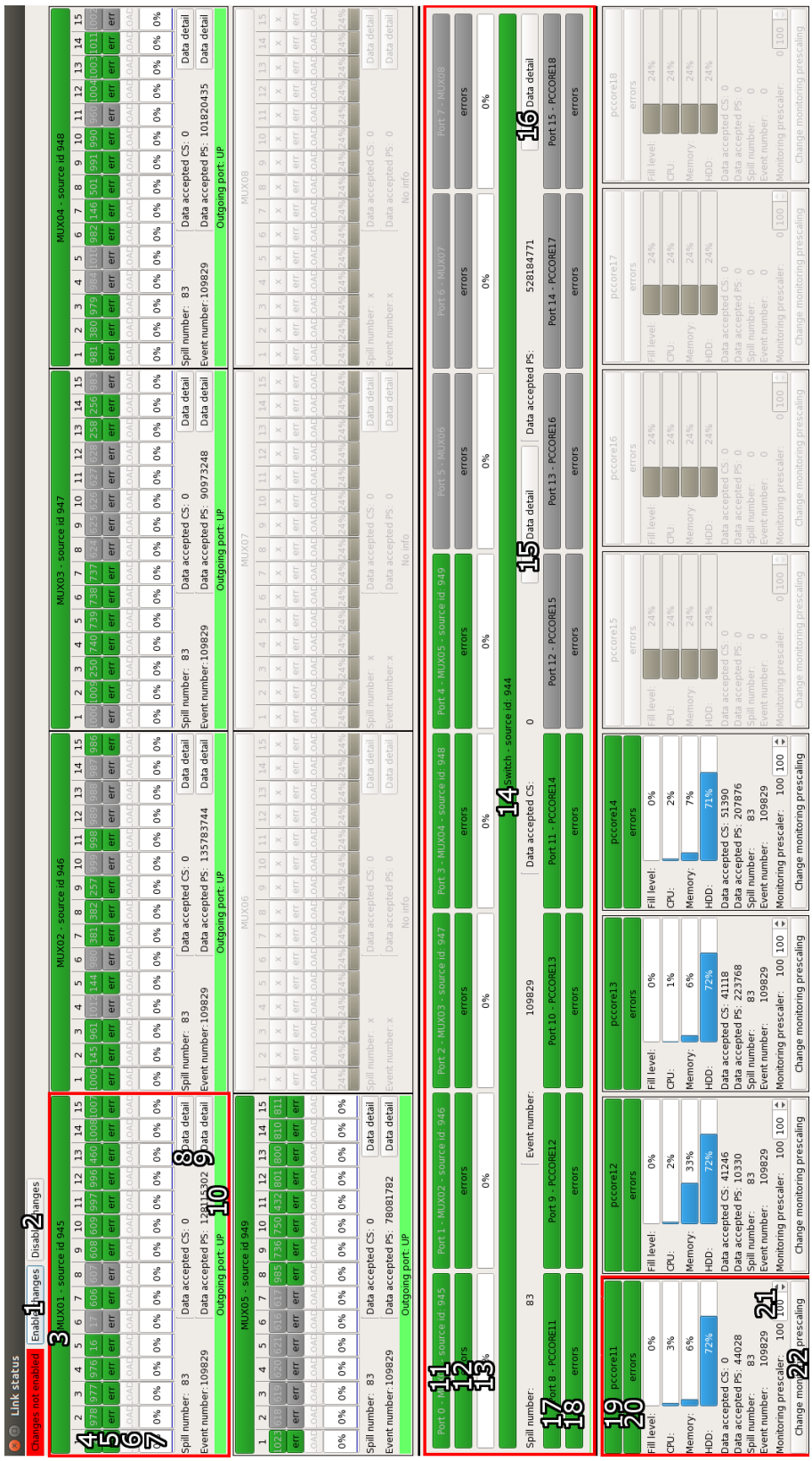


Figure 3.2: The link status window

The areas framed by the red rectangles mark the following elements of individual hardware layers, from top to bottom:

- A Mux frame, each representing a single Multiplexer
- The switch frame, representing the switch
- A PCCORE[1] frame, each representing a single readout computer

The labels denote the following:

1. A button used to enable the control features of the Link status window, so as unwanted changes are not inadvertently made during a run
2. A button used to disable the control features of the Link status window
3. A button used to toggle the outgoing port of the given multiplexer
4. A button used to toggle a single ingoing port of the given multiplexer
5. A button used to open the Error view window associated with a single incoming port of the given multiplexer
6. A convenience button used to send the LOAD command to the source id associated with a single incoming port of the given multiplexer
7. Multiplexer Port memory occupancy
8. A button used to open the Data detail window associated with the multiplexer (current spill)
9. A button used to open the Data detail window associated with the multiplexer (last spill)
10. A label indicating the status of the outgoing port of the given multiplexer
11. A button used to toggle a single ingoing port of the switch
12. A button used to open the Error view window associated with a single ingoing port of the switch
13. Switch port memory occupancy
14. A bar representing the overall state of the switch
15. A button used to open the Data detail window associated with the switch (current spill)
16. A button used to open the Data detail window associated with the switch (last spill)
17. A button used to toggle a single outgoing port of the switch

[1] An acronym used to refer to the readout computers of the DAQ (Personal Computer COMPASS Readout). This acronym is also often used to refer to the spillbuffers, or shortened to "CORE".

18. A button used to open the Error view window associated with a single outgoing port of the switch
19. A button used to toggle the incoming port of the given readout computer
20. A button used to open the Error view window associated with the given readout computer
21. A textbox containing the *div* value of the monitoring prescaler associated with the given readout computer
22. A button used to set the value of the monitoring prescaler given in 21.

3.3 LOAD window

The Load window implements the frontend electronics loading functionality.

The window is split into two distinct parts: the left part displays frames, each representing a single detector (source id group), and listing all source ids associated with it. The following information is displayed for each source id: PCCOFE[2] id, s-link multiplexer source id (cf. [34]), multiplexer source id, related multiplexer port, related switch port. The right part consists of input items which can be used to filter the frames in the left part.

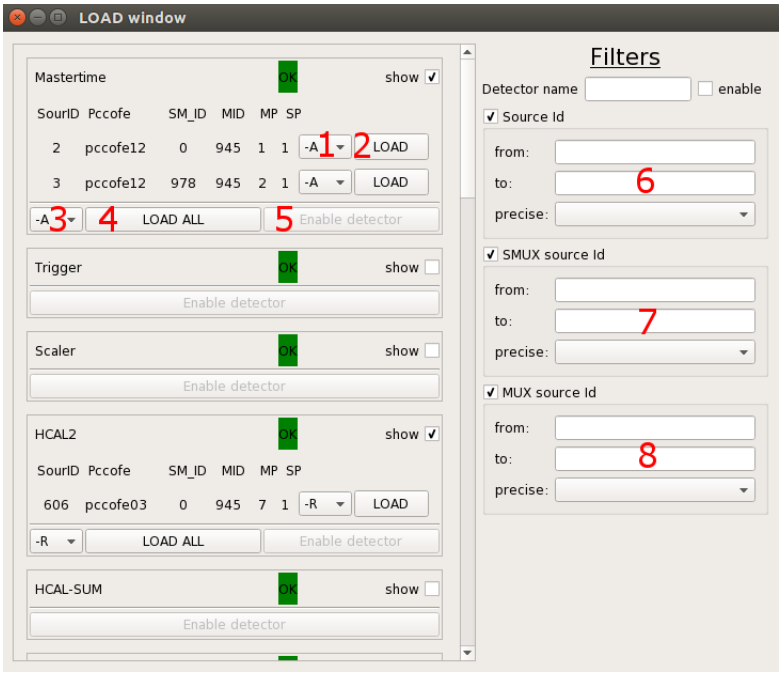


Figure 3.3: The LOAD window

[2]An acronym used to refer to the computers which are used to control COMPASS detector front-end electronics (Personal Computer COMPASS FrontEnd)

The red labels in Figure 3.3 denote the following:

1. A drop-down menu allowing for the selection of the argument to be sent with the LOAD command. This feature is scarcely (if at all) used, as most front-end electronics are re-initialized with one specific argument. The correct argument for re-initialization is pre-selected for user convenience.
2. A button used to send the LOAD command to a single source id
3. Cf. 1.
4. A button used to send the LOAD command to all source ids associated with a given detector
5. A convenience button used to toggle all new DAQ ports associated with a given detector
6. Input items allowing the user to filter the detectors by a single source id of their frontend electronics
7. Input items allowing the user to filter the detectors by the S-link multiplexer source id
8. Input items allowing the user to filter the detectors by the multiplexer source id they are associated with

3.4 Data view

The Data view window implements the accepted data monitoring functionality.

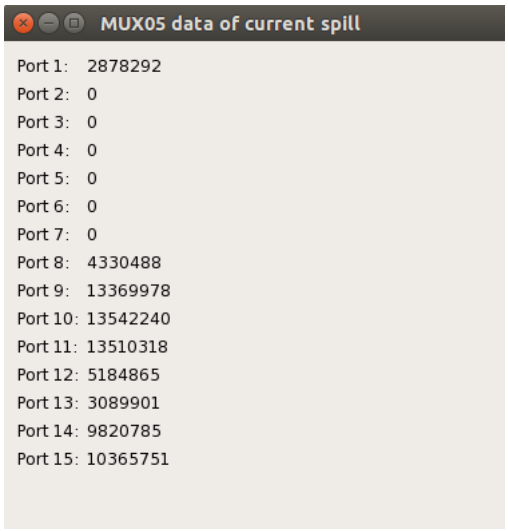


Figure 3.4: The Data view window

3.5 Error view

The Error view window implements the following functionality: error monitoring and hardware error masking.

The Error view window can be accessed through the Link status window. Values of the individual error bits are represented by the background color of the labels.

- Gray – error not occurring
- Red – error occurring
- Yellow – error masked

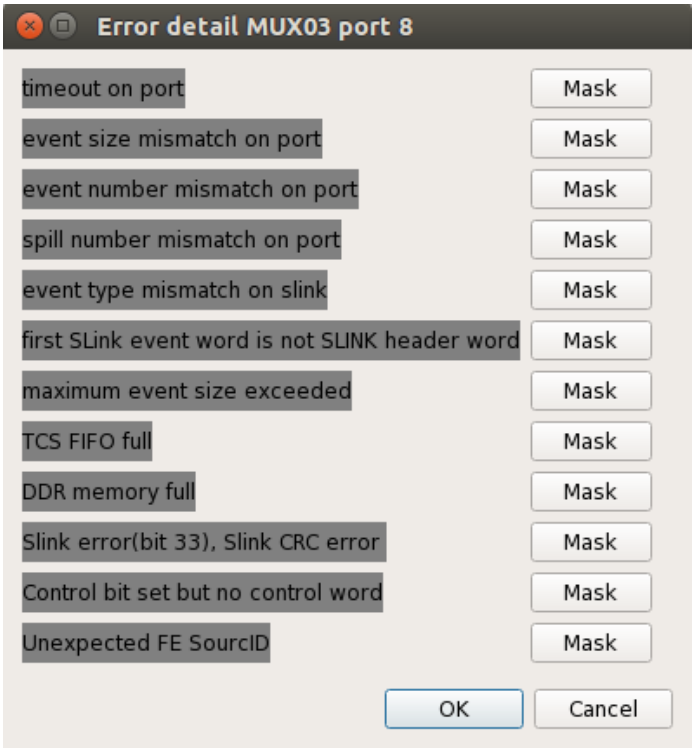


Figure 3.5: The Error view window

Chapter 4

Analysis

This Chapter deals with related work, analysis of the internal implementation of the GUI, and overall requirements for a remote interface for the DAQ control system.

4.1 Related work

The software of the control system of the original DAQ, described in [25], was heavily based on DATE (Data Acquisition and Test Environment). DATE is a software system developed by the ALICE DAQ group designed to perform data-acquisition activities in a distributed multi-processor environment, offering a C++ API. DATE has been designed with emphasis on scalability, fit for large systems counting hundreds of computers, but is suitable for small systems as well, even as small as a single machine with one processor – in such case, that machine will assume the entirety of all roles (LDC[1], GDC[2], run control, and monitoring) and perform their functions [26]. The TCS mentioned in Chapter 1 was deeply intertwined with the old DAQ (the same applies for the new DAQ) and its detailed description can be found in [6].

According to [25], the run control part of DATE was adapted to the needs of COMPASS, incorporating a trigger interface. A graphical user interface which was used for run control was part of this system, offering only very limited monitoring capabilities of the event building network [27]. It was allowed for more than a single instance of this graphical user interface to run simultaneously, but a user switching mechanism was not present. Error handling capabilities were not present either – problems had to be resolved by restarting the control system.

The fundamental ideas behind the new DAQ and its control system described in Chapters 1 and 2 are presented in [4]. A snapshot of the GUI process during early development stages is included, putting emphasis on modularity. A list of DAQ user roles and their respective privileges is also conceived in [4] (Administrator, Expert HW, Expert DAQ, Operator, Visitor), giving rise to the philosophy of GUI lock-in – the Operator role being privileged to using the GUI in control mode and Visitor role in Monitoring

[1] Local data concentrator, equivalent to a readout buffer in COMPASS terminology
[2] Global data collector, equvialent to an event builder in COMPASS terminology

mode (later, these two user roles were merged into a single role privileged to using the GUI in both modes).

The concept is further expanded on in [11] – here, the emphasis shifts from modularity to ergonomy. The reason for the usage of the Qt framework is also given here – originally, the main reason behind its use was to only simplify implementation of the GUI, but its role has changed during early development due to its extensive multi-threading and XML processing support – it was soon included in all processes of the DAQ. The Qt framework was especially useful for implementation of the state machines, which were originally implemented in SMI++[3], but Qt proved to be far more flexible and time efficient. A detailed account of the implementation of the GUI is given in [30].

Results of COMPASS DAQ prototype tests are presented in [29], focusing on readout speed tests rather than user interace testing, the peak rate at which the DAQ can acquire data was measured to be 1.1 GB/s (for comparison, the data flow from all 4 LHC experiments combined is about 25 GB/s).

Results of the pilot run of the system and their analysis are shown in [31], amounting to 195 TB of recorded data during a period of two months. The final, user-tested version of the GUI is shown as well. Future plans for the DAQ are discussed in [32] – in the future, a fully programmable crosspoint switch will be introduced in order to provide a fully customizable DAQ network topology between readout buffers, event builders and readout computers.

A different approach to implementation of run control software can be found at the CMS experiment, whose DAQ runs online software on about 3000 computers used for intelligent buffering and processing of event data [33]. Unlike COMPASS, whose run control is unified for all detectors, the CMS run control structure is organized into eleven different sub-systems, each sub-system corresponding to a detector. The run control system is implemented using web technologies, particularly Java 1.5.0 Web Services. Inter-process communication is realised using the XML data format and the SOAP protocol and the web service interfaces are specified with WSDL using the Apache Axis WS implementation. The service functionalities include, but are not limited to: authentication, account management and job control. The job control service allows for execution and supervision of any software process involved in data taking. These services are then exposed to software referred to as the Function Manager (FM), which is the basic element of the control system, consisting of an input handler, state machine engine, event processor and a resource proxy. Various applications for control and monitoring of the CMS DAQ can be created by writing code against the interfaces provided by the FM, appropriate functionality being provided upon authentication.

4.2 GUI implementation

Being a multi-threaded application, the GUI comprises four threads:

- The main thread, which handles interaction with the user and most of the application logic of the GUI
- The thread associated with the CommObj (Communication object) class, which handles all DIM communication

[3] State Manager Interface, a framework for implementing distributed control systems using (finite) state machines, originally developed at the DELPHI experiment at CERN

- The receiver processor thread, which handles inbound DIALOG communication
- The sender processor thread, which handles outbound DIALOG communication

The GUI is based on the model-view-controller design pattern – the information displayed and actions available are completely reliant on the information messages it receives from the Master. If an instance of the Master is not present when the GUI is launched, an error message will be displayed.

The information messages come in the form of a DIALOG service approximately every 50 milliseconds. Each message contains the following information:

- Master process state
- Slave process names, types, states and monitoring prescaler values
- Current spill number
- The value of the maximum spill number setting
- Event size information
- Id of the control GUI instance (if one exists) – this is used to decide whether the instance is in control mode
- A complete list of FPGA register values – this includes information such as error bit values, error mask values, memory occupancies, node-local spill numbers, node-local event number and event type, source id, firmware version, etc. In total, there are 35 registers per multiplexer, 30 on the switch, 24 per spillbuffer, as well as 15 per each multiplexer port, 14 per each switch port and 9 per each spillbuffer port.

Whenever an information message is received, it is subsequently parsed and the displayed information is updated.

Several of the GUI functionalities are not directly part of the new DAQ – while they were not originally meant to be included in the GUI, it was requested for them to be added in order to make the interface more comprehensive for shift work – over time, many connections to other systems and tools were added, namely:

- TCS – DIM is used for communication with this system, and the Master is not used as the mediator in this case – the GUI communicates with this system directly
- LOAD command – in this case, the GUI directly issues a shell command which launches the LOAD script with appropriate arguments
- DAQ dead-time configuration – several database queries have to be performed in order to change the DAQ dead-time – the GUI performs these queries directly
- Logbook tools – this is another case where the GUI issues a shell command in order to launch an application
- Logbook database – when a new run is started, the meta-information concerning the run is created and stored into the database directly by the GUI

4.2.1 Class description

This section provides a brief overview of the internal class structure of the GUI. A description of roles of a selection of several essential classes is provided below.

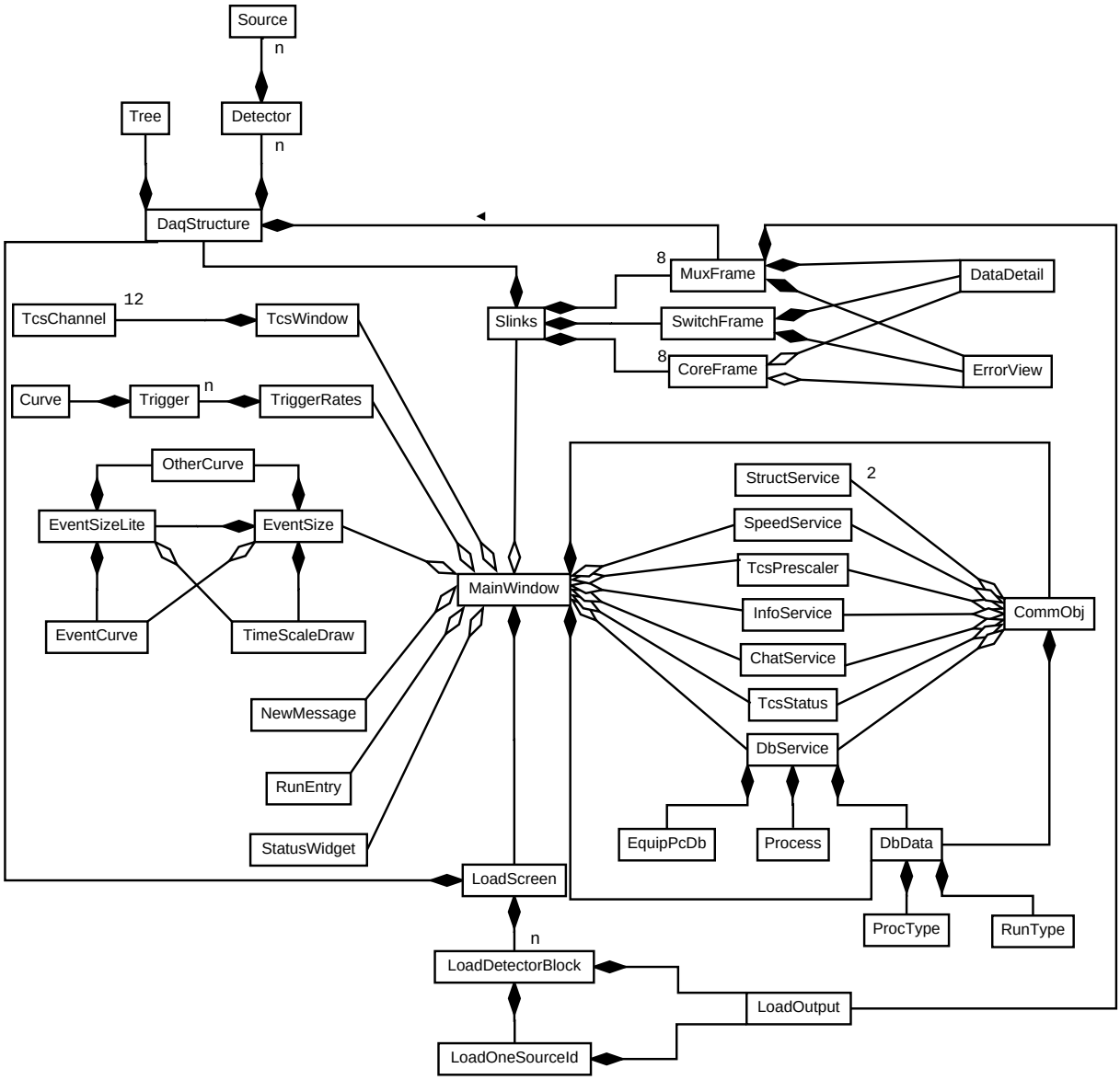


Figure 4.1: UML class diagram of the GUI. Not included: low-level classes used for registry representation and message encapsulation. Cf. section 1.5.2 of [16] for more information.

- **MainWindow** – Implements the logic of the Main Window, i.e. run control, DAQ configuration, TCS control, FPGA monitoring and event size monitoring

- **MuxFrame, SwitchFrame, CoreFrame** – Implement the monitoring and control functionalities related to the FPGA multiplexers, FPGA switch, and readout computers in the Link status window, respectively.
- **Slinks** – Contains instances of the MuxFrame, SwitchFrame and CoreFrame classes and conducts their individual method calls.
- **DataDetail** – Implements functionalities of the Data detail window, i.e., accepted data monitoring on individual ports of the FPGA multiplexers and the switch.
- **ErrorView** – Implements functionalities of the Error view window, i.e., source id error monitoring for a given port and the ability to mask the errors.
- **TcsChannel** – Implements the monitoring functionalities of the trigger system channel monitoring window pertaining to individual channels.
- **TcsWindow** – Contains instances of the TcsChannel class and implements the control functionalities related to the TCS, i.e., setting of the prescaler and calibration triggers.
- **LoadScreen** – Implements the LOAD window – in particular, its filtering functionalities. Instances of the LoadDetectorBlock class are contained within this class.
- **LoadDetectorBlock** – Implements one of the LOAD window functionalities – the ability to load a group of source IDs at once. Instances of the LoadOneSourceId are contained within this class.
- **LoadOneSourceId** – Implements one of the LOAD window functionalities – the ability to load a single source ID.
- **LoadOutput** – Implements the Load output window functionality, i.e., output of the LOAD command displaying.
- **DaqStructure** – Used to create and store an abstract representation of the hardware structure of the DAQ. The representation is created by reading an XML file containing information concerning the hardware structure. This class contains instances of the Detector class.
- **Detector** – Represents structural information concerning a single detector, i.e., which source ids are associated with it. This class contains instances of the Source class.
- **Source** – Represents all relevant information concerning a single source id.
- **TriggerRates, Trigger, Curve, EventSize, EventsizeLite, EventCurve, OtherCurve, TimeScaleDraw** – Implement functionalities related to the event size display widget.

4.3 Motivation for the creation of a remote DAQ control system interface

If a system (i.e., a detector) malfunctions during a run, an on-call expert for that given system is notified by the shift crew. He or she then proceeds to find the cause of the problem – most such analyses can be performed remotely, be it through verbal communication with the shift crew or connecting to the COMPASS local area network using the SSH (Secure Shell) protocol and making use of various software tools.

A vast majority of such problems are false alarms or software problems and can be solved remotely – only a small portion actually requires the expert's presence in the experiment area.

The DAQ is a system fundamental for most of the diagnosis processes – the on-call expert normally needs to utilize it in order to carry out analysis of the problem or to perform testing after having solved it. Currently, the only method to control the DAQ remotely is to connect to the COMPASS local area network using the SSH protocol with X11 forwarding and launch the GUI or to verbally communicate with the shift crew, both options being highly impractical – an instance of the GUI launched in such a manner requires a network bandwidth far larger than what is normally available to most of the experts. A suitable remote access solution for the DAQ would hasten the process of such diagnostic processes or even eliminate the need for the experts to arrive in person.

4.4 CERN Security restrictions

The CERN Computing and Network Infrastructure Security Policy for Controls [35] states that all interactive access to a domain from users external to that domain must pass application gateways and that the application gateways must not be directly visible from outside CERN. As a result, when accessing a domain from outside CERN, the users have to authenticate against the CERN outer perimeter gateways and the gateways to that domain.

The control applications, such as the COMPASS DAQ control system, are then to be run on such gateways or on computers accessible via SSH or RDP protocols from that gateway. In the case of COMPASS network, this means the PCCOGW01[4] and PCCOGW02 gateways.

Figure 4.2 presents a visualization of how one can access computers in the COMPASS domain from outside the CERN GPN (general purpose network). The user first has to authenticate against the lxplus.cern.ch cluster, enabling them to authenticate again against the PCCOGW## computers, from which they can access individual computers in the internal COMPASS network using SSH. All computers in the COMPASS network use the SLC 6 (Scientific Linux CERN [38]) operating system.

[4]Stands for Personal Computer COmpass GateWay

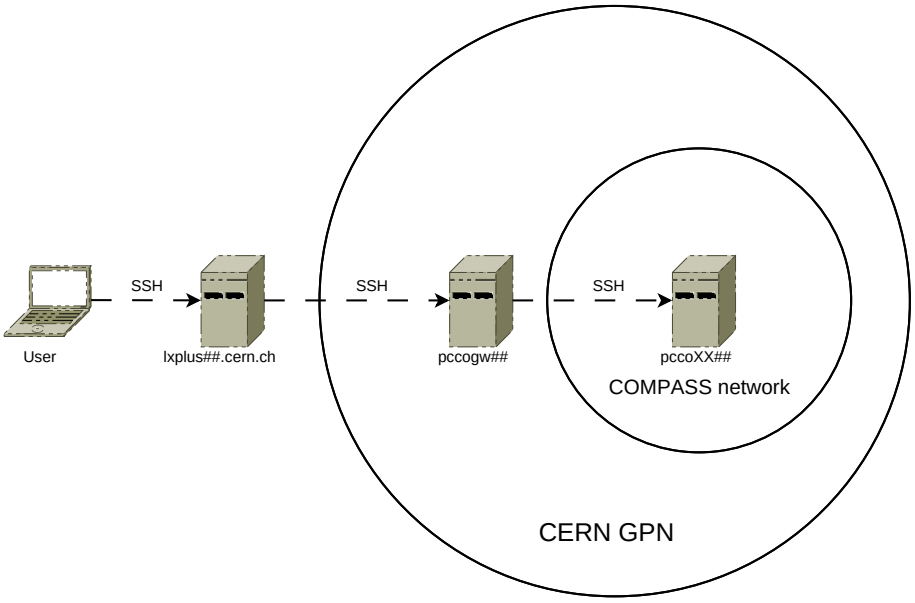


Figure 4.2: A depiction of how computers in the internal COMPASS network can be accessed from outside of the CERN GPN

4.5 Possible remote access solutions

This section discusses several alternatives of approach to DAQ control system remote access and weighs their respective advantages and disadvantages.

4.5.1 Command line interface

One of the possible remote access solutions would be a command line interface (CLI) with functionality similar to the GUI. The user would connect to a control computer using SSH and launch the application on that computer. This would eliminate the need for X11 forwarding and allow for the use in a low-bandwidth, high-latency network environment.

4.5.2 Remote GUI client

The most bandwidth-heavy portion of the current remote control model is the transfer of the graphics using the X11 system. If the GUI was to be run locally on a computer outside of the CERN GPN and forward the network communication with the rest of the DAQ through the CERN GPN and COMPASS gateways using the Transport Layer Security (TLS) protocol [37] and the Socket Secure (SOCKS) protocol [36], the bandwidth requirements would be reduced significantly while preserving the functionality of the GUI.

Internally, such functionality could be implemented using the Qt framework – the *QSslSocket* and the *QNetworkProxy* classes support TLS and SOCKS protocols, respectively [19].

4.5.3 Web interface

Remote access could also be provided by a web application interface hosted on a computer in the COMPASS domain. Such an interface could then be viewed on computers outside the CERN GPN using one of the commercially available web browsers which support the SOCKS protocol. In this scenario, HTTP would be tunelled through a SOCKS proxy, which would eliminate the need to transfer X11 graphics over the network, transferring solely the internal data using HTTP [36, 39].

4.5.4 Analysis conclusion

It would seem as a logical conclusion to choose the Remote GUI client option as the solution – it would be merely an internal modification of an existing interface. However, such a solution would have numerous disadvantages:

As mentioned in section 4.2, the GUI implements several functions which communicate directly with systems present in the COMPASS network environment. Such features would cease functioning if the GUI was to be run outside of this environment.

One of the possible solutions of this problem would be to move the implementation of these features to the Master process, removing them from the GUI. This modification would also result in the need for major network communication adjustments. It would not be possible to test such major changes during a run, resulting in the need to defer the deployment until the next long shutdown of the experiment (currently, there exists no full-fledged testing environment which simulates the entire hardware setup of the DAQ and its interactions with the TCS).

DAQ structure type configuration is another feature implemented in a way that is incompatible with the idea of a remote GUI client. Once a structure type is chosen, the GUI performs a large number of database queries directly, creating an XML file which describes the structure. The implementation of this functionality would also have to be moved to the Master process, putting additional computational strain on it – the loading of DAQ structure is the most expensive action in the GUI in terms of processing power.

Another disadvantage of this approach would be the need for installation and setup on the user's computers. The Qt framework is a pivotal dependency of the GUI and would have to be installed by the user, followed by installation and setup of the remote GUI.

A web application interface would present similar problems – it would necessitate major changes in the DAQ infrastructure, moving the implementation to the Master process, prompting for a re-implementation of the GUI as well. Furthermore, this would also require a certain degree of user-side setup.

On the contrary, the command-line interface approach would preserve the current DAQ software architecture, internal interaction with other systems, as well as user interaction with the DAQ interface – the user would connect to the COMPASS network as normal, but would merely need to use a CLI command

instead of the GUI command to launch such an interface, resulting in a flat learning curve when it comes to launching the interface for the first time, as opposed to the previous solutions. However, the learning curve would possibly not be entirely flat when it comes to using the interface.

Another argument for the command-line interface approach is the possibility of use on mobile devices through an SSH client. Therefore, taking all circumstances into account, it can be concluded that the command-line interface approach is a suitable solution and this work is hereinafter concerned with its creation.

Chapter 5

Solution design and description

This chapter deals with the design of a command-line interface for the COMPASS DAQ control system as well as with description of the solution.

5.1 Basic design questions

The main idea was to design a command-line interface which would mimic the structure of the GUI to a certain degree, while still respecting the limitations and advantages of a text-based interface – emphasis was placed on the following aspects:

- **Dynamicity** – the information displayed and actions available should react to the state of the DAQ in real time without the need for user input
- **Self-descriptiveness** – it should be possible to use the CLI[1] without the need to peruse a manual beforehand (but one should be available)
- **Ease of use** – the interface should feel intuitive and employ features the user may already be familiar with from other environments or tools
- **System consistency** – the CLI should provide exactly the same features the GUI provides and implement them in the same manner

The first questions which had to be addressed were the method of interaction with the user and the way of launching the application. While there exist stateless command-line applications which are launched from an outside shell with given arguments for the purpose of processing a single command (e.g., the Unix grep utility), this approach is unfit for this particular case, given the dynamic nature of the system. A stateful approach, where a command-line application "captures" the shell it is launched from (e.g., the MySql command-line tool) would be far more suitable – therefore, this approach was chosen. The way of launching the CLI should be analogous to the way of launching the GUI – the ideal approach is therefore

[1] "CLI" represents the name of the application, rather than the acronym – a convention analogous to the one introduced earlier concerning the GUI acronym will be used hereinafter

for CLI to be launchable on any compatible computer within the COMPASS network by simply typing *CLI* into its respective terminal.

Another issue was how to deal with the dynamic nature of the system as well as the large quantity of monitoring data being produced while preventing the interface from becoming overly convoluted. The following ground rules were established to address it:

- The information provided by the interface should be divided into a number of distinct categories and should be only shown when the user explicitly requests it
- There should exist methods to monitor the dynamic aspects of the DAQ in real time without the need for any user interaction once launched
- Only one dynamic aspect of the DAQ should be allowed to be monitored at a time per instance of the CLI – in order to monitor more aspects at a time, multiple CLI instances would have to be launched
- While not monitoring a dynamic part of the DAQ, the amount of dynamic output should be minimal (limited only to critical messages)

These design concepts gave rose to the current existing implementation of the CLI, described in the sections below.

5.2 Commands

Once launched and initialized, the application prints out a welcome message, prompting the user to input commands. The most basic command, which is always available, is the *c* command. This command, whose existence is announced in the welcome message, prints the list of available commands at a given time with their respective descriptions. The descriptions are brief, but comprehensive – designed in a way such that a user which is already familiar with the DAQ control system can immediately grasp the functionality of the command.

The set of available commands changes depending on the state of the Master process – whenever the Master process enters a state classified as a transfer state or an error state, a message is printed in the CLI, informing the user and disabling most commands, re-enabling them only once it has returned to a stable or a working state.

Furthermore, the commands take on a tree hierarchy. In other words, not all commands are available immediately (from the main menu). Several of the commands are classified as *menu* commands which enable commands belonging to a given sub-menu, disabling all other commands for clarity. Whenever the user leaves the root node of the menu structure, the *b* command, which is used to return back a level in the menu structure, becomes available.

A large portion of the commands, especially those used for monitoring purposes, display dynamic output. In other words, such a command will "capture" the CLI, barring it from displaying any other output, while displaying information which updates in real time. Such output is always printed to the same visual position when updated, such that it is effortless to read. Once an instance of the CLI is requested

to display dynamic output, it remains in that state until the *return* key is pressed. This also holds true when the Master process enters a state incompatible with producing information associated with the output, or even crashes. In practice, this means CLI instances intended for monitoring do not have to be re-launched or re-configured whenever the Master process enters a different state, is restarted, or crashes.

The CLI assumes the same lock-in philosophy as the GUI. The user can lock into the control mode using the *lock* command, enabling control functionality. This turns any other existing control instance of the CLI or GUI into the monitoring mode. For this purpose, the commands are divided into two classes: monitoring commands and control commands. The control commands are only available when the CLI is in the control mode. It should be noted that a command's being classified as monitoring does not necessarily imply that it performs actual monitoring capabilities – the classification only represents the fact that the command is available in the monitoring mode of the CLI. A listing of commands with their descriptions follows.

```
Welcome to RCCARS Command line UI 1.0
The user guide is at the COMPASS wiki
Bugs/suggestions to: antonin.kveton@cern.ch
Press c at any time for a list of available commands
```

Listing 5.1: The welcome message which is printed when the CLI is launched

```
Executing command "c"
c - Shows all available commands
chat_send - Sends a chat message
lock - Locks into control
poststates - Shows the states of the master and the slaves
quit - Closes the CLI
runconfig_menu - Enters the run configuration menu
runinfo - Shows comprehensive information about the current run
tcs_menu - Enters the TCS menu
```

Listing 5.2: Output produced by the *c* command in a monitoring instance of the CLI during the *Ready* Master process state

Command:	Command type:	Availability in stable and working states:	Availability in transfer and error states:
b	Monitoring	All	All
c	Monitoring	All	All
calibration_info	Monitoring	All	None
chat_send	Monitoring	All	None
configureslaves	Control	Ready	None
coreinfo	Monitoring	Configured, Dry run	None
daqdeadtime_change	Control	Waiting, Ready, Configured	None
datadetail_m	Monitoring	Configured, Dry run	None
datadetail_s	Monitoring	Configured, Dry run	None
errormonitoring	Monitoring	Configured, Dry run	None
fpgainfo	Monitoring	Configured, Dry run	None
LOAD	Control	All	None
loadprescalers	Control	All	None
lock	Monitoring	All	All
maskerror_m	Monitoring	Configured, Dry run	None
maskerror_r	Monitoring	Configured, Dry run	None
maskerror_s	Monitoring	Configured, Dry run	None
mux_menu	Monitoring	Configured, Dry run	None
muxinfo	Monitoring	Configured, Dry run	None
numberofspills_set	Control	Waiting, Ready, Configured	None
numberofspills_show	Monitoring	Waiting, Ready, Configured	None
pccore_menu	Monitoring	Configured, Dry run	None
pdatadetail_m	Monitoring	Configured, Dry run	None
pdatadetail_s	Monitoring	Configured, Dry run	None
portinfo_m	Monitoring	Configured, Dry run	None
portinfo_s	Monitoring	Configured, Dry run	None
poststates	Monitoring	All	All
quit	Monitoring	All	All
recording_set	Control	Waiting, Ready, Configured	None
runconfig_menu	Monitoring	Waiting, Ready, Configured	None
runconfig_show	Monitoring	Waiting, Ready, Configured	None
runinfo	Monitoring	All	None
runnumber_show	Monitoring	Waiting, Ready, Configured	None
runtype_set	Control	Waiting, Ready, Configured	None
calibration_set	Control	All	None
channel_set	Control	All	None
setmonprescaler_r	Control	Configured, Dry run	None
setport_m	Control	Configured, Dry run	None
setport_s	Control	Configured, Dry run	None
slinks_menu	Monitoring	Configured, Dry run	None
spillstructure_show	Monitoring	Waiting, Ready, Configured	None
startrun	Control	Configured	None
startslaves	Control	Waiting	None
stoprun	Control	Dry run	None
stopslaves	Control	Ready	None
structuretype_set	Control	Waiting	None
switch_menu	Monitoring	Configured, Dry run	None
switchinfo	Monitoring	Configured, Dry run	None
spillstructure_switch	Control	Waiting, Ready, Configured	None
tcs_menu	Monitoring	All	None
triggervalues_show	Monitoring	All	None
unconfigureslaves	Control	Configured	None
viewmux	Monitoring	Configured, Dry run	None
viewport_m	Monitoring	Configured, Dry run	None
viewport_r	Monitoring	Configured, Dry run	None
viewport_s	Monitoring	Configured, Dry run	None

Table 5.1: A complete list of the 56 CLI commands, their types, and their availabilities based on the Master process state

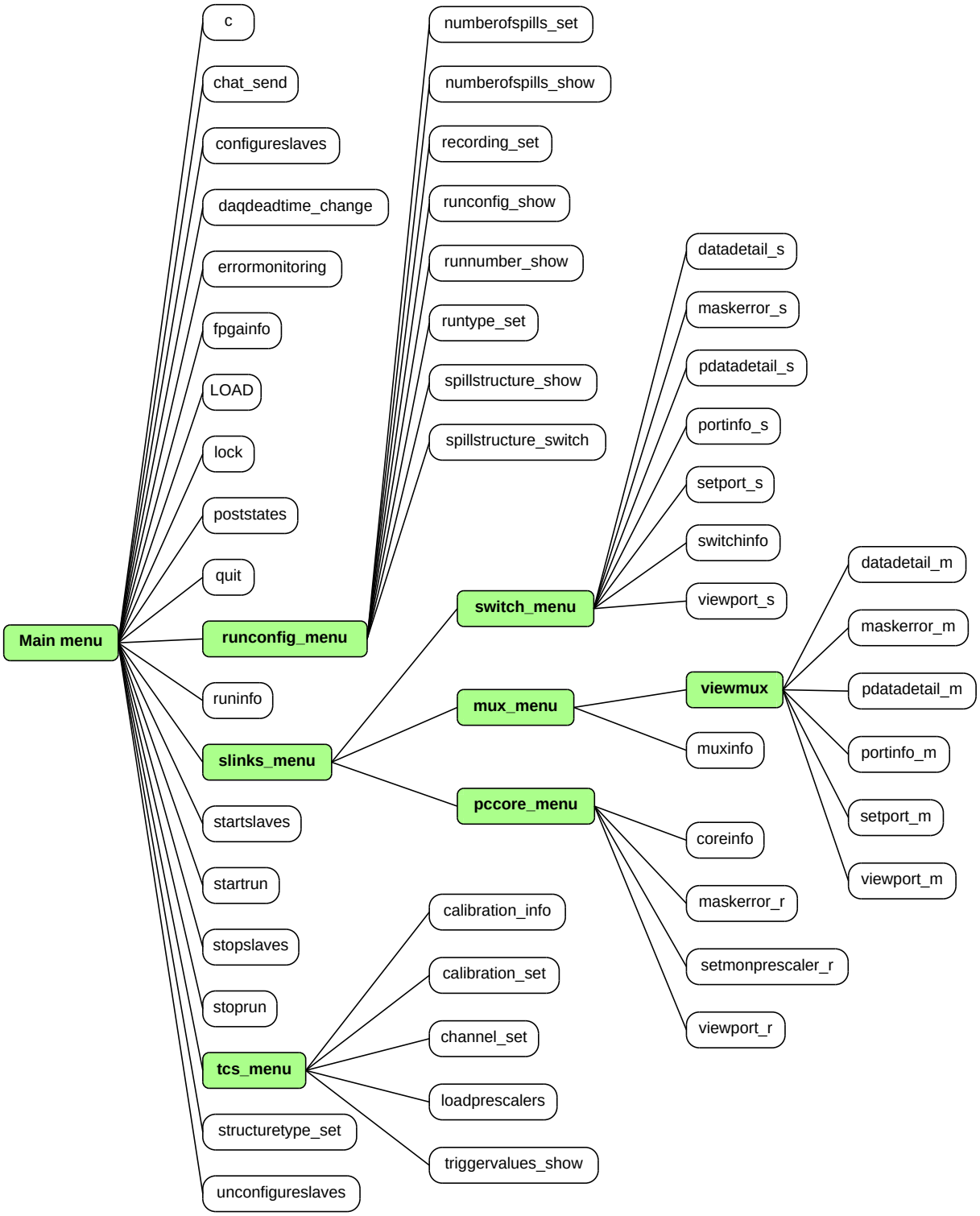


Figure 5.1: The menu structure of the CLI

5.2.1 Main menu

This section provides a description of commands available immediately upon launching the CLI, provided the Master process is in the appropriate state for that command to become available.

5.2.1.1 Chat_send

This command can be utilized to send a custom chat message to all other instances of the CLI/GUI.

5.2.1.2 Lock

As mentioned previously, once the *lock* command is used, the CLI enters the control mode. The user is prompted for their name and an optional message which is to be sent to all other instances of the CLI/GUI.

5.2.1.3 Quit

This command terminates the CLI.

5.2.1.4 Poststates

The *poststates* command is a command with dynamic output which is used to view the states of the individual processes of the DAQ. It displays information identical to that displayed in the status window in the GUI.

```
Master1: dry run
SC_RE11: dry run
SMC01_RE11: dry run
SC_RE12: dry run
SMC02_RE12: dry run
SMC03_RE13: dry run
SMC04_RE14: dry run
SWITCH_RE11: dry run
SMC05_RE15: dry run
SC_RE14: dry run
SC_RE13: dry run
SMC06_RE15: dry run
SR_RE11: dry run
SR_RE12: dry run
SR_RE14: dry run
SR_RE13: dry run

Dynamic output: to end, press ENTER
```

Listing 5.3: Output produced by the *poststates* command – as with all dynamic output commands, the values update in real time without changing location

5.2.1.5 Errormonitoring

This command was created to serve as an easy way to monitor hardware errors of the DAQ. In the GUI, the link status window serves for this purpose – whenever an error occurs, a given error window or port lights up in red color. Such design would be very difficult to visually comprehend when implemented in text form, and therefore a different approach was chosen for the CLI.

The dynamic output of this command functions as follows: if there are hardware errors occurring in the DAQ, details concerning those errors are printed. Otherwise, a "No errors occurring on slinks" message is printed.

This approach allows the user to see the details of the error immediately, whereas in the GUI, the user normally has to perform several other actions in order to find out the details concerning the error. Namely, the actions of mouseovering the given port in order to determine the detector associated with it and opening the Error detail window in order to determine the type of the error occurring are the most time-consuming actions and it is not commonplace that the error disappears before the user performs these actions, forcing them to use the Message Browser application to retroactively view the errors.

The output of this command displays each error in a single line of text, decreasing the likelihood of having to resort to use of the Message Browser. The information displayed for each error occurring is as follows:

- Hardware component name
- Port index
- Source id
- Name of the detector associated with the source id, if applicable
- Error type

Port communication is also monitored by this command – if a port, the communication with which has been lost, is present, the following information is shown:

- Hardware component name
- Port index
- Source id
- Name of the detector associated with the source id, if applicable

```
Slinks error overview:

Error on MUX04, port 10 (source id 983 - SMUX-SciFI-D-6), Timeout on port
MUX04, port 4 enabled but not up (source id 635 - ECAL0_635)

Dynamic output: to end, press ENTER
```

Listing 5.4: Output produced by the *errormonitoring* command

5.2.1.6 Runinfo

The *runinfo* command is a command which dynamically prints meta-information concerning the current data-taking run. If no run is active when this command is used, information concerning the last run is posted instead. The following information is printed:

- Current run number
- Current spill count
- Maximum spill count setting
- Burst (whether a spill is occurring)
- Current spill structure setting

```
Run number: 276989
Spillcount: 293/2000
Burst: 1
Spill structure: Artificial

Dynamic output: to end, press ENTER
```

Listing 5.5: Output produced by the *runinfo* command

5.2.1.7 Fpgainfo

Using dynamic output, this command allows for direct FPGA register value monitoring in real time. First, the list of FPGAs is printed, and the user is prompted to select one. Next, the list of register groups is printed for that given FPGA (one group for meta-information concerning the FPGA itself and one group for each of its ports). After the user selects which register group they wish to view, the values concerning that group are displayed in real time.

5.2.1.8 Daqdeadtime_change

The DAQ deadtime configuration file can be changed using this command.

5.2.1.9 LOAD

This command serves as the means of front-end electronics loading. Internally, this command calls the console LOAD script which can also be used outside of the CLI, and therefore, the syntax is identical.

5.2.1.10 Structuretype_set

This command allows for DAQ structure type configuration – the user can choose between various structure types retrieved from the database before transitioning to the *configured* state.

```
FPGA fw version 2015072410
Source ID 0x385|901
Global reset 0
Event number 0x3763|14179
Current spill number 0xb6|182
bx0 0
Event type of current event 1b
Time of current event 401a97
LEvent number 0x371a|14106
Last spill number 0xb5|181
Lbx0 0
LEvent type of current event 1f
LTime of current event 9a49e2
Header enable 1
Reserve 0
Slink UP 1
Time enable 1
bx000 reserved 0
TCS receiver enabled 1
ECC mode 0
TCS receiver ID 0
TCS errors 8
TCS DAQ mask 1
Number of connected RE 3

Dynamic output: to end, press ENTER
```

Listing 5.6: Output produced by the *fpgainfo* command, displaying the register values of a spillbuffer

5.2.1.11 Startrun

This command can be used to start a run (i.e. perform a Master state shift from the configured state into the dry run state). As mentioned previously, there are several parameters that need to be set before starting a run (number of spills, recording setting, run type, TCS prescaler template). If these parameters are not set, the default values are used – in the case of the number of spills and recording values, this is desirable behavior, as the maximum number of spills seldom changes, and the data should be almost always recorded (with the exception of SPS technical stops etc.). However, the run type should always be set manually before a run is started, as this is a parameter that tends to change often.

If the run type is of the default value (*not_defined*), this command will print a message conveying this information to the user, prompting them about their desire to proceed and start the run. If the recording is disabled, the user will be warned in a similar way. Once a run is started, all of the important parameters are also printed.

5.2.1.12 Other run control commands

Startrun is not the only command which allows to induce a Master state shift. The rest is listed below:

- *startslaves*: *Waiting →Ready*
- *stopslaves*: *Ready →Waiting*
- *configureslaves*: *Ready →Configured*
- *unconfigureslaves*: *Configured →Ready*
- *stoprun*: *Dry run →Configured*

5.2.2 TCS menu

This section describes all commands related to the trigger control system, which can be accessed by using the *tcs_menu* command.

5.2.2.1 Triggervalues_show

This is yet another command which uses dynamic output. It allows for TCS channel monitoring, displaying the *In*, *Out* and *Div* values for each channel. If the value of *In* is equal to 0 and the value of *Div* is not equal to 0 for the same channel, the line describing that channel lights up in red, indicating a trigger problem.

```
Ch:     Name:       In:  Out:  Div:
-----------------------------------
  0  TigerDT0      5050  5049     1
-----------------------------------
  1        MT    501353  5014   100
-----------------------------------
  2        LT      1453  1453     1
-----------------------------------
  3        OT     36371 36371     1
-----------------------------------
  4        CT   3719013     0     0
-----------------------------------
  5        VI  10476135     0     0
-----------------------------------
  6      Halo   1592531     0     0
-----------------------------------
  7        BT  60676834  1734 35000
-----------------------------------
  8     ECAL0      2454     0     0
-----------------------------------
  9      LAST    758408  1517   500
-----------------------------------
 10     TRand    422965     0     0
-----------------------------------
 11     NRand    991525     0     0
-----------------------------------

Dynamic output: to end, press ENTER
```

Listing 5.7: Output produced by the *triggervalues_show* command

5.2.2.2 Calibration_info

This command displays information pertaining to the calorimeter calibration channels. The rate, *onspill* and *offspill* settings are displayed for each channel.

5.2.2.3 Calibration_set

The *calibration_set* command serves as a means of calorimeter calibration trigger setup. The trigger rates can be set to three different values for each channel. The setting concerning *onspill* and *offspill* event generation is also set using this command.

5.2.2.4 Channel_set

This command allows the user to set the *div* value of a single TCS channel prescaler.

5.2.2.5 Loadprescalers

Using this command, it is possible to load one of the TCS channel prescaler templates from a database. This sets the *div* values of all the TCS channels to those defined by the template.

5.2.3 Run configuration menu

The description of commands included in the *runconfig_menu* can be found below.

5.2.3.1 Spillstructure_switch

Using this command, one can change the spill structure from SPS to artificial and vice versa.

5.2.3.2 Recording_set

This command allows the user to choose whether data taken during a run will be stored. If this command is not used before a run is started, the value will default to *true*.

5.2.3.3 Numberofspills_set

The *numberofspills_set* command can be used to change the maximum number of spills per run. If this command is not used before a run is started, the value will default to a value received from the Master process (the value from the last run, usually 200).

5.2.3.4 Runtype_set

This command can be utilized in order to set the run type before starting a run. If not specified before a run is started, the run type will default to the *not_defined* run type.

5.2.3.5 Spillstructure_show, runnumber_show, numberofspills_show

These commands print the spill structure being used, current run number, or the current maximum spill count setting, resepectively.

5.2.3.6 Runconfig_show

The *runconfig_show* command can be used to confirm the run configuration before starting a run. In other words, values of all settings which need to be set in order to start a run are printed:

- Run type
- Maximum spill count
- Whether the data is to be recorded or not
- Spill structure

5.2.4 S-Link menu and its sub-menus

This section provides a description of commands enabled after using the *slinks_menu* command, or any of its related sub-menu commands.

5.2.4.1 Muxinfo, switchinfo, coreinfo

Using dynamic output, these commands print the following meta-information concerning each of the connected multiplexers, the switch, or the readout computers, respectively:

- Current node-local spill number
- Current node-local event number
- Total data accepted during current spill
- Total data accepted during previous spill
- Port status

In the case of *coreinfo*, the monitoring prescaler setting is also shown.

5.2.4.2 Viewmux

The *viewmux* command is a de-facto menu command – after the user selects a multiplexer to view, a set of commands related to that given multiplexer is enabled (the complete list of enabled commands is shown in fig 5.1).

5.2.4.3 Datadetail_s, Pdatadetail_s

These commands use dynamic output in order to display data accepted during current and previous spill on each of the ingoing ports of the switch, respectively.

5.2.4.4 Portinfo_s

The *portinfo_s* command displays the following information concerning each of the switch ports:

- Port index
- Whether the port is ingoing or outgoing
- Whether the port is enabled or disabled
- Source id (for ingoing ports only)

5.2.4.5 Viewport_m, viewport_s, viewport_r

These commands allow for dynamic monitoring of errors on a given multiplexer, switch, or readout computer port, respectively. When one or more errors occur on the port, the error types are posted. This is a built-in redundancy with the *errormonitoring* command, allowing the user to monitor a faulty port elaborately.

5.2.4.6 Portinfo_m

The *portinfo_m* command displays information concerning the ports of a given multiplexer and its related source ids. Specifically, the following information is posted for each port:

- Port index
- Source id
- Associated source ids[2] and related detector names

[2]If a given source id corresponds to a multiplexer, the associated source ids are those that correspond to electronics connected to that multiplexer

- Whether the port is enabled or disabled

```
VIEWING MUX01, srcid: 945

Port 1 srcid: 2 (Mastertime_1)
Associated srcids:
2 - Mastertime

Port 2 srcid: 978 (SMUX-Mastertime/Trigger)
Associated srcids:
3 - Mastertime 65 - Trigger 66 - Trigger 67 - Trigger

Port 3 srcid: 977 (SMUX-Trigger1)
Associated srcids:
68 - Trigger 69 - Trigger 70 - Trigger

Port 4 srcid: 976 (SMUX-Trigger2)
Associated srcids:
71 - Trigger 72 - Trigger 73 - Trigger 74 - Trigger

Port 5 srcid: 16 (Scaler_1)
Associated srcids:
16 - Scaler

Port 6 srcid: 17 (Scaler_2)
Associated srcids:
17 - Scaler

Port 7 disabled
Port 8 disabled
Port 9 disabled
Port 10 disabled
Port 11 disabled
Port 12 disabled
Port 13 disabled
Port 14 disabled
Port 15 disabled
```

Listing 5.8: Output produced by the *portinfo_m* command

5.2.4.7 Datadetail_m, Pdatadetail_m

These commands use dynamic output in order to display data accepted during current or previous spill on each of the ingoing ports of the given multiplexer, respectively.

5.2.4.8 Maskerror_m, maskerror_s

These commands can be used to perform hardware masking of multiplexer, switch or spillbuffer errors, respectively – the user selects the port and error type to be masked. This mask is then applied into the FPGA registry.

5.2.4.9 Setport_m, setport_s

These commands allow the user to toggle a single multiplexer or switch port, respectively.

5.2.4.10 Setmonprescaler_r

This command serves as the means of setting the value of the monitoring prescaler on a single readout computer.

5.3 Command argument input

While the CLI is designed to have a mostly flat learning curve, there should also be tools which allow advanced users to speed up their work at the expense of the curve becoming steeper. This is why the CLI allows for two approaches when using commands which require arguments.

The first approach is to simply use commands without specifying arguments at all. If any arguments are required, the user will be prompted for them.

The other approach is to specify the parameters after having typed the command. While the disadvantage of this method is the user having to learn the parameters used, this manner of using commands can be faster and more comfortable for the user. The user themself can choose which approach suits them the most. If the arguments are input incorrectly or insufficiently using this method, the user will be prompted for them using the first method. The following section describes how the non-promptive approach is structured.

```
>viewmux -m this_is_incorrect
Executing command "viewmux"
Invalid input
Which mux?
>
```

Listing 5.9: An example of incorrect direct argument input and the following prompt

5.3.1 Non-promptive approach

The CLI uses "switches" to pass arguments directly. A switch is a pair of characters, consisting of the hyphen character and one or more alphabetic characters (i.e. *-e*). This method is advantageous in that the order of the parameters is arbitrary. There exist four different classes of switches:

5.3.1.1 Required parametric switches

These switches comprise parameters which are necessary for a command to run and have a numeric or string input value associated with them.

Input format: *<command> -<switch> <value>*

Example: *viewmux -m 5*

5.3.1.2 Optional parametric switches

These switches are used to represent parameters which have a numeric or string input value associated with them, but are not necessary for a command to run.

Input format: *<command> -<switch> <value>*

Example: *lock -m Hello*

Here, *-m* is used to send an optional message to other users when locking into control

5.3.1.3 Required non-parametric switches

These switches represent arguments which are necessary for the command to run, but do not have a numeric input value associated with them. The options are grouped into sets whose members are mutually exclusive. It is necessary to provide at least one member of each set for the command to run successfully – in other words, this switch class forces the user to make a choice between several options. A typical example of such a set is {*enable*, *disable*}. While such a set could be represented by the values 0 and 1 using the required parametric switch approach, this method converts the values into a human-readable format.

Input format: *<command> -<switch>*

Example: *setport_m -e*

-e is used to enable a port. For this command to function, the required parametric switch *-p <port index>* has to be included as well.

5.3.1.4 Optional non-parametric switches

This switch type does not have a numeric input value associated with it and is not needed in order for the command to run. Similarly to the previously listed switch class, these parameters are also grouped into sets whose members are mutually exclusive. However, as opposed to required non-parametric switches, sets of size equal to 1 can also be present.

In the case of required non-parametric switches, sets of size lesser than 2 are not logical, as the idea of such switches is to force the user to make a choose a single member of the set. In optional non-parametric switches however, the optionality can be represented by either having selected a member from the set, or no member at all, which is consistent with the existence of a set of size equal to 1.

Input format: *<command> -<switch>*

Example: *calibration_set -on*

Here, *-on* represents "onspill" and is not mutually exclusive with *-off*

5.4 User experience features

In order to enhance user experience, several features which improve ease of use were added:

- **Tab completion** – when in the midst of typing a command, the user can press the tab key to complete it. If the string used is ambiguous (i.e., more than one command begins with such a sub-string), a list of such commands is printed. This feature is almost a necessary one in the case of this application, as the self-descriptive names of commands are often substantially long.
- **Command history** – the CLI stores command history during individual sessions. The *up* and *down* keys can be used to cycle through recently used commands.
- **Expert mode** – this is a feature which allows experienced users to speed up their usage of the CLI. When the CLI is launched with the *e* argument, this mode will be triggered, which results in elimination of the menu hierarchy. Therefore, all commands related to the active Master process state are available immediately. This feature is the reason for the need to differentiate the names of commands which serve the same purpose, but are situated in different parts of the menu hierarchy (e.g., setport_m, setport_s).

Chapter 6

Implementation

Counting over 10 thousand single lines of code, the application is based on C++11 and extensively utilizes the Qt framework, especially for inter-thread communication using the signal-slot system. Integration with the rest of the DAQ is implemented using the DIALOG network communication library. The DIM network communication system is also utilized in the application, particularly for communication with the TCS.

In total, the CLI utilizes 5 threads: the main thread, which handles most of the application logic, two threads which handle inbound and outbound network DIALOG communication, one thread which handles DIM network communication and lastly, a thread dedicated to interaction with the user.

In terms of class structure, the *ui_object* class is central to the application, as it aggregates the class which handles network communication with those that are used to represent various subsystems of the DAQ as well as those which handle interaction with the user.

The *communication_object* class is the class responsible for network communication. It mantains signal-slot connections to the *receiver_processor_thread*, *sender_processor_thread* classes and aggregates various TCS DIM service classes. Whenever a message is received from the Master process or from the TCS, an appropriate signal is emitted by the *communication_object* class, being connected to a slot in the *ui_object* class, which then parses the message and updates the representations in the various classes it manages.

The *input_object* class implements most of the interaction with the user – it handles interpretation of user input and implements the user experience features from the previous chapter as well as general mechanisms used to display dynamic output in the terminal.

Every command is implemented as a separate class derived from the *base_command* class. The *ui_object* manages all such classes and based on messages it receives from the Master process, it decides which commands are available to the user at a given time.

A detailed description of a selection of classes follows.

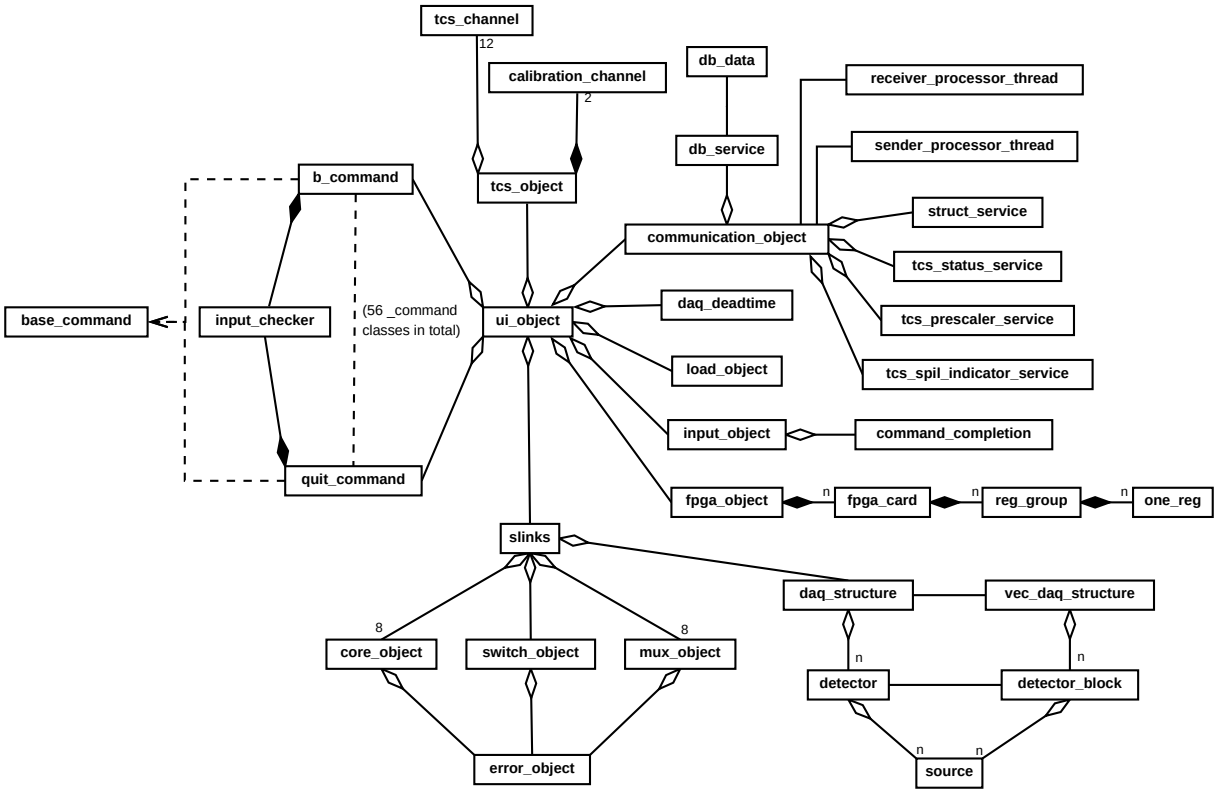


Figure 6.1: A simplified UML class diagram of the application

6.1 Base_command class

The *base_command* class serves as the base class for all commands. The derived classes override its virtual methods:

- **Initialize_inputchecker** – this method is used to setup the instance of the inputchecker class, i.e., to specify which parameters and their respective values are valid for that given command.
- **Ask_for_args** – if the arguments have not been received directly, this method is called to prompt the user for them. The return value is 1 or 0 depending on whether correct arguments have been received, 1 indicating success. If a command has no arguments associated with it and does not override this method, then the return value will default to 1.
- **Process_args** – this method is called to process arguments received directly, which includes setting necessary flags, values, etc.
- **Perform** – this method is used to implement the logic of the command.
- **Compose_dynamic_output** – if the command is one that produces dynamic output, then the construction of its contents is defined in this method.

Concerning non-virtual methods, *parse_args* is a notable method of this class which is used to parse and convert direct argument input into a valid inner argument representation without the need to be overridden by every *_command*[1] class separately.

Once the *input_object* class emits the *command_entered_signal*, the connected *process_input_slot* of the *ui_object* class is executed in the main thread, parsing the input so that it is split into the command name and arguments. Using the command name as a key, the instance of the related *_command* class is retrieved from a container and the *exec* method is called. Its source code can be seen in listing 6.1.

```
QVariant base_command::exec()
{
    if(arg_n > 0)
    {
        if(input_checker->is_absolute_leeway())
        {
            process_args();
            return perform();
        }
        if(input_checker->check_args(arguments))
        {
            parse_args();
            process_args();
            return perform();
        }
        else
        {
            if(ask_for_args() == 1)
            {
                return perform();
            }
            else
            {
                return QVariant(0);
            }
        }
    }
    else
    {
        if(ask_for_args() == 1)
        {
           return perform();
        }
        else
        {
            return QVariant(0);
        }
    }
}
```

Listing 6.1: The *exec* method of the *base_command* class

[1]This expression will be used to refer to any class derived from *base_command*

6.2 Input_object class

In order to provide the desired functionality, a separate thread had to be dedicated to execute the logic provided within this class. Specifically, the tab-command completion and command history functions are impossible to implement in an environment which only scans user input upon the press of the *return* key. The thread is used to observe the standard input continuously – without the need for the *return* key to be pressed. This allows for multiple keys to serve as "special keys", which induce an immediate action, such as a command being completed upon a press of the tab key.

Internally, this behavior is implemented using the POSIX functions included in the *termios* header (cf. [41]). In order to achieve the desired behavior, canonical[2] mode has to be disabled as well as echo mode. An infinite loop is then used to monitor the standard input buffer. This approach necessitates building the terminal environment from the ground up – the basic input features, such as functionality of the *delete*, *backspace*, and other keys had to be re-implemented.

For this reason, a multitude of elementary terminal manipulation methods are present within this class:

- **Get_term_row** – returns the size of the terminal in rows
- **Get_term_col** – returns the size of the terminal in columns
- **Move_cursor** – moves the cursor visual to the specified row and column. This, and other various cursor and text manipulation (such as coloring) is handled using ANSI escape sequences (cf. [42]).
- **Clear_command** – mimics the visual of the return key functionality in canonical mode, i.e., when the return key is pressed, the visual input buffer is cleared, and the cursor visual is moved to the beginning of the last terminal row

This way of managing the terminal is however not always the one used. When the user is being prompted for command arguments, the terminal returns to canonical mode, the sole reason being ease of implementation. The aforementioned input-observing loop is situated In the *get_input_slot* method, which is called once when the thread starts, and every time a command has finished executing. Once a command has been entered, and the return key pressed, the thread of execution leaves this method, enabling canonical mode and prompting the main thread, namely *ui_object*, to take over user input and output. The implementation of this behavior is shown in Figure 6.2.

[2] An input processing mode in which no input is available to the program until it has been terminated by newline, EOF, or EOL characters [40]

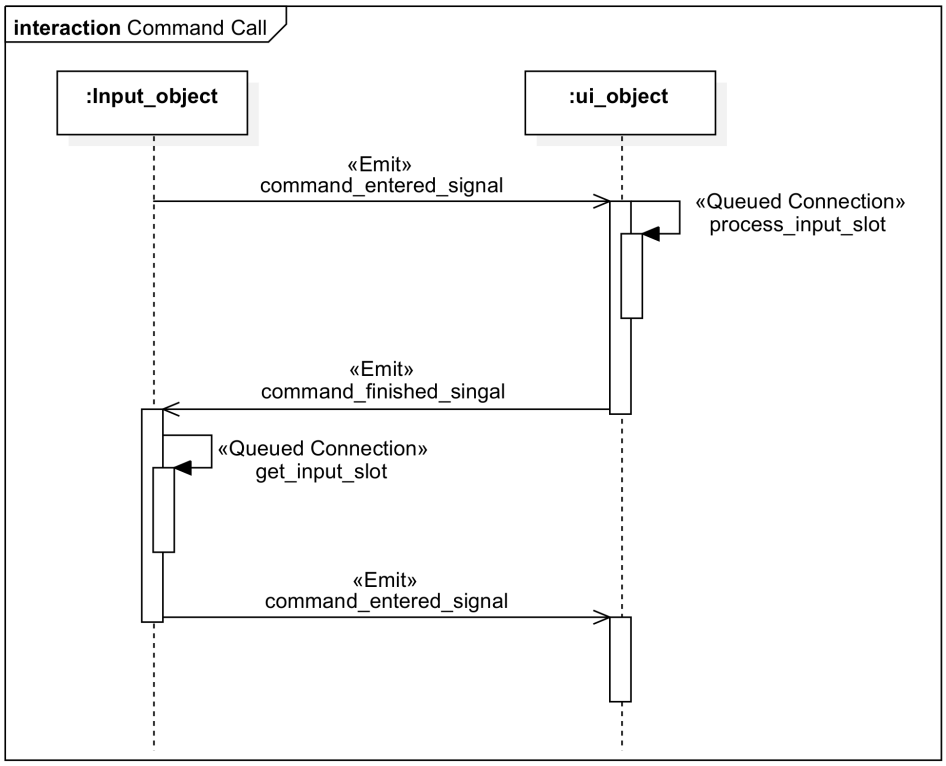


Figure 6.2: Signal-slot structure of a command call

Lastly, this class plays a major role in displaying dynamic output. When a command which prints dynamic output is called, the information is passed to this class using a signal-slot connection, and the main thread is used to call the *refresh_dynamic_command_slot* of this class, printing output and updating it whenever a message from the relevant system is received. Figure 6.3 displays the call structure for Master process messages. Output produced by the overridden *compose_dynamic_output* method of the given command class is passed to this slot, the terminal is cleared using an ANSI escape sequence, and the passed output is printed. If a press of the *return* key is detected, the dynamic flag is unset and the terminal returns to normal mode in which it accepts commands.

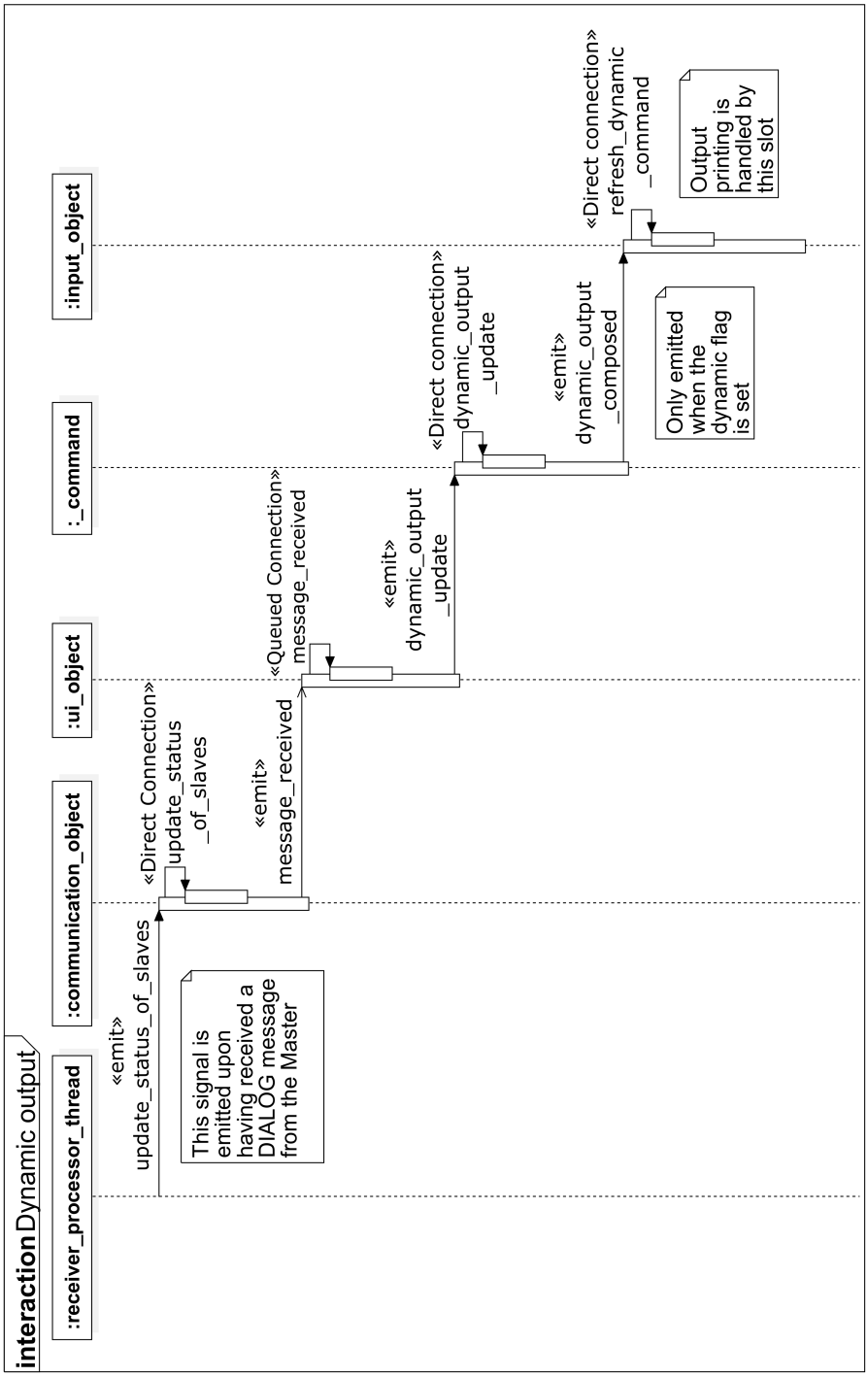


Figure 6.3: Signal-slot structure of the dynamic output feature. Note the two lower direct connections, meaning the calls are executed in the main thread. This is so the input thread can scan for a press of the *return* key to end dynamic output. The *_command* class represents only commands with dynamic output capabilities in this case. Parameters have been omitted in this figure.

6.3 Ui_object classs

The main role of the *ui_object* class is to implement the reactive logic of the application in respect to the rest of the DAQ. Parsing of messages received from the Master process is implemented in its *parse_message* method, being the second most demanding part of the application in terms of processing power – as mentioned previously, these messages are received every 50 milliseconds on average.

A major portion of the messages consists of a complete list of FPGA register values – the *FPGA_object* class is used govern and store this information. Furthermore, some of the register values are also passed to the classes aggregated by the *slinks* class. The *RegistersLib* class is used to represent the registers, being a standardized class shared by several processes of the DAQ.

The *ui_object* class also manages command availability. The class uses three attributes for this purpose: *commands* (a hash table mapping string keys to *base_command* class pointers), *available_commands* (analogous to *commands*) and *commands_lifo* (a stack of such hash tables). *Commands* is used to contain pointers to all instances of *_command* classes, *available_commands* stores only those which are available to the user at a given time, and *commands_lifo* is used to manage the menu structure. The *enable_command* and *disable_command* methods are used to toggle availability of the individual commands, using the key (command name) as a parameter. These methods are then called with the appropriate arguments in the *update_state* method, which occurs when the Master process state changes.

The *tcs_object* class is another class aggregated by the *ui_object* class, managing information concerning the TCS. Using a signal-slot connection with the *communication_object* class, the *ui_object* updates the TCS information encapsulated within it received from the DIM system. There are three DIM services which provide information concerning the TCS: *TCS_prescaler_service* (trigger values and prescaler values), *TCS_status_service* (calibration trigger configuration) and *TCS_spill_indicator_service* (burst information). Finally, the *ui_object* class also manages a connection to the database (using DIALOG to communicate with a process which communicates with the database directly). This represents information such as the lists of structure types, run types, or prescaler presets. This information is requested during initialization or based on the Master process state.

6.4 Slinks class

This class aggregates classes related to the new DAQ hardware, i.e., *mux_object*, *switch_object* and *core_object* – its main role is to initialize and manage the instances of these classes. Since this class is used to represent the hardware structure of the DAQ, it is only instantiated once the Master process has entered the *configured* state (i.e., after a structure type has been chosen and an XML file describing it has been created by the Master process). Similarly to the GUI, the *daq_structure* class aggregated by this class handles parsing of the DAQ structure file and maintains a direct connection to the database to retrieve additional information related to the individual source ids.

The *update_link* method of this class is called every time a message is received from the Master process, passing an updated representation of the register values to *mux_object*, *switch_object* and *core_object*

classes. *The update_state_slot* method is called every time the DAQ is reconfigured, re-reading the DAQ structure file.

```
<?xml version="1.0" encoding="UTF-8"?>
   <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
   attributeFormDefault="unqualified">
         <xs:element name="structure">
               <xs:complexType>
                     <xs:sequence>
                           <xs:element name="hardware">
                                 <xs:complexType>
                                       <xs:sequence>
                                             <xs:element name="controlPR" type="xs:int"/>
                                             <xs:element name="type" type="xs:string"/>
                                             <xs:element name="action" type="xs:string"/>
                                             <xs:element name="port" type="xs:int"/>
                                             <xs:element name="source" type="xs:int"/>
                                             <xs:element name="process" type="xs:int"/>
                                             <xs:element name="name" type="xs:string"/>
                                             <xs:element name="contact" type="xs:string"/>
                                             <xs:element name="Tooltip" type="xs:string"/>
                                             <xs:element ref="hardware" minOccurs="0"
                                             maxOccurs="unbounded"/>
                                       </xs:sequence>
                                 </xs:complexType>
                           </xs:element>
                     </xs:sequence>
               </xs:complexType>
         </xs:element>
   </xs:schema>
```

Listing 6.2: XML schema definition of the DAQ structure file created by the Master process

The *mux_object*, *switch_object* and *core_object* classes also contain methods named *update_link* and *update_state*, both handling the updating of inner representations when the corresponding *Slinks* class method is called.

6.5 Input_checker class

The *input_checker* class handles input validation of command arguments. A single instance is present in each *_command* class whose related command accepts arguments. After initialization, the input can be checked for validity using the overloaded *is_valid* method.

This class uses the concept of "phases" when pertaining to promptive input, each phase corresponding to a set of valid inputs that is available at a point in time when asking the user using the overriden *ask_for_args* method of a given *_command* class. For direct input, the phases are mapped to individual switches.

Parametric switches can be flagged as "leeway switches", meaning no input validation will be performed for that switch (e.g., lock-in messages). A command can also be flagged as absolute leeway. This means

no input checking will be done and any parameters the user might have input will be passed over to the command. The LOAD command is flagged as such, as it only internally calls the LOAD utility which implements its own input validation.

```
>LOAD -A pccofe01 800 810 811 1024
```

Listing 6.3: An example of LOAD command argument input

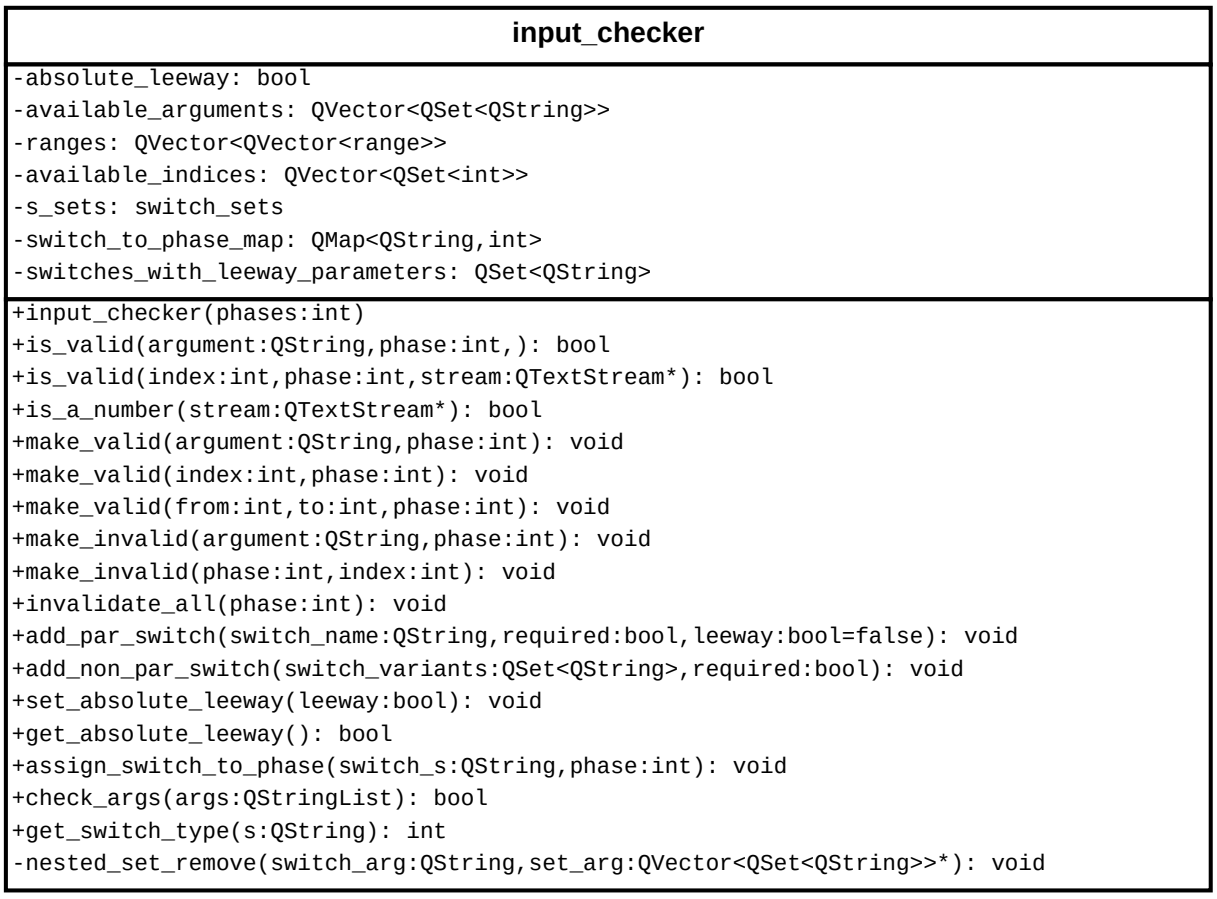


Figure 6.4: The input_checker class

6.6 Maskerror_s_command

Out of the 56 *_command* classes, this class has been selected as a representative to demonstrate typical command implementation.

Initialization of the two-phase input_checker is shown in listing 6.4. A required parametric switch is then associated with each phase, *-p* representing the switch port index with the range of 0 to 15, and *-i* the error

index with the range of 0 to 11. A required non-parametric switch is also added, indicating whether the given mask should be enabled or disabled. Note there are only two phases despite there being three bits of information. This is given by the nature of the promptive argument input in this particular command, which shows the status of the masks to the user, and if a port and index are chosen, the mask is set to the other possible state (i.e., unmasked, if the error is already masked, and vice-versa).

```
void maskerror_s_command::initialize_inputchecker()
{
    ipc = new input_checker(2);
    ipc->make_valid(0,15,0);
    ipc->make_valid(0,11,1);

    ipc->add_par_switch("-p",true);
    ipc->assign_switch_to_phase("-p",0);

    ipc->add_par_switch("-i",true);
    ipc->assign_switch_to_phase("-i", 1);

    QSet<QString> variants;
    variants.insert("-e");
    variants.insert("-d");

    ipc->add_non_par_switch(variants,true);
}
```

Listing 6.4: The implementation of the *initialize_inputchecker* method in the *maskerror_s_command* class

The *process_args* method shown in listing 6.5 sets the required values after extracting the direct argument information from the representation created by the *parse_args method*. By convention, the names of all attributes which represent the value of user input are prepended with *user_i_*, no matter which way they were received.

```
QVariant maskerror_s_command::process_args()
{
    if(non_par_switches.contains("-e"))
        user_i_mask = true;

    if(non_par_switches.contains("-d"))
        user_i_mask = false;

    user_i_port = switch_to_val_map["-p"].toUInt();
    user_i_error_num = switch_to_val_map["-i"].toUInt();

    return QVariant(1);
}
```

Listing 6.5: The implementation of the *process_args* method in the *maskerror_s_command* class

The listing of the *ask_for_args* method shows the implementation of how the user is prompted for input in the two separate phases, first being prompted for the port and then the error index. Note the canonical input processing mode.

```
QVariant maskerror_s_command::ask_for_args()
{
    QTextStream(stdout) << "Which port?" << endl;
    QTextStream input(stdin);
    input >> user_i_port;

    if(!ipc->is_valid(user_i_port,0,&input))
        return QVariant(0);

    for (quint8 i = 0; i < 12; i++)
    {
        QTextStream(stdout) << "Mask " << i << ": " << slinks_ptr->error_descriptions[i] << " - ";

        if(switch_object_ptr->error_objects[user_i_port]->is_masked(i))
            QTextStream(stdout) << "Masked";
        else
            QTextStream(stdout) << "Not masked";

        QTextStream(stdout) << " " << endl;
    }

    QTextStream(stdout) << "Enter the index of error which you would like to mask/unmask:" << endl;
    QTextStream input2(stdin);
    input2 >> user_i_error_num;

    if(!ipc->is_valid(user_i_error_num,1,&input2))
        return QVariant(0);

    if(switch_object_ptr->error_objects[user_i_port]->is_masked(user_i_error_num))
        user_i_mask = false;
    else
        user_i_mask = true;

    return QVariant(1);
}
```

Listing 6.6: The implementation of the *ask_for_args* method in the *maskerror_s_command* class

Finally, the *perform* method, the implementation of which is shown in listing 6.7, composes a message to be sent to the Master process. The user input is serialized into a pre-defined format, encapsulated using the Transport Protocol (a standard for message encapsulation within the DAQ, see section 1.5.2 of [16] for details), and a signal which triggers its sending to the Master process using DIALOG is emitted.

```
QVariant maskerror_s_command::perform()
{
    QByteArray msg;
    QByteArray temp("SCCM");

    if(user_i_mask)
        temp.append("13  |");
    else
        temp.append("12  |");

    temp.append(QByteArray::number(switch_object_ptr->error_objects[user_i_port]->get_cprocessid()));
    temp.append("|");
    temp.append(QByteArray::number(user_i_port));
    temp.append("|");
    temp.append(QByteArray::number(user_i_error_num));
    temp.append("    ");
    msg=transportProtocol.makeMsg(*gui_id,42,"1111",temp);

    emit sendCommandMessageSignal("S_RUN_CONTROL",msg);

    return QVariant(1);
}
```

Listing 6.7: The implementation of the *perform* method in the *maskerror_s_command* class

Chapter 7

Testing

This Chapter aims to describe the philosophy and methodology used during the testing of the application.

7.1 Integration tests

As the application interacts with a number of different systems, it was essential to test whether the interaction between systems behaves as expected, especially in error states, e.g. one of the systems being unavailable.

In order to monitor the network behavior of the application, the DIALOGcommunicationGUI and DIM information display (DID) applications were used for DIALOG and DIM network communication, respectively. These applications allow for monitoring of commands and services registered with the DIALOG/DIM application server, as well as their subscribed clients, and the messages being sent. This was used to confirm the process subscribes to the services correctly and sends correct commands.

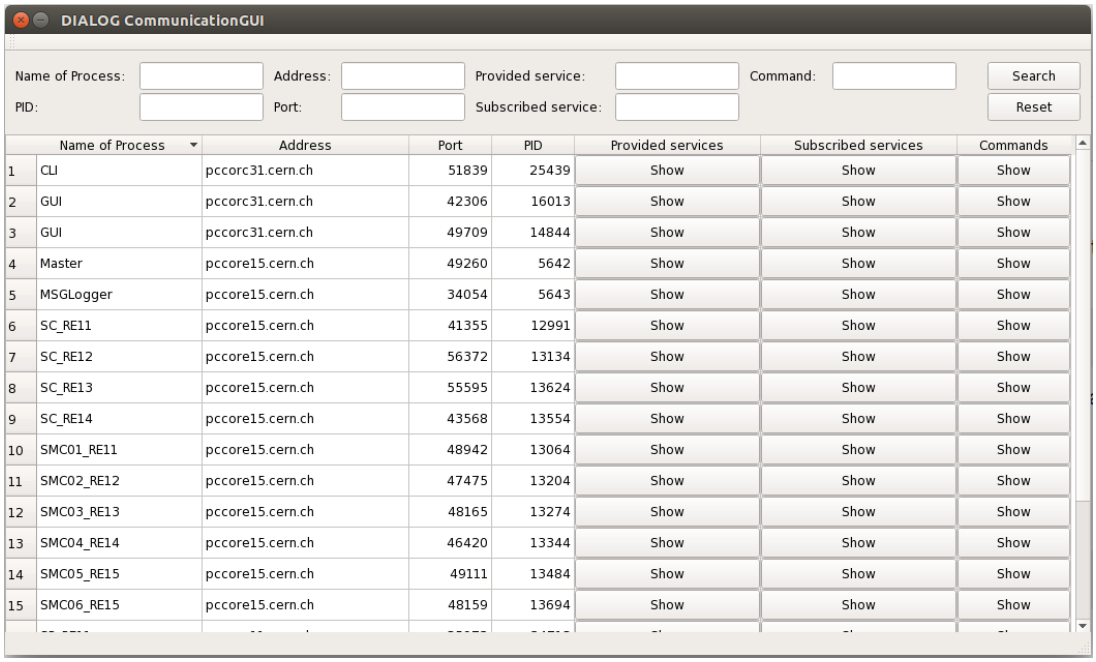


Figure 7.1: The DIALOGcommunicationGUI utility

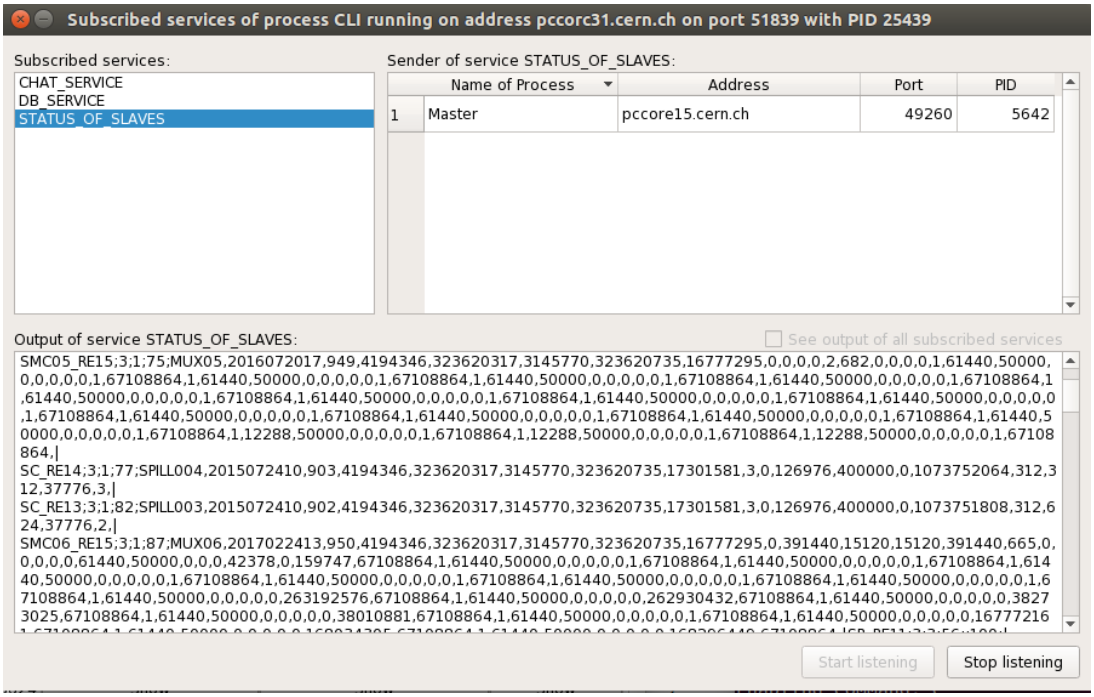


Figure 7.2: Monitoring subscribed services and their output using DIALOGcommunicationGUI

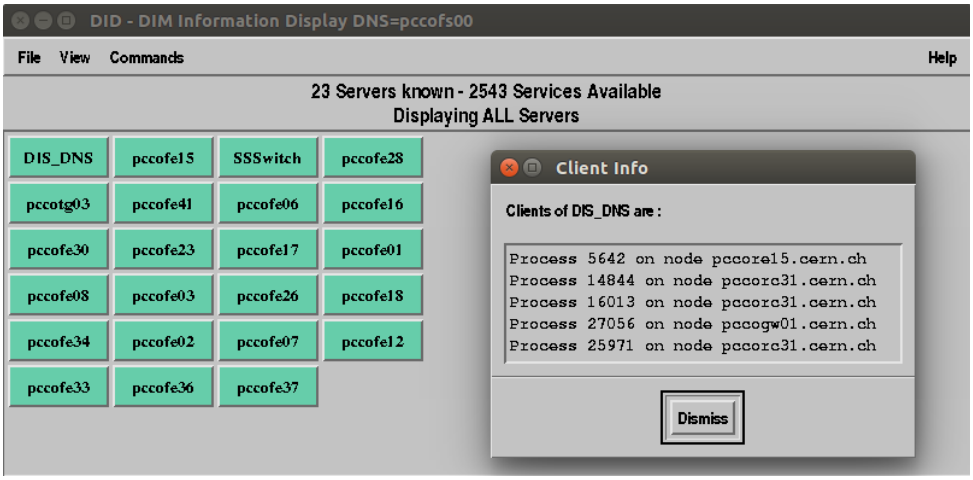


Figure 7.3: The DIM information display utility

The selection of test cases described below combines the use of the above mentioned tools with functional tests.

Test case 1 – TCS DIM unavailable
This test case evaluates the behavior of every TCS-related command while the various TCS DIM commands/services are unavailable.

Command:	Behavior:	Expected?
Calibration_info	Prints a "DIM error" message	Yes
Triggervalues_show	Values show -1, channels named "DIM error"	Yes
Channel_set	Prints a "DIM command TCS/main/Prescaler unknown" message	Yes
Calibration_set	Prints a "DIM command TCS/main/Control unknown" message	Yes

Table 7.1: TCS related commands evaluated during integration testing

Test case 2 – Spill structure DIM unavailable
The need to possess information concerning the spill structure creates dependency on yet another external system. All commands interacting with this information were tested. Note that the commands print a "Not yet received" value instead of an error value, as spill structure information is only received every 30s.

Command:	Behavior:	Expected?
Runinfo	Prints a Value of the spill structure cell reads "Not yet received"	Yes
Startrun	Value of the spill structure cell reads "Not yet received"	Yes
Spillstructure_show	Value of the spill structure cell reads "Not yet received"	Yes
Runconfig_show	Value of the spill structure cell reads "Not yet received"	Yes
Spillstructure_switch	Prints a "DIM command SSS.command unavailable" message	Yes

Table 7.2: Spill structure related commands evaluated during integration testing

Test case 3 – Master process unavailable
In this test case, the CLI was started during absence of the Master process. In this case, a "No Master process found" message was printed repeatedly every time the application tried to establish a connection, which is the expected behavior.

Test case 4 – Master process crash and restart
This test case evaluates the behavior of the application during a Master process crash and its subsequent restart. The SIGKILL signal was sent to the Master process to simulate a crash, followed by a command to start the process again.

Behavior during two different states of the application was tested: non-canonical input mode and dynamic output mode. In the first case, a "No Master process found" message was printed repeatedly, similarly to the previous test case. Once the Master process had been restarted, the application printed a "Master state changed, please wait" message followed by a "...done" message, enabling commands appropriate for the Master waiting state, which is the expected behavior.

In the second case, the dynamic output stopped updating once the Master process was killed, and resumed updating as soon as it was restarted and transitioned to a compatible state, which is the expected behavior.

7.2 Functional tests

Automated functional testing was performed using Expect [41] – a tcl-based utility for programmed dialogue with interactive programs, simulating user interaction. As it would be highly impractical and time consuming to create a multitude of tests manually, the Autoexpect utility is provided along with Expect – a tool for Expect code generation based on user-recorded interaction. As such, Autoexpect was

used to create to generate a basis of the tests, which was then subsequently modified. A selection of the tests performed can be found below.

```
set timeout -1
spawn $env(SHELL)
match_max 100000
send -- "CLI\r"
expect -exact "...done\r"
send -- "tcs    "
expect -exact "tcs      [47;0H[Ktcs_menu"
send -- "\r"
expect -exact "\r
Executing command \"tcs_menu\"\r
Enabling command: c\r
Enabling command: b\r
Enabling command: trigger_values\r
Enabling command: calibration_info\r"
send -- "c\r"
expect -exact "c\r
Executing command \"c\"\r
b - Returns back a level in the menu\r
c - Shows all available commands\r
calibration_info - Posts current calibration settings\r
trigger_values - Posts current trigger info and settings\r"
send -- "b\r"
expect -exact "b\r
Executing command \"b\"\r"
send -- "q      "
expect -exact "q        [47;0H[Kquit"
send -- "\r"
expect -exact "\r
Executing command \"quit\"
sleep 1
send -- "exit\r"
expect eof
```

Listing 7.1: An Expect test which launches the application, waits for initialization to finish, enters the TCS menu using tab completion, uses the *c* command, validates its output, and exits the application

7.3 Performance analysis

Effective performance is a feature vital to the CLI, as several instances need to be run at the same time. It is thus essential the application is not CPU or memory-heavy. The Linux Pidstat utility was used to log CPU and memory usage statistics of CLI instances in various states for the duration of one week, peaking at 3% CPU usage and 1% memory usage on the target machines, which is an acceptable result.

```
Linux 2.6.32-573.22.1.el6.x86_64 (pccorc31.cern.ch)     27.2.2017       _x86_64_        (2 CPU)

#      Time       PID    %usr %system  %guest    %CPU   CPU  minflt/s  majflt/s     VSZ    RSS   %MEM  Command
 1488194684     13821    2,00    0,00    0,00    2,00     0      0,00      0,00 1189896  17908   0,46  CLI
```

Listing 7.2: Output of the *pidstat -r -h -u -p <PID> 15* command

Furthermore, the Valgrind memory analyzer tool was used to check for memory leaks, confirming their absence. The Valgrind function profiler tool was also used in order to optimize the application.

```
set timeout -1
spawn $env(SHELL)
match_max 100000
send -- "CLI\r"
expect -exact "
...done\r"
send -- "runconfig_menu\r"
expect -exact "runconfig_menu\r
Executing command \"runconfig_menu\"\r
Enabling command: c\r
Enabling command: b\r
Enabling command: runconfig_show\r
Enabling command: spillstructure_show\r
Enabling command: runnumber_show\r
Enabling command: numberofspills_show\r"
send -- "numb   "
expect -exact "numb     [59;0H[Knumberofspills_show"
send -- "\r"
expect -exact "\r
Executing command \"numberofspills_show\"\r"
expect -re "The active spillcount is: (\[0-9]+)\r"
send -- "b\r"
expect -exact "b\r
Executing command \"b\"\r"
send -- "q      "
expect -exact "q        [59;0H[Kquit"
send -- "\r"
expect -exact "\r
Executing command \"quit\"
sleep 1
send -- "exit\r"
expect eof
```

Listing 7.3: An Expect test which launches the application, waits for initialization to finish, enters the run configuration menu, uses the *numberofspills_show* command, validates its output, and exits the application

7.4 Usability tests

Five different participants which regularly need to access the DAQ remotely were selected for usability testing. They were asked to try to carry out the tasks they normally perform remotely. This included:

- Starting a run
- Viewing FPGA register values
- Changing the DAQ deadtime
- Viewing trigger values

After a short period of familiarization with the interface, the participants managed to carry out all the tasks with relative ease. The only exception was the action of starting a run, during which one participant

was unsure how to set the parameters and needed assistance. This could be possibly solved by reworking the *startrun* command to accept direct argument input in the future after further evaluation. For the time being, the command description printed out by the *c* command was updated to address this issue.

One of the participants also inquired about viewing the history of the trigger values (i.e., being able to scroll up and view previous values). As it is possible to view the history once dynamic output is terminated, the participant was informed and this information was subsequently included in the user manual of the application.

Conclusion

After a thorough analysis of the COMPASS DAQ control system, its graphical user interface and related systems, a command-line interface was designed, developed, deployed and successfully tested in the CERN environment – it has been a part of the software equipment of the COMPASS experiment since September 2016. A presentation on the interface was given at the COMPASS weekly meeting, and the application has been met with positive feedback. Members of the COMPASS collaboration now make use of this interface in order to access the DAQ remotely.

Suggestions to rework the software architecture of the DAQ and the integration with other systems have also been given in order to improve modularity of the system, especially for reuse at other experiments, but also to facilitate conversion of the GUI to a process disjoint from the COMPASS network. These changes are currently planned to be carried out by the author of this thesis over the course of the years 2017 and 2018.

The new DAQ and its control system are currently planned to be used at one physics experiment other than COMPASS – the experiment NA64, whose aim is to extend the current understanding of dark matter and possibly prove its existence [43]. In conclusion, the application developed in this thesis currently sees frequent use at the COMPASS experiment, and is likely to enjoy the privilege of being used at additional experiments in the future.

Bibliography

[1] P. Abbon *et al.*: **The COMPASS experiment at CERN**, CERN-PH-EP/2007-001, January 2007

[2] V. Yu. Alexakhin *et al.*: **COMPASS-II Proposal**, CERN-SPSC-2010-014; SPSC-P-340, May 2010

[3] **COMPASS: COmmon Muon Proton Apparatus for Structure and Spectroscopy** [online] Available at: http://wwwcompass.cern.ch [Accessed 15 February 2017]

[4] J. Nový: **COMPASS DAQ - Basic Control System**; Diploma thesis, Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University, 2012

[5] CASTOR - **CERN Advanced Storage manager** [online] Available at: http://castor.web.cern.ch [Accessed 2 February 2017]

[6] B. Grube: **A Trigger Control System for COMPASS and A Measurement of the Transverse Polarization of Λ and Ξ Hyperons from Quasi-Real Photo-Production**; Doctoral Thesis, Technical University of Munich, 2006

[7] **iMUX/HGESICA module** [online] Available at: http://wwwcompass.cern.ch/twiki/pub/Detectors/FrontEndElectronics/imux_manual.pdf [Accessed 2 February 2017]

[8] **Electronic developments for COMPASS at Freiburg** [online] Available at: http://hpfr02.physik.uni-freiburg.de/projects/compass/electronics/catch.html [Accessed 2 February 2017]

[9] **The GANDALF Module** [online] Available at: http://wwwhad.physik.uni-freiburg.de/gandalf/pages/hardware/the-gandalf-module.php?lang=EN [Accessed 2 February 2017].

[10] H. Sakulin: **Field Programmable Gate Arrays**, In: International School of Trigger and Data Acquisition, Krakow, February 2012

[11] J. Nový: **Processing of large quantity of data from the COMPASS experiment**; Written dissertation preparation, Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University, 2016

[12] V. Baggiolini *et al.*: **The CESAR project - Using J2EE for accelerator controls**; In: 9th International Conference on Accelerator and Large Experimental Physics Control Systems, Gyeongju, Korea, 13-17 October 2003, pp.269 Available at: http://accelconf.web.cern.ch/AccelConf/ica03/PAPERS/TU512.PDF

[13] P. Bordalo *et al.* **Control Systems: an Application to a High Energy Physics Experiment (COMPASS)**; In: Proceedings of the 2012 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR 2012) 20-25 arXiv:1206.3709

[14] **COMPASS shift list** [online] Available at: https://compassshifts.web.cern.ch (credentials required) [Accessed 17 February 2017]

[15] **The IPbus Protocol** [online] Available at: http://ohm.bu.edu/ chill90/ipbus/ipbus_protocol_v2_0.pdf [Accessed 12 December 2016]

[16] A. Květoň: **State machines of the data acquisition system of the COMPASS experiment at CERN**; Bachelor's thesis, Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University, 2015

[17] Gaspar C., Dönszelmann: **DIM, a Portable, LightWeight Package for Information Publishing, Data Transfer and Inter-process Communication**, Presented at: International Conference on Computing in High Energy and Nuclear Physics (Padova, Italy, February 2000)

[18] **Distributed Information Management System** [online] Available at: http://dim.web.cern.ch/dim/ [Accessed 4 February 2017]

[19] **Qt documentantion** [online] : Available at: http://doc.qt.io [Accessed 4 February 2017]

[20] **Qt developer network** [online] Available at: http://www.qt.io/developers/ [Accessed 4 February 2017]

[21] P. Abbon *et al.*: **The COMPASS Setup for Physics with Hadron Beams**, CERN-PH-EP-2014-247, October 2014

[22] I. Antcheva *et al.*: **ROOT – A C++ framework for petabyte data storage, statistical analysis and visualization** In: Computer Physics Communications Volume 180, Issue 12, December 2009, Pages 2499–2512

[23] **The S-LINK Interface Specification** [online] Available at: http://hsi.web.cern.ch/HSI/s-link/spec/spec/s-link.pdf [Accessed 4 June 2015]

[24] M. Jandek: **User interface for control of logbook of COMPASS experiment at CERN**; Bachelor's thesis, Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University, 2016

[25] L. Schmitt *et al*: **The DAQ of the COMPASS experiment** In: 13th IEEE-NPSS Real Time Conf., Montreal, Canada, 1823 May 2003 p 439-444

[26] T. Anticic *et al.*: **ALICE DAQ and ECS User's Guide** CERN, EDMS 616039, January 2006

[27] L. Schmit: **DATE Run Control Tutorial for COMPASS** [online] Available at: http://wwwcompass.cern.ch/compass/detector/daq/date/runControl/ [Accessed 18 April 2017]

[28] B. Franek *et al.*: **SMI++ object oriented framework for designing and implementing distributed control systems** In: Nuclear Science Symposium Conference Record, October 2004 IEEE DOI: 10.1109/NSSMIC.2004.1462600

[29] M. Bodlák *et al.*: **FPGA based data acquisition system for COMPASS experiment** In: Journal of Physics: Conference Series. 2014-06-11, vol. 513, issue 1 DOI: 10.1088/1742-6596/513/1/012029 Available at: http://iopscience.iop.org/1742-6596/513/1/012029/

[30] J. Vondra: **Graphical user interface for control system of the COMPASS experiment at CERN**; Bachelor's thesis, Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University, 2014

[31] J. Nový *et al.*: **Pilot run of the new DAQ of the COMPASS experiment** In: Journal of Physics: Conference Series. 2015-06-11, vol. 664, issue 8. DOI: 10.1088/1742-6596/664/8/082042 Available at: http://iopscience.iop.org/article/10.1088/1742-6596/664/8/082042

[32] Y. bai *et al.*: **Overview and future developments of the FPGA-based DAQ of COMPASS** In: Journal of Instrumentation. 2016-02-09, vol. 11 Available at: http://iopscience.iop.org/article/10.1088/1748-0221/11/02/C02025

[33] G. Bauer *et al.*: **The run control system of the CMS experiment** In: Journal of Physics: Conference Series, Volume 119, Part 2 Available at: http://iopscience.iop.org/article/10.1088/1742-6596/119/2/022010

[34] T. Schmidt: **A Common Readout Driver for the COMPASS Experiment**; Doctoral Thesis, Albert Ludwigs University of Freiburg, May 2002

[35] **CNIC security policy for controls** [online] Available at: https://edms.cern.ch/ui/file/584092/2.4/CNIC_Security_Policy_V2.4.pdf [Accessed 17 February 2017]

[36] **SOCKS protocol version 5** [online] Available at: https://tools.ietf.org/html/rfc1928 [Accessed 12 February 2017]

[37] **The Transport Layer Security (TLS) Protocol Version 1.2** [online] Available at: https://tools.ietf.org/html/rfc5246 [Accessed 12 February 2017]

[38] **Linux at CERN** [online] Available at: http://linux.web.cern.ch/linux/scientific6/ [Accessed February 20176]

[39] **Tips & tricks for software used at COMPASS** [online] Available at: http://physics.mff.cuni.cz/kfnt/cern/softintro/ [Accessed 12 February 2017]

[40] **The GNU C Library manual** [online] Available at: https://www.gnu.org/software/libc/manual/ [Accessed 24 february 2017]

[41] **The Linux man-pages project** [online] Available at: https://www.kernel.org/doc/man-pages/ [Accessed 21 february 2017]

[42] **The Linux Documentation Project** [online] Available at: http://www.tldp.org [Accessed 21 February 2017]

[43] S. Andreas *et al.*: **Proposal for an Experiment to Search for Light Dark Matter at the SPS** arXiv:1312.3309 [hep-ex]

Appendix A

CD contents

- This document in the pdf and tex formats
- Source code of the CLI application
- User manual

Appendix B

User manual

DAQ CLI user guide

Antonín Květoň
Report bugs/send suggestions to: antonin.kveton@cern.ch

February 27, 2017

Contents

1 Introduction

The daq CLI operates under the same locking philosophy as the GUI. The functionality it provides is identical to that of the GUI. The main advantage of the CLI is the fact that one does not need *ssh -X* in order to connect to it remotely. Therefore, one can even control the DAQ from their smartphone, if they so desire.

To launch the CLI, simply type *CLI* on any compass machine.

To view commands available at a given time, use the *c* command (a brief description of what the command does will be printed as well).

2 Dynamic output

A large portion of the commands prints dynamic output, i.e., the values update in real time. Such a command will "capture" the CLI - no other output will be printed. Dynamic mode can be left by pressing the enter key, as prompted on the screen. Should the terminal window be too small to contain the dynamic output, it can be resized - the output will adjust to the size with the next value update.

As it is only possible to launch one dynamic command at a time, multiple instances of the CLI need to be launched simultaneously, should the user wish to use several dynamic commands at a time and/or control the DAQ. The CLI is very light-weight, so there is no problem to run multiple instances of it on the PCCORC machines.

If one wishes to view history of the values of a command with dynamic output, they should terminate dynamic output and simply scroll up.

3 Parameters

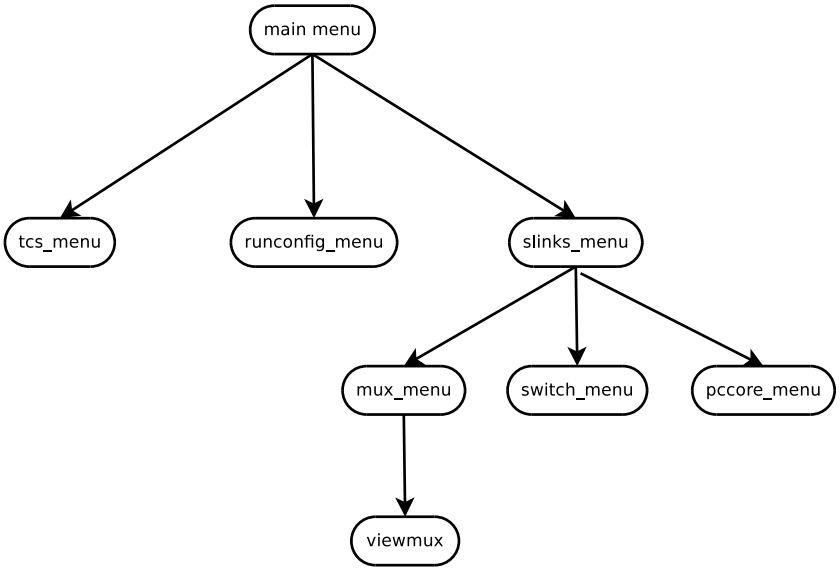
Some of the CLI commands take parameters, but entering them along with the command is never mandatory - if the user does not know the format of the parameters, they should simply launch the command without parameters, as they will be prompted for them afterwards.

4 Quality of life features

It is worth mention that the CLI supports classic TAB command completion, as well as command history (up and down arrow).

5 Menu hierarchy

Many of the commands are hidden under the menu hierarchy - this means whenever a "menu" command is entered, all current commands are disabled and new commands are enabled. The menu hierarchy is the following:



6 A brief how-to

Please bear in mind that control commands will only show up if the CLI is locked in and that the availability of commands is entirely dependent on the state of the Master process.

6.1 Viewing DAQ process states

In order to view the states of the daq processes, use the *poststates* command.

6.1.1 Changing DAQ structure type

In order to change the DAQ structure type, make sure the state of the Master is *waiting*, and subsequently use the *structuretype* command.

6.2 Run control

The following commands are identical to the run control arrow buttons in the GUI:

1. startslaves
2. stopslaves
3. configureslaves

4. unconfigureslaves
5. startrun
6. stoprun

6.2.1 Configuring and starting a run

The following items need to be set before starting a run:

1. Number of spills
2. Prescaler settings
3. Recording enable
4. Run type

If not explicitly set, the values of *Number of spills* and *Prescaler settings* will be pulled from the database, *Recording enable* will default to *true* and *Run type* to *not_defined*. This makes it so that for normal use, the user only has to select the run type.

If recording is disabled or run type not defined, the user will be warned upon starting a run.

All of the above (except for prescaler settings) can be changed in the runconfig menu. The prescaler settings can be changed in the tcs menu.

6.3 Monitoring of DAQ errors

The command *errormonitoring* can be used to quickly display any FPGA register enabled error bits or port errors. This dynamic command provides defacto identical information to the slinks window in the GUI.

6.4 Frontend loading

The LOAD command can be executed from the main menu in exactly the same manner as in any other enviroment on a compass machine, i.e. *LOAD* ⟨parameters⟩.

7 Command parameters

This section describes parameters of the individual commands if one wishes to input arguments directly instead of being prompted for them.

Parameters which are marked as **required** are necessary to include for direct

argument input to work, otherwise the user will be prompted for them.

If the category **required exclusive** is present, exactly one argument from the category has to be included in order for the command to function.

7.1 lock

Required: *-n* ⟨name of the user⟩
Optional: *-m* ⟨lock-in message⟩

7.2 chat_send

Required: *-n* ⟨name of the user⟩, *-n* ⟨message to send⟩

7.3 structuretype_set

Required: *-i* ⟨index of the structure type⟩

7.4 runtype_set

Required: *-i* ⟨index of the run type⟩

7.5 numberofspills_set

Required: *-n* ⟨number of spills⟩

7.6 loadprescalers

Required: *-i* ⟨index of the database-pulled prescaler setting⟩

7.7 recording_set

Required exclusive: *-e* (enable) or *-d* (disable)

7.8 spillstructure_switch

Required exclusive: *-a* (artificial spill structure) or *-s* (SPS spill structure)

7.9 viewmux

Required: *-m* ⟨mux to view⟩

7.10 setport_m

Required: *-m* ⟨mux to set⟩, *-p* ⟨port to set⟩
Required exclusive: *-e* (enable) or *-d* (disable)

7.11 viewport_m

Required: *-m* ⟨mux to view⟩, *-p* ⟨port to view⟩

7.12 maskerror_m

Required: *-m* ⟨mux to set⟩, *-p* ⟨port to view⟩, *-i* ⟨error number to mask⟩
Required exclusive: *-e* (enable mask) or *-d* (disable mask)

7.13 portinfo_m

Required: *-m* ⟨mux to view⟩

7.14 datadetail_m

Required: *-m* ⟨mux to view⟩

7.15 pdatadetail_m

Required: *-m* ⟨mux to view⟩

7.16 setport_s

Required: *-p* ⟨port to set⟩
Required exclusive: *-e* (enable) or *-d* (disable)

7.17 maskerror_s

Required: *-p* ⟨port to set⟩, *-i* ⟨error number to mask⟩
Required exclusive: *-e* (enable mask) or *-d* (disable mask)

7.18 maskerror_r

Required: *-r* ⟨PCCORE to set⟩, *-i* ⟨error number to mask⟩
Required exclusive: *-e* (enable mask) or *-d* (disable mask)

7.19 setmonprescaler_r

Required: *-r* ⟨PCCORE to set⟩, *-s* ⟨prescaling value to set⟩

7.20 channel_set

Required: *-ch* ⟨channel to set⟩, *-v* ⟨prescaling value to set⟩

7.21 calibration_set

Required: *-ch* ⟨channel to set⟩, *-v* ⟨prescaling value to set⟩, *-r* ⟨rate to set⟩:
Optional: *-on* (set onspill to true, else false), *-off* (set offspill to true, else false),

8 Expert mode

When lauching the CLI with the *e* argument, i.e., "*CLI e*", the interface will be launched in expert mode. The sole purpose of expert mode is to make command input faster - the menu hierarchy is eliminated and all commands are available immediately.
Please note: mux commands have to be executed with parameters in this mode, as there was no input from the *viewmux* command! If not launched with parameters, the command will assume mux = 1.

9 Known bugs