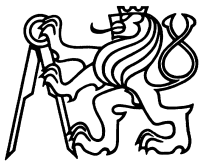
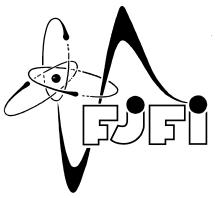


Czech Technical University in Prague
Faculty of Nuclear Sciences and Physical Engineering



State machines of the data acquisition system of the COMPASS experiment at CERN

Stavové automaty systému pro sběr dat experimentu COMPASS v CERN

Bachelor's Degree Project

Author: **Antonín Květoň**

Supervisor: **Ing. Josef Nový**

Consultant: **Ing. Vladimír Jarý, Ph.D.**

Language advisor: **odb.as. Irena Dvořáková, prom.fil**

Academic year: 2014/2015

Acknowledgment:
I would like to thank Ing. Josef Nový and Ing. Vladimír Jarý, Ph.D. for supervising my Bachelor's Degree Project, Ing. Martin Bodlák and prof. Ing. Miroslav Finger, DrSc. for making my journey to CERN possible and odb.as. Irena Dvořáková, prom.fil. for advising me on the language part of my work.

Declaration:
I declare that this Bachelor's Degree Project is all my work and I have cited all sources I have used in the bibliography.

Prague, July 7, 2015 Antonín Květoň

Název práce::

Stavové automaty systému pro sběr dat experimentu COMPASS v CERN

Autor: Antonín Květoň

Obor: Aplikovaná informatika

Druh práce: Bakalářská práce

Vedoucí práce: Ing. Josef Nový, FJFI ČVUT v Praze

Konzultant: Ing. Vladimír Jarý, Ph.D., FJFI ČVUT v Praze

Abstrakt: Cílem této bakalářské práce je nejdříve seznámit čtenáře se systémem sběru dat experimentu COMPASS v CERN, detailně popsat jeho stavové automaty a navrhnout a implementovat algoritmy pro jejich synchronizaci a obecné zpracování chyb. Součástí práce je návrh algoritmu pro obnovení konzistence a detailní popis jeho implementace a integrace do systému. Řešení bylo úspěšně implementováno a otestováno.

Klíčová slova: CERN, COMPASS, DAQ, MASTER-SLAVE, STAVOVÉ AUTOMATY

Title:

State machines of the data acquisition system of the COMPASS experiment at CERN

Author: Antonín Květoň

Abstract: The purpose of this Bachelor's Degree Project is to first acquaint the reader with the data acquisition system of the COMPASS experiment at CERN, then give a detailed description of its state machines and, finally, develop and implement algorithms for their synchronization and general error solving. The solution uses a consistency restoration algorithm, whose design, implementation and integration into the system are described in detail in this project. The solution has been successfully implemented and tested.

Key words: CERN, COMPASS, DAQ, MASTER-SLAVE, STATE MACHINES

Contents

Introduction

This work covers part of development of the new COMPASS data acquisition system (DAQ), described in [1] and [2], and all the results presented here are in accordance with the intended final design of the system.

The work begins by briefly outlining the history, design and purpose of the COMPASS experiment at CERN and then shifts its focus to its DAQ. First, an overview of the essential hardware technologies used in the DAQ is provided, followed by a description of the hardware structure of the DAQ and the role of the technologies in it. The next section gives an overview of the software technologies and the software structure. This work focuses on the software layer, and, therefore, a greater number of pages are dedicated to this section than the previous one.

The analysis of the state machines begins in Chapter 2. A very detailed overview of the state machines' purpose and structure is given, followed by a description of the implementation of the software processes which utilize the state machines. Finally, the motivation for synchronization of the state machines is given, which concludes the introductory part of the work.

Chapter 3 deals with the design of the synchronization and error-solving algorithms. It gives a description of how the algorithms function and explains the resulting changes in the structure of the state machines.

Chapter 4 is concerned with the implementation of the described algorithms. It presents a description of the code used and explains the purpose of a majority of the methods used in great detail. At its end, the Chapter discusses threading. Finally, several examples of usage of the algorithms during concrete scenarios are shown before the work is concluded.

Chapter 1

Panoramic overview of the COMPASS DAQ

1.1 The COMPASS experiment

The purpose of the COMPASS (Common Muon and Proton Apparatus for Structure and Spectroscopy) experiment is the study of nucleon spin structure and hadron spectroscopy. The experiment, which utilizes a polarized target and is situated at the Super Proton Synchrotron (SPS) at CERN in Geneva, Switzerland, was approved conditionally in 1997 and commissioned in 2001. In 2002, the experiment started operating and has since been making use of the various particle beams available at the CERN M2 beam line, primarily the muon and hadron beams [3]. Particle identification is carried out employing a Ring-imaging Cherenkov (RICH) detector, two electro-magnetic calorimeters, two hadron calorimeters and two muon filters. [4]

Over the span of eight years from 2002 to 2009, data-taking had been taking place with the exception of year 2005, in which an accelerator upgrade was realized. The beams used during these years were primarily muon beams and partially hadron beams. As the apparatus of the experiment had been proven to be very versatile, an extension of the COMPASS program has been approved in 2010 by the CERN research board, prolonging its lifespan by seven years and shifting the focus of the experiment to tests of chiral perturbation theory, study of the Drell-Yan process and research in field of Generalised Parton Distributions. [5] [6]

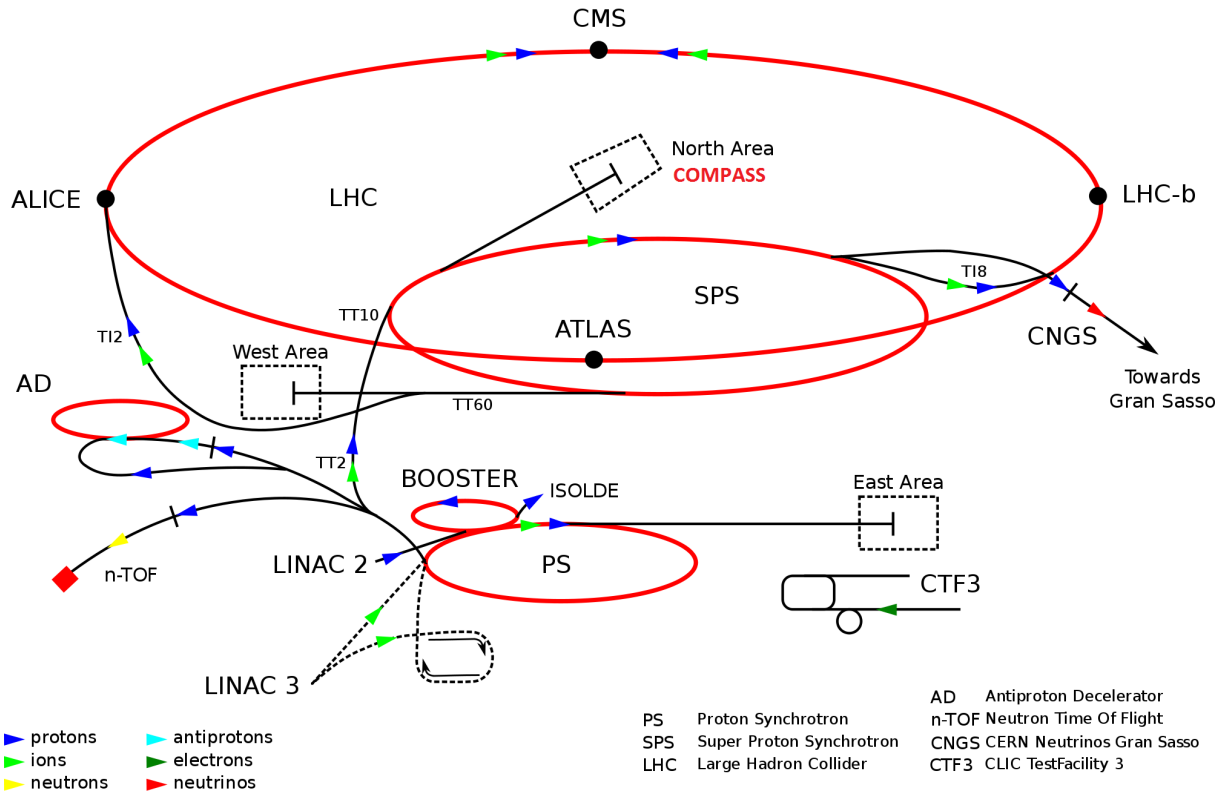


Figure 1.1: COMPASS location within the CERN accelerator complex [1]

1.2 Overview of hardware technologies used in the DAQ

1.2.1 S-Link

The S-Link is a standard for connection of one layer of front-end electronics to the next layer of read-out developed at CERN in 1995. An S-Link comprises a Link Source Card (LSC) and a Link Destination Card (LDC). The former is the transmitter of the data and is mounted on the front-end electronics while the latter is the receiver of data and is mounted on the read-out electronics. Its features include: low latency, error detection and a self-test function. [7]

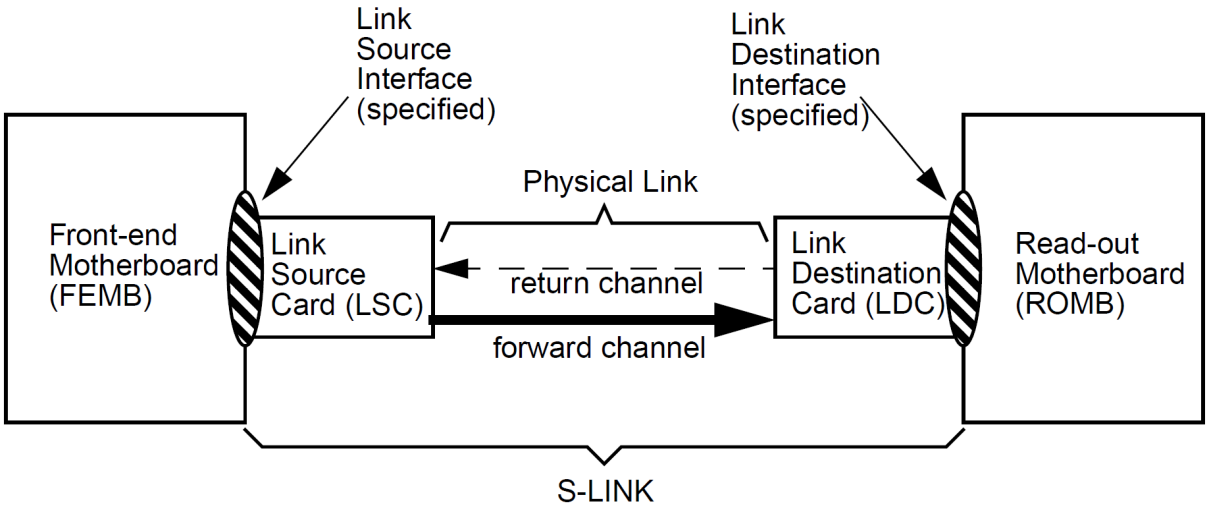


Figure 1.2: The S-Link concept [7]

1.2.2 FPGA cards

FPGAs (Field-programmable gate arrays) are programmable chips which allow the user to implement virtually any digital function. An FPGA chip consists of configurable logic blocks (CLB), a system of customizable interconnects used to connect the blocks to one another and configurable I/O pins connected to the internal matrix made up of the two previously listed elements.

FPGAs typically use volatile memory (i.e., the information is lost upon the loss of power). However, a class of FPGAs which use non-volatile memory to store its configuration does also exist.

The class of programming languages used to program FPGAs is referred to as HDL (Hardware Description Languages). Examples include VHDL and Verilog. The HDL used in the DAQ is VHDL. [1] [8]

1.3 Hardware structure of the DAQ

The COMPASS DAQ is currently undergoing a major hardware and complete software replacement, the first part of which was finished in 2014, and the second part of which is planned to be completed in 2016. As can be seen in Figure 1.3, the new DAQ can be divided into five basic layers, the first one being front-end cards which process analog data from the detectors and convert them to digital form. The front-end cards are connected to HGeSiCA, CATCH and Gandalf modules which make up the second layer. The second layer handles the first level of multiplexing (consolidating multiple data streams into a single stream). More information about HGeSiCA, CATCH and Gandalf modules can be found at [9], [10] and [11], respectively. The data from some of the HGeSiCA and CATCH modules go through SLink

multiplexers and the data from Gandalf modules through TIGER VXS data concentrators, creating a sublayer. SLink multiplexers and TIGER VXS data concentrators are used for multiplexing.

Using S-Links, this sublayer is connected to the third layer, which comprises eight FPGA cards which are referred to as Data Handling Cards (DHC) within this context. The third layer handles another level of multiplexing. S-Links are also used to connect the third layer to the fourth layer, which is made up of a single DHC with switch firmware – this layer handles event building. The fifth layer, again utilizing S-Links for connection to the previous layer, consists of readout computers which run the DAQ software. These computers are collectively referred to as the readout engine. The connection of an S-Link and the memory of a readout computer is handled by a Spillbuffer – a PCI Express card with an FPGA chip and 2GB RAM, which is also partially used for buffering. The acquired data which are to be stored are then sent directly to the CERN CASTOR facility. [6] [12] [13]

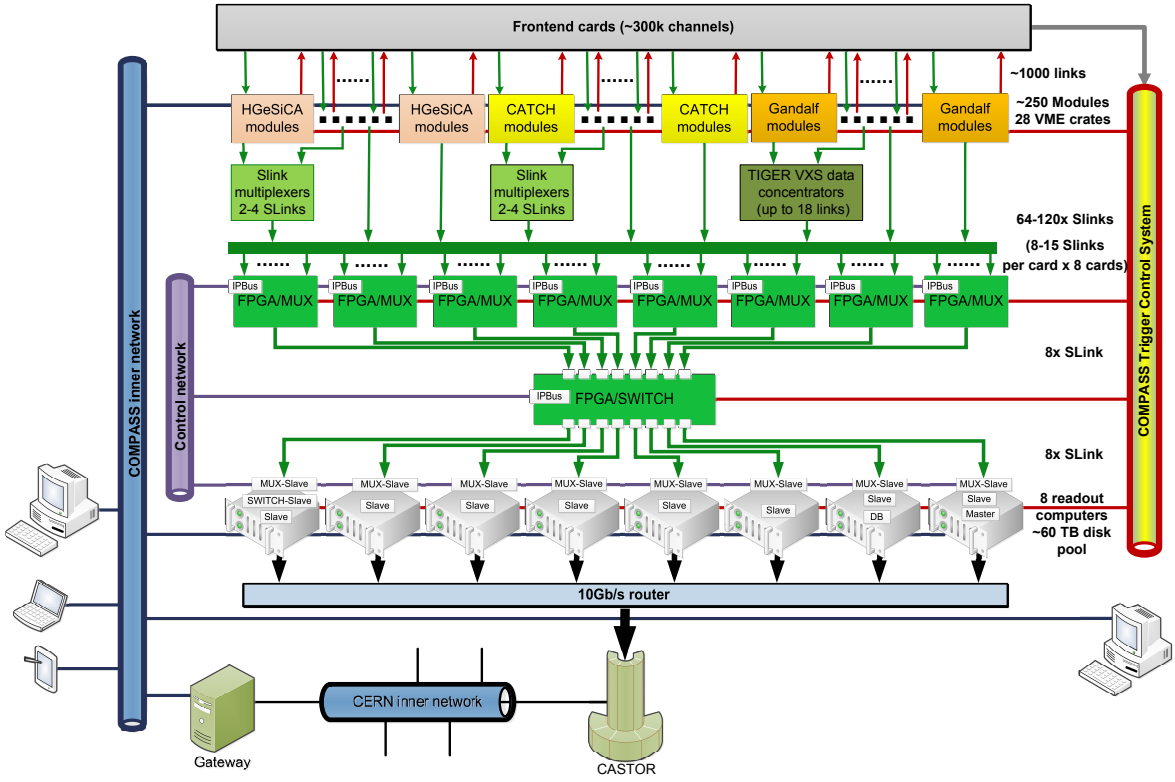


Figure 1.3: Hardware structure of the DAQ [2]

1.4 Overview of software technologies used in the DAQ

1.4.1 Qt framework

Qt is a C++-based cross-platform application framework used for software application development originally developed by Trolltech, starting in 1991. Following its acquisition by Nokia in 2008, the framework was acquired by Digia in 2012. Supported platforms include: Linux/X11, Mac OS X, Windows and embedded Linux.

The primary purpose of the Qt framework is development of applications with graphical user interfaces (GUIs) – this is made simple and efficient with the Qt IDE (Qt creator). Nevertheless, it is possible to develop GUI-less applications using Qt, making use of the wide variety of non-GUI functionalities the framework offers.

The features of Qt which extend the C++ language are realized using the Meta-object system, which consists of three parts: [14]

1. The `QObject` class, which is the base class of all Qt classes – all objects derived from this class can take advantage of the Meta-object system.
2. The `Q_OBJECT` macro, which is used to enable the meta-object features – the macro is situated inside the private section of the given class.
3. The Meta-Object Compiler (moc) – a preprocessor tool which, upon activation, reads header files and generates meta-object code for classes inside whose declaration the `Q_OBJECT` macro is found. The meta-object code, in the form of a C++ source code file, implements the Qt extensions of the C++ language.

1.4.1.1 Signals and slots

Signals and slots are a pivotal feature of the Qt framework. Being a type-safe alternative to callback functions, signals and slots are used for communication between objects – a signal belonging to an object can be "connected" to a slot belonging to a different object. A signal is "emitted" when a given event occurs, and if a slot is connected to it, the code contained in the slot is executed. Signals and slots are particularly important for the DAQ, as one of their main roles in it is acting as the means of inter-thread communication. [14]

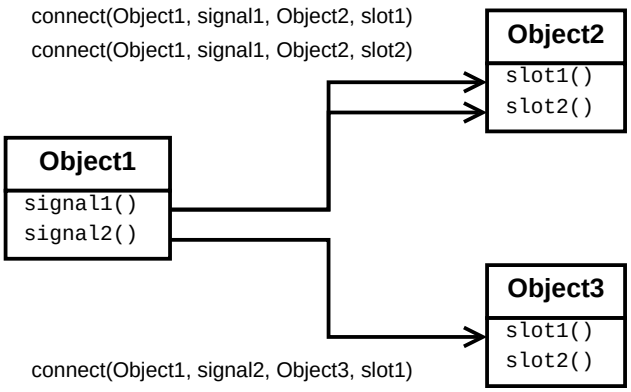


Figure 1.4: The signal-slot system

1.4.2 DIM

The DIM (Distributed Information Management) system is a communication system based on a server-client structure used for data transport over a network. It also incorporates a name server in order to allow for transparency, i.e., a client does not need to know where a server is running. The system was originally developed at the DELPHI experiment at CERN and as it was designed in a generic fashion, it has since found use in several other experiments. [15] [16]

DIM consists of three basic components: Server, Client and the Name Server. The server and client incorporate the idea of "services" and "commands" in order to communicate with each other, while the Name Server acts as the middleman to establish the initial connection in between the server and the client.

Servers register their services with the Name Server, making them available to the clients. The clients can then request service information from the Name Server and subscribe to the according service of the Server. Once subscribed to a service, the clients receive service data every time the server "updates" the service. Clients can send commands (i.e., data) to the server, enabling two-way communication. It should be noted that services serve as a means of 1 to n communication, while commands serve as a means of 1 to 1 communication.

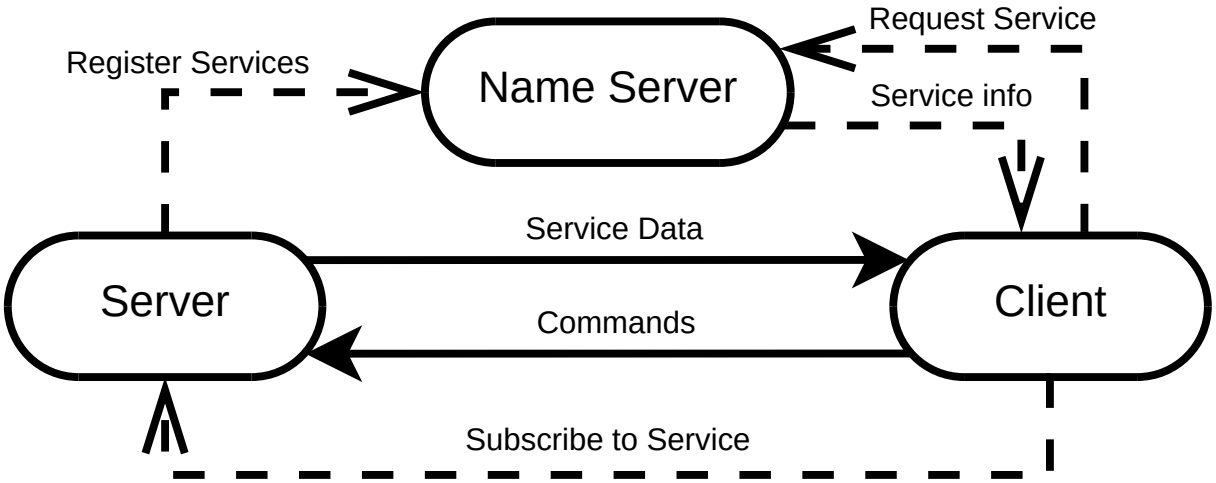


Figure 1.5: The DIM concept

1.4.3 IPBus

The IPBus suite is a package which provides an interface for UDP-based control of Ethernet-attached hardware devices, mainly FPGA cards. It was originally developed for the CMS experiment and consists of two parts, the firmware part and the software part. The firmware part is implemented in VHDL and its purpose is to mediate access to the memory and registers of the hardware devices through Ethernet. The software part is implemented in C++ and its purpose is to provide a connection to the interface of the firmware part. [2] [17]

1.5 Software structure of the DAQ

The DAQ software is deployed on the readout engine, the individual computers of which run the Scientific Linux CERN 6 (SLC6) operating system – detailed information concerning SLC6 can be found at [18]. The software is C++-based and uses the Qt Framework not only for its GUI, but also for its threading. Furthermore, Qt data types and a variety of non-GUI classes are also used in the software. The Qt version used in the DAQ software is 5.2.1. Python and Bash script also find use in the DAQ, their scripts being particularly useful for starting processes remotely using SSH. Finally, XML is used to describe the hardware configuration of the DAQ in so-called XML structure files and the IPBus configuration in so-called XML connection files and address files.

Six main functions are provided by the DAQ software: configuration of the hardware, monitoring of the data taking process, remote control of the hardware, data flow control, logging of information and errors and log browsing. [2]

The DAQ software also includes a connection to an SQL database. The database is used to store, among others: configuration information of the DAQ's hardware, information logs and error logs.

The events read out from the detectors by the DAQ are stored in the DATE format, which is the format used in the DATE (Data Acquisition and Test Environment) software. The DATE software is the data acquisition software that was originally developed for the ALICE experiment and a modified version of which was being used in the COMPASS DAQ before the overhaul. [19]

1.5.1 Processes of the DAQ

The software comprises the following processes: Master, Slave Control, Slave Readout, RunControl GUI, MessageLogger and MessageBrowser.

The Master is a vital process for the DAQ – using DIM, it mediates communication between the Run-Control GUI and the slave processes as well as the communication between the slave processes and the configuration database. It also plays a major role in the DAQ's error handling.

The purpose of the Slave Control process is to configure and monitor the FPGA cards – it is the only process which communicates with the FPGA cards directly. All communication with the FPGA cards is carried out using IPbus.

The Slave Readout is a very resource-demanding process responsible for readout of data from connected devices, as well as its processing and subsequent storage. It comprises a large number of threads.

The RunControl GUI, which can run in two different modes, is the means of user interaction with the DAQ. The first mode, Run Control, provides the user with complete control over the DAQ as well as information concerning the current run and status of the hardware. Only one instance of this mode can run at a time. The second mode, Monitoring, retains the information and status providing capabilities, but does not provide the user with any direct control over the DAQ. There is no practical limit to how many instances of this mode can run at a time.

The MessageLogger is a process which receives status and error messages from all parts of the DAQ and stores them in the database.

The MessageBrowser is a GUI tool used for visualization of these messages. [12]

A diagram showing communication between individual processes of the DAQ can be found in Figure 1.6. The meaning of the colors is as follows:

- **Blue**: Processes vital for data acquisition
- **Red**: Optional monitoring toools
- **Yellow**: Main control GUI
- **Green**: Firmware interface

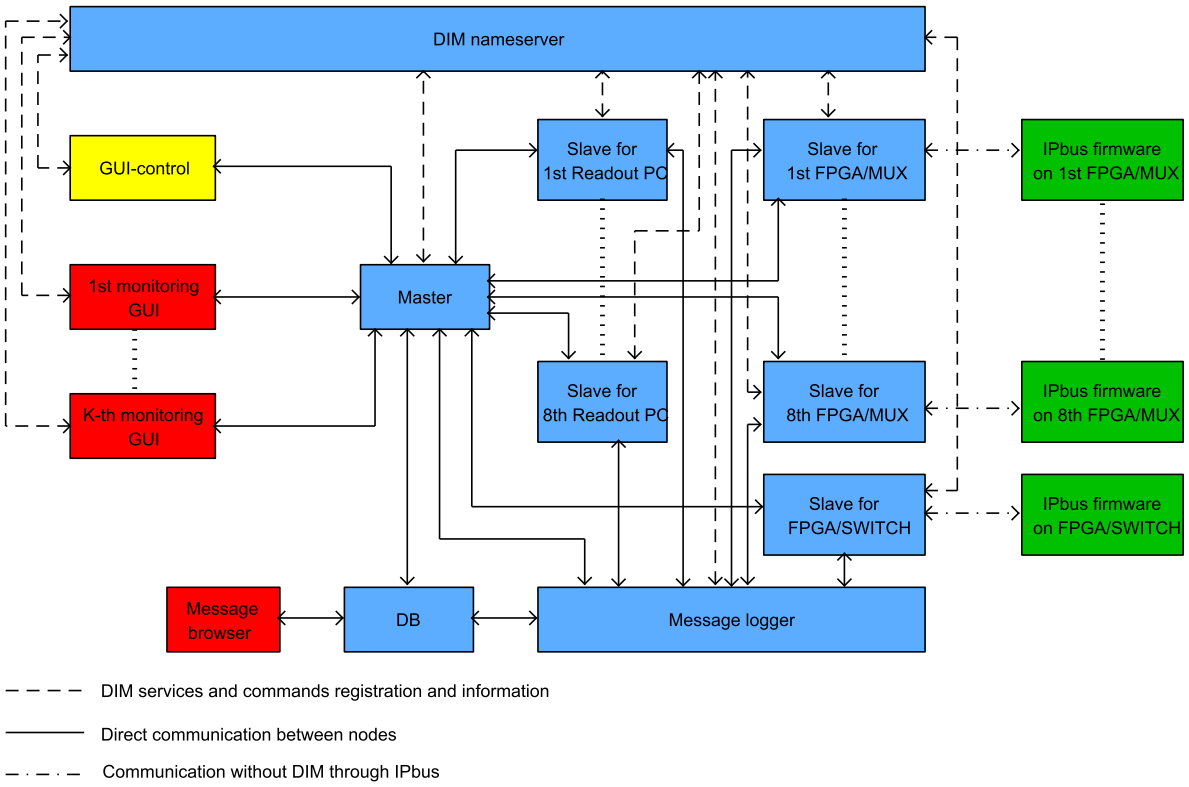


Figure 1.6: The communication diagram of the DAQ [2]

1.5.2 Transport protocol

The transport protocol is the standard for message encapsulation in the DAQ. Every process which uses DIM for communication with the other processes receives and sends messages solely in the transport protocol format.

Using Qt, particularly the `QByteArray` class, the transport protocol is implemented as a library with message making and message parsing functions as well as simple support functions for type conversion. The structure of the protocol can be seen in Table 1.1.

Head			
1.	Data size	4 B	Size of data in 32 b words (= header + body + trailer)
2.	Version	4 B	Version of data protocol
3.	Sender ID	4 B	Unique ID of sender
4.	Message number	4 B	Number of message (identifier)
5.	Receiver ID	4 B	Unique ID of receiver
6.	Message ID	4 B	Unique ID of message (type of message)
7.	Time	4 B	Time stamp 1/2
8.	Time	4 B	Time stamp 2/2
Body			
9.	Body	4·N B	Body of message (N ≥ 0)
Trailer			
10.	Reserved	4 B	0x00000000
11.	Reserved	4 B	0x00000000
12.	Message number	4 B	Number of message (identical to 4.)

Table 1.1: Structure of the transport protocol [1]

Chapter 2

Analysis of the state machines of the DAQ

The Master, Slave Control and Slave Readout processes each have a state machine associated with it, providing a simple, yet powerful indicator which clearly defines the status of the process at a given point in time. Apart for being important for synchronization of the individual processes, this concept has several advantages, namely: the information concerning the slave processes sent to the Master over DIM is light-weight and comprehensive and a simple link between the process layer and the user interface layer is provided in order to convey process information to the user in a straightforward form.

Except for the first block (`Starting/Closing`), the master state machine diagram is identical to the slave state machine diagram – the master state machine generally represents the states of all the slave machines. Moving along the state machine diagrams is referred to as shifting to a higher state if the distance between the basal (`Turned off`) state node and the current state node increases as a result of a state change, or shifting to a lower state if the distance decreases. Correspondingly, a state is referred to as a higher or lower relative to a second state, depending on the difference in distance from the initial state.

The generalized implementation of the state machines is as follows: a variable `Status` of unsigned integer type is defined in `main.cpp` in order to hold state information. At the start of the process, this variable is assigned an initial value which changes when specific methods are called. Furthermore, a state machine object (SMO) is implemented. The SMO runs a `while` loop as long as the process is running and continuously executes commands depending on the value of `Status`. This design calls for the process to be multi-threaded so that the SMO can operate independently of the other threads. A class instance which operates within a separate thread will be referred to as a core object in this Bachelor's Degree Project. A pointer to the `Status` variable is passed to both core objects in their respective constructors.

2.1 Master

The Master can be found in one of the following states (ID represents the value of the `Status` variable for the given state):

1. **`Turned off`** (ID: N/A) – An abstract state which represents that the Master is turned off

2. **`Starting`** (ID: 21) – The Master is being initialized (starts threads and the DIM server and registers its services)
3. **`Waiting`** (ID: 1) – The Master is initialized and is ready to receive commands
4. **`Closing`** (ID: 22) – The Master is being turned off (ends threads)
5. **`Starting Slaves`** (ID: 23) – The Master is starting the slave processes (retrieves configuration from the database and runs a Python script which utilizes SSH in order to start the slave processes on remote computers)
6. **`Ready`** (ID: 2) – The Master has received confirmation that the slave processes are turned on and ready to receive commands from the Master
7. **`Turning Slaves off`** (ID: 24) - The slave processes are being turned off by the Master (using DIM commands)
8. **`Configuring`** (ID: 25) – The Master is retrieving configuration information from the database and configuring the slave processes (using DIM commands)
9. **`Configured`** (ID: 3) – The Master has received confirmation that the slaves are configured and ready for a run
10. **`Unconfiguring`** (ID: 26) – The Master is unconfiguring the slave processes
11. **`Starting run`** (ID:27) – The Master has sent the DIM command to initiate a run and is waiting for the slaves to enter the `Dry run` state
12. **`Stopping run`** (ID: 28) – The Master has sent the DIM command to stop the run and is waiting for the slaves to enter the `Configured` state
13. **`Dry run`** (ID: 11) – The Master has received confirmation that the slaves have entered the `Dry run` state and initiated the data taking process. This version of data taking is error-tolerant, and will not cease taking data when an error occurs.
14. **`Run`** (ID: 12) – The Master has received confirmation that the slaves have entered the `Run` state and initiated the data taking process. This version of data taking is not error-tolerant, and will cease taking data immediately when an error occurs.
15. **`Error`** (ID: 41-48) – Error 1-8

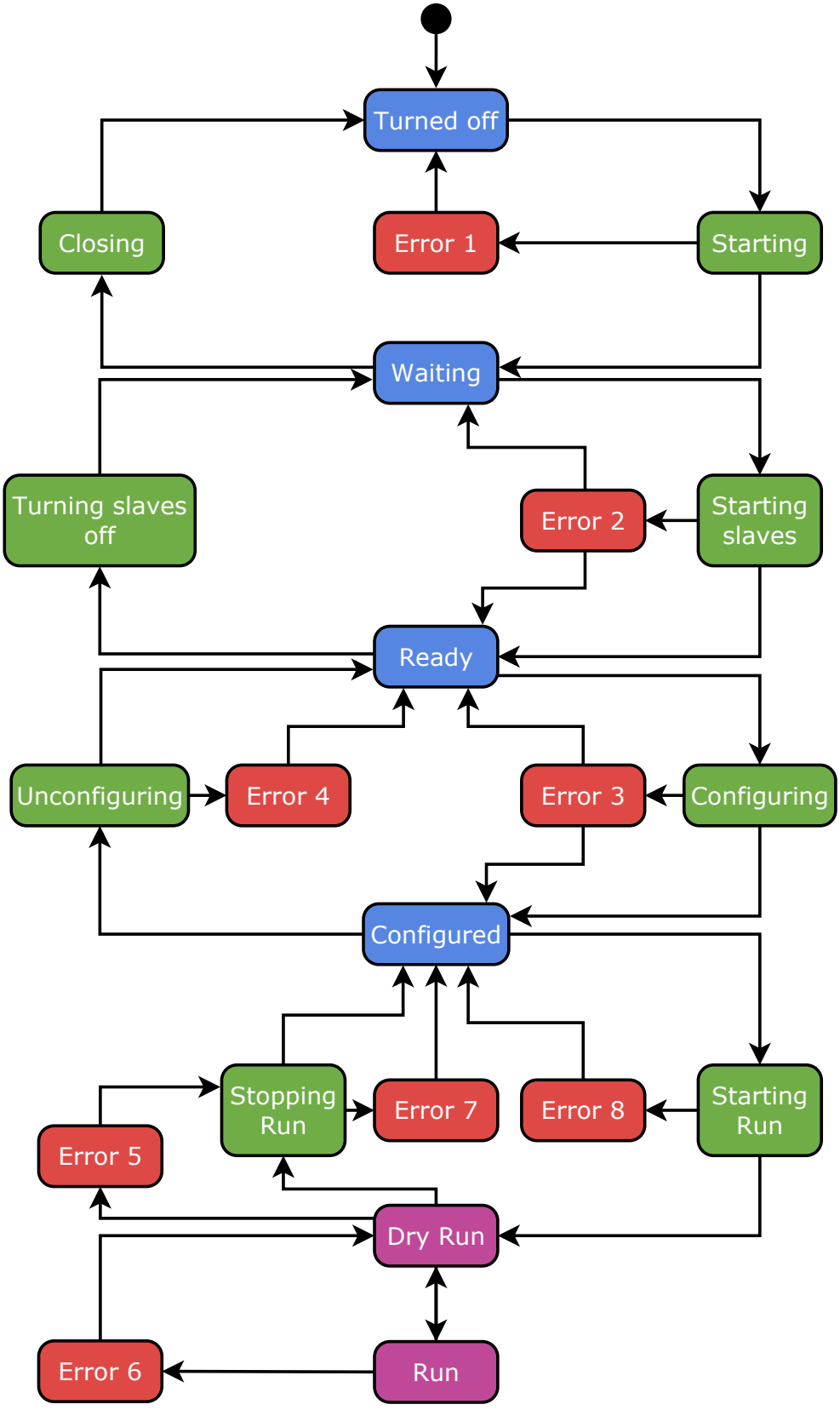


Figure 2.1: The master state machine diagram

2.2 Slave

The slave processes can be found in one of the following states:

1. **`Turned off`** (ID: 0) – An abstract state which represents that the slave is turned off (The ID for this state is only used by the Master)
2. **`Starting`** (ID: 21) – The slave is being initialized (starts threads and the DIM server)
3. **`Ready`** (ID: 2) – The slave is initialized and is ready to receive commands
4. **`Closing`** (ID: 22) – The slave is being turned off (ends threads)
5. **`Configuring`** (ID: 25) – The slave is being configured (loads settings and connects to appropriate devices)
6. **`Configured`** (ID: 3) – The slave is configured and is ready to receive commands
7. **`Unconfiguring`** (ID: 26) – The slave is being unconfigured (clears settings and disconnects from appropriate devices)
8. **`Starting run`** (ID: 27) – The slave is shifting into the `Dry run` state
9. **`Stopping run`** (ID: 28) – The slave is terminating the run (in the case of slave Readout, the slave remains in this state until all of the events have been processed)
10. **`Dry run`** (ID: 11) – The slave is partaking in the error-tolerant data taking process
11. **`Run`** (ID: 12) – The slave is partaking in the error-intolerant data taking process
12. **`Error`** (ID: 41-48) – Error 1-8

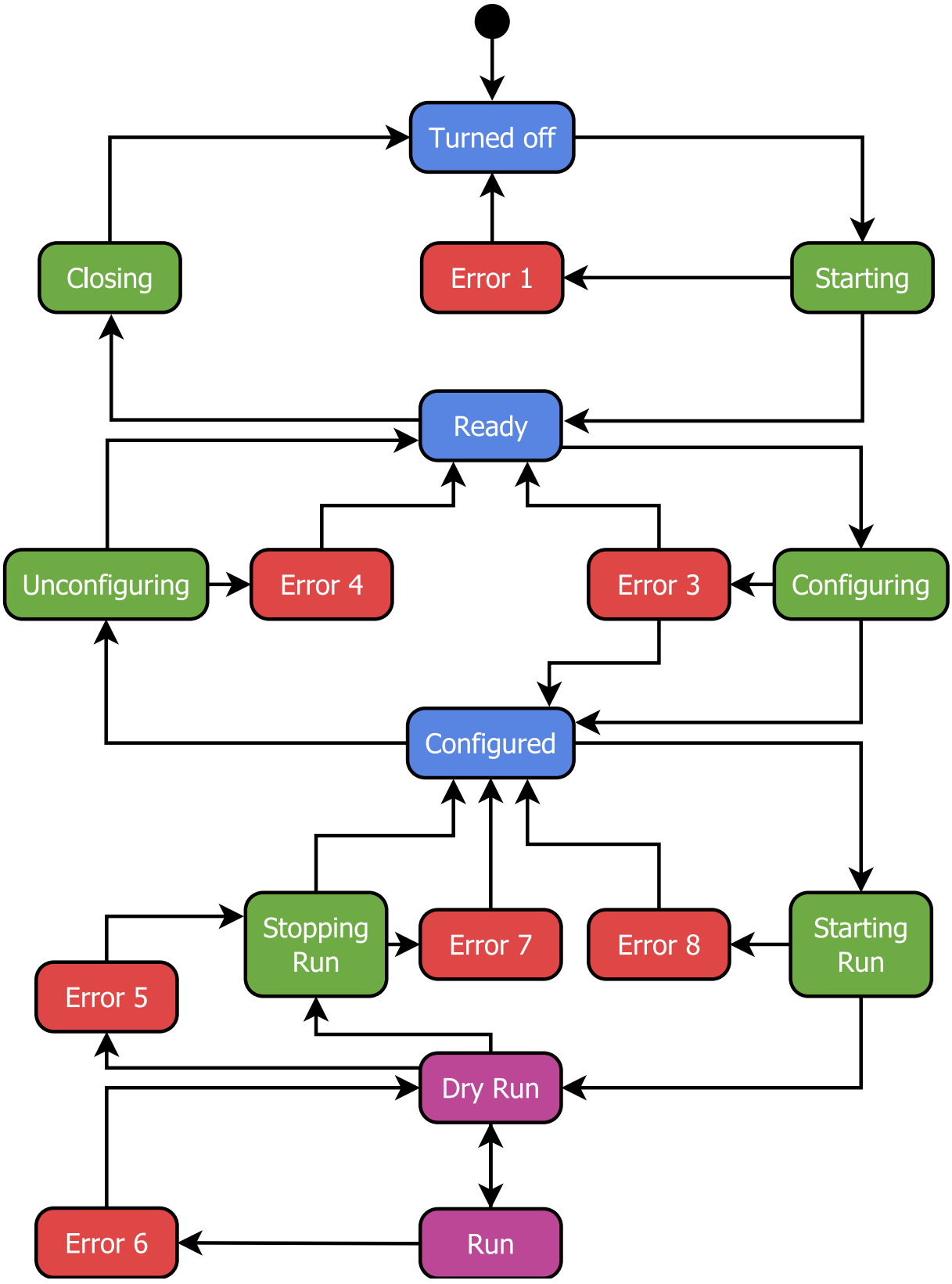


Figure 2.2: The slave state machine diagram

2.3 Classification of the states

1. **Stable state** - The process is on standby and awaiting a command (denoted by blue in Figures 2.1 and 2.2)
2. **Transfer state** - The process is working and will not respond nor accept any commands (denoted by green in Figures 2.1 and 2.2)
3. **Working state** - The process is working and can receive commands (denoted by purple in Figures 2.1 and 2.2)
4. **Error state** - The process has encountered an error and will not accept any commands until it transfers into a stable state (denoted by red in Figures 2.1 and 2.2)

2.4 Error states and their purpose within the state machines

Error states are the leading actors of the DAQ's visible error handling. By design, the basic principle of error states is such that they resemble transfer states (i.e., a given process will not accept any commands until it has left an error state) and attempt to resolve any possible error that could have occurred. Not a single possible erroneous situation should be left unhandled and it should be impossible for an error state to become unresponsive or require any input from the user. That way, error handling at the process level is fully automated and encapsulated.

Furthermore, every non-error state which can result in an error state should be the only state which can result in that specific error state (i.e., the non-error states should not share the error states they can result in) so that it is always clear in which state the error occurred.

Error states are unique in that they can result in one than more stable state. In particular, this is the case in the error states which are a result of the transfer states which indicate the process is shifting to a higher state. This branching differentiates in between the case where the error handling process has succeeded in repairing the error (results in a higher state) and the case where it has not succeeded (results in a lower state).

2.5 Implementation

2.5.1 Master

The Master comprises two core objects: the SMO and the DIM Communication Object (DCO). The SMO makes use of Qt signals and slots as a means of inter-thread communication, although solely to send and receive information, warning and error messages. The pivotal element of the SMO class is the slot `Run`, which contains the aforementioned `while` loop containing the `switch` statement. The `Status`

variable is taken as an argument of the `switch` statement, so that different actions are performed for each state. For the most part, this includes emitting regular status information messages, setting variables or in the case of the error states, emitting error messages. If the value of `Status` is that of a transfer state, in some cases it is changed to the next possible stable state during the first pass through the `switch` statement. In other cases, a simple check with respect to the states of the slaves is performed to determine whether the value should be changed to that of the next stable state.

A much greater role is played by the DCO, whose class contains the implementation of communication with the other parts of the DAQ through DIM (the Master acts both as a DIM server and a DIM client). The relationship of the `Status` variable and the DCO is such that the value of `Status` changes when DCO methods that are equivalent to transitioning from a stable state to a transfer state are called. Similarly to the SMO, the DCO contains a method named `Run`, which executes a loop as long as the process runs. The block of code enclosed by this loop handles composition of information messages which contain information concerning the slave processes and, for the purpose of relaying these messages, regular updates of the DIM service which the GUI subscribes to. The DIM communication received by the Master is stored in a queue, which is then also processed in the `run` method. A very brief description (within the scope of what is important for this Bachelor's Degree Project) of Master's DIM communication with the other processes can be found in Table 2.1.

Name:	Sender/publisher:	Receiver/ subscriber:	Type:	What is sent/published:
`INFO_SERVICE_[ID]`	Slave Control	Master	DIM service	State information
`INFO_SERVICE_[ID]`	Slave Readout	Master	DIM service	State information
`INFO_SERVICE_42`	Master	MessageLogger	DIM service	Regular status updates, warning messages, error messages, fatal error messages
`RUN_CONTROL_[ID]`	Master	Slave Control	DIM command	Commands to change state
`RUN_CONTROL_[ID]`	Master	Slave Readout	DIM command	Commands to change state

Table 2.1: Master's DIM communication with other processes. The `ID` represents a concrete slave.

File name:	Contents:
`Dimcommunicationobject`	The DCO class and its subclasses - the entirety of code concerning DIM communication with the other processes
`Handler`	Methods for creation and extension of XML connection files
`Main`	Core object constructor calls, declarations of global variables, threading code
`SlaveControl`	A class containing methods for setting and getting settings and state information of slaves of the Slave Control type
`SlaveReadout`	A class containing methods for setting and getting settings and state information of slaves of the Slave Readout type
`Statemachineobject`	The SMO class
`VirtualSlave`	A common polymorphic base class for the classes `SlaveControl` and `SlaveReadout`

Table 2.2: A listing of the Master's source files. Each file name (except for `Main`) represents two files, the header file and the cpp file.

Library name:	Description:	Path:
`Libdim`	The DIM library	/online/CMAD/dim64/linux/
`Transportprotocol`	The Transport protocol Library	/online/CMAD/TransportProtocol64/
`Database`	A library containing all of the database access methods	/online/CMAD/Database/
`Registerslib`	A library containing methods for management of FPGA and spillbuffer registers	/online/CMAD/cmad/RegistersLib/

Table 2.3: A listing of the Master's dependencies

2.5.2 Slave Control

The core object and thread design in Slave Control is identical to that of Master, although the terminology is different. While the SMO and its thread preserve their nomenclature, all dim communication with the Master is handled by the Informator Object (INF). A class called Ipbusfunctions which handles all of the communication with the FPGA cards is part of this process, and its methods are utilized both in the SMO and the INF.

2.5.3 Slave Readout

The Slave Readout process comprises four threads: SMO, INF, FIFO and Event Processor. The FIFO thread handles temporary storage of the events read out from the spillbuffer. The Event Processor thread is concerned mainly with calling caling an adjustable number of threads which handle conversion of the events into the DATE format and their storage on local hard disk drives. It also handles distribution of the events of monitoring tools. More information concerning the two latter entries can be found in [12]. Similarly to Master and Slave Control, all interactions with the `Status` variable are carried out inside the

SMO and INF threads. While the INF thread preserves the same purpose and implementation as the one of Slave Control, the SMO thread plays a much greater role than the ones previously mentioned. The SMO method `run` contains complete implementation of the management, configuration and readout of the connected readout devices as well signal-slot communication with the FIFO thread for the purpose of storing read-out events. This is done within the following cases of the `switch` statement: 11 (`Dry run`), 27 (`Starting run`), 28 (`Stopping run`). Several run end conditions are implemented in case 11, giving the SMO the ability to autonomously stop the dry run and shift to a lower state.

2.5.4 Implications of the state machine design

It is apparent that the above described design of the state machines calls for synchronization of the individual state machines of the master and slave processes. When stable or working, the Master should always be in a state identical to that of all the slaves. In the case of transfer and error states, the Master should not leave such states until every single slave has done the same. At the time of this analysis, the DAQ lacked automated tools to carry out this process and it was the main aim of this Bachelor's Degree Project to design and implement algorithms for synchronization of the DAQ's state machines. A general outline of the tasks these algorithms should be able to accomplish can be found below:

1. To check for and resolve state discrepancies of the slaves and the Master. It was commonplace that when a single slave process became unresponsive for any reason, died or merely entered a state different from the other slaves, the entire DAQ software had to be restarted just to re-establish synchronization of the states of the processes.

2. To ensure time-outs of the Master's transfer states which interact with the slaves. When a slave process became desynchronized, unresponsive or dead during a master transfer state, the master process became stuck in the transfer state and has to be restarted.

3. To eliminate duplicate slave processes created by restarts of the master process. The master process turns all the slaves on while it is in the `Starting slaves` state, and turns all of them off while it is in the `Turning slaves off` state. However, if the master process was terminated before it had a chance to go through the `Turning slaves off` state, the slave processes were never terminated and new ones were created the next time the master process was turned on and went through the `Starting slaves` state, creating duplicates and therefore causing process congestion of the computers running the slave process.

Chapter 3

Design of the error-handling algorithms

All of three tasks stated at end of the previous Chapter are closely intertwined. As can be found below, the algorithms for the Tasks 2 and 3 are merely an extension of the algorithm for the Task 1.

The main idea was to design a versatile master-side algorithm which will be easy and fast to adjust, should the structure of the DAQ's state machines be changed in the future. The algorithm will periodically perform consistency checks, i.e., examine whether the Master and all the slaves are in the identical state. Should a consistency check fail, the algorithm will force the Master to enter one of the appropriate error states and based on which state it has forced the Master to enter, it will attempt to perform an appropriate version of consistency restoration. If the consistency restoration fails, the Master will enter a lower state. Whether the consistency restoration has failed or not is determined by a time-out, i.e., a check is performed whether the algorithm has succeeded at setting all the slave processes to the desired state within an allocated time period (30 seconds) or not. A detailed description of all the cases and actions can be found in Table 3.1.

Consistency check failed in state:	Error state to enter:	Attempt to set all slaves to state:	If successful, enter state:	If unsuccessful, enter state:
`Run`	`Error 6`	`Configured`	`Dry Run`	`Dry Run`
`Dry Run`	`Error 5`	`Configured`	`Stopping Run`	`Stopping Run`
`Configured`	`Error 9`	`Configured`	`Configured`	Ready
`Ready`	`Error 10`	`Ready`	`Ready`	`Waiting`
`Waiting`	N/A	`Turned Off`	`Waiting`	`Waiting` (and kill slaves)

Table 3.1: Behavior of the consistency restoration algorithm

As the stable states `Ready` and `Configuring` previously lacked any error handling in this context, the only outgoing connections lead to transfer states. However, with the introduction of this algorithm, the need to add outgoing connections to error states arose. In order for the error states to be unambiguous (such that each error state can only be triggered by one unique non-error state), it was necessary to introduce two new error states into the state machine diagram to indicate that the Master has encountered a slave state consistency error during one of the aforementioned states and is in the process of resolving it. The

states mentioned are `Error 9` (ID: 49), `Error 10` (ID: 50) and their roles in the state machine diagram can be found in Figure 3.3.

It should be noted that the periodic consistency checks are performed only in stable and working states, as can be inferred from Table 3.1. The concept of the transfer states of the master state machine is such that the Master remains in the transfer state until all of the slaves have entered the desired state, allowing for slaves to exist in differing states by design, therefore rendering the idea of periodic consistency checks inapplicable in this case. Handling of slave inconsistencies in transfer states is realized through time-outs. Should the Master remain in a transfer state which interacts with the slaves for a period of time longer than the one allocated (10 seconds), meaning it is almost certain a state inconsistency exists, it will incur a time-out error and enter the appropriate error state, in which a consistency restoration will be carried out, or simply wait for all of the slave processes to enter the stable state in the case no actual functional inconsistencies exist.

The transfer state `Turning slaves off` had to receive similar treatment as the states `Ready` and `Configuring`, as it also lacked an outgoing connection to an error state. Therefore, a third new error state was introduced to the state machine diagram. This state's name is `Error 11` (ID: 51) and its role in the state machine diagram can be found in Figure 3.3. With this, the only non-error states that do not have outgoing connections to error states are states `Turned off`, `Closing` and `Waiting`. `Turned off` is an abstract state and undeniably cannot have any other outgoing connections than to the `Starting` state. The `Closing` state is a state which does need an error connection, as the only error which can occur in this state is that the Master will turn off – which is the desired effect of this state.

Should a consistency check fail during either the `Dry run` state or the `Run` state, it is necessary to immediately terminate the run, in order that the data are not spoiled by errors which fill up the data stream once a slave inconsistency occurs. Another reason for the immediate termination of the run is protection of hardware. In the event a slave of the Slave Readout type dies, buffers start to fill up, resulting in a cascade effect which could eventually lead to hardware damage.

Concerning Task 3, a simple process which kills all active slaves when the Master is going through the `Starting` and the `Starting slaves` states will be designed. This process is also going to be used in the consistency restoration algorithm to kill unresponsive slaves.

It could be argued that the `Waiting` state needs to have an outgoing connection to an error state in order to indicate it is solving the hypothetical inconsistency scenario where some slaves had already been started and need to be turned off, but it has been concluded that it is impossible for such a situation to occur during normal operation after the implementation of the error-handling algorithms (as all of the states that have outgoing connections to the `Waiting` state also solve this scenario). The only possible situation this scenario can occur in is when the user manually turns on one or more slaves while the Master is in the `Waiting` state. That should not happen, and therefore the additional error state was not introduced. Nevertheless, this scenario will still be solvable by the consistency restoration algorithm as a safety precaution.

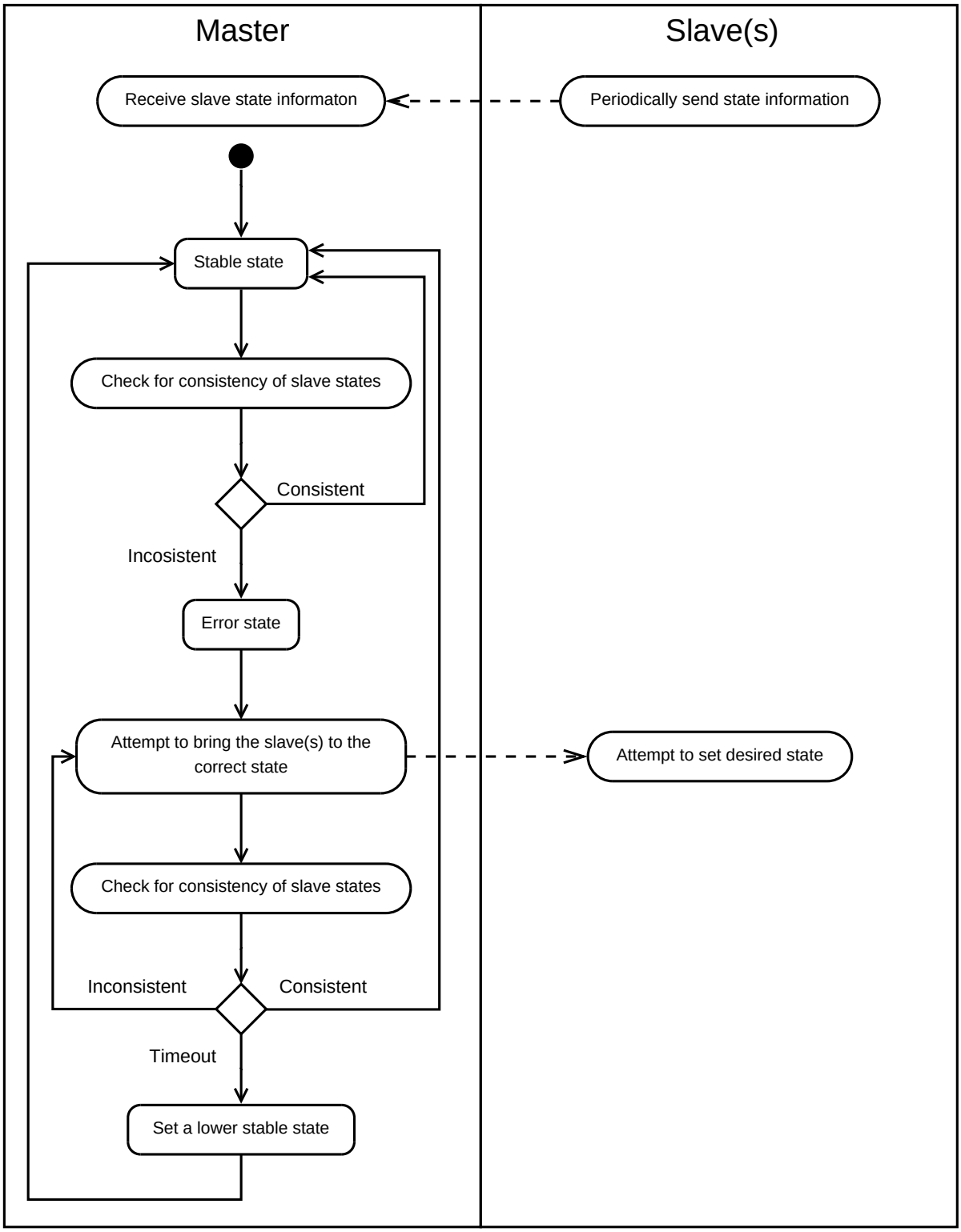


Figure 3.1: Handling of slave inconsistencies in Master's stable states

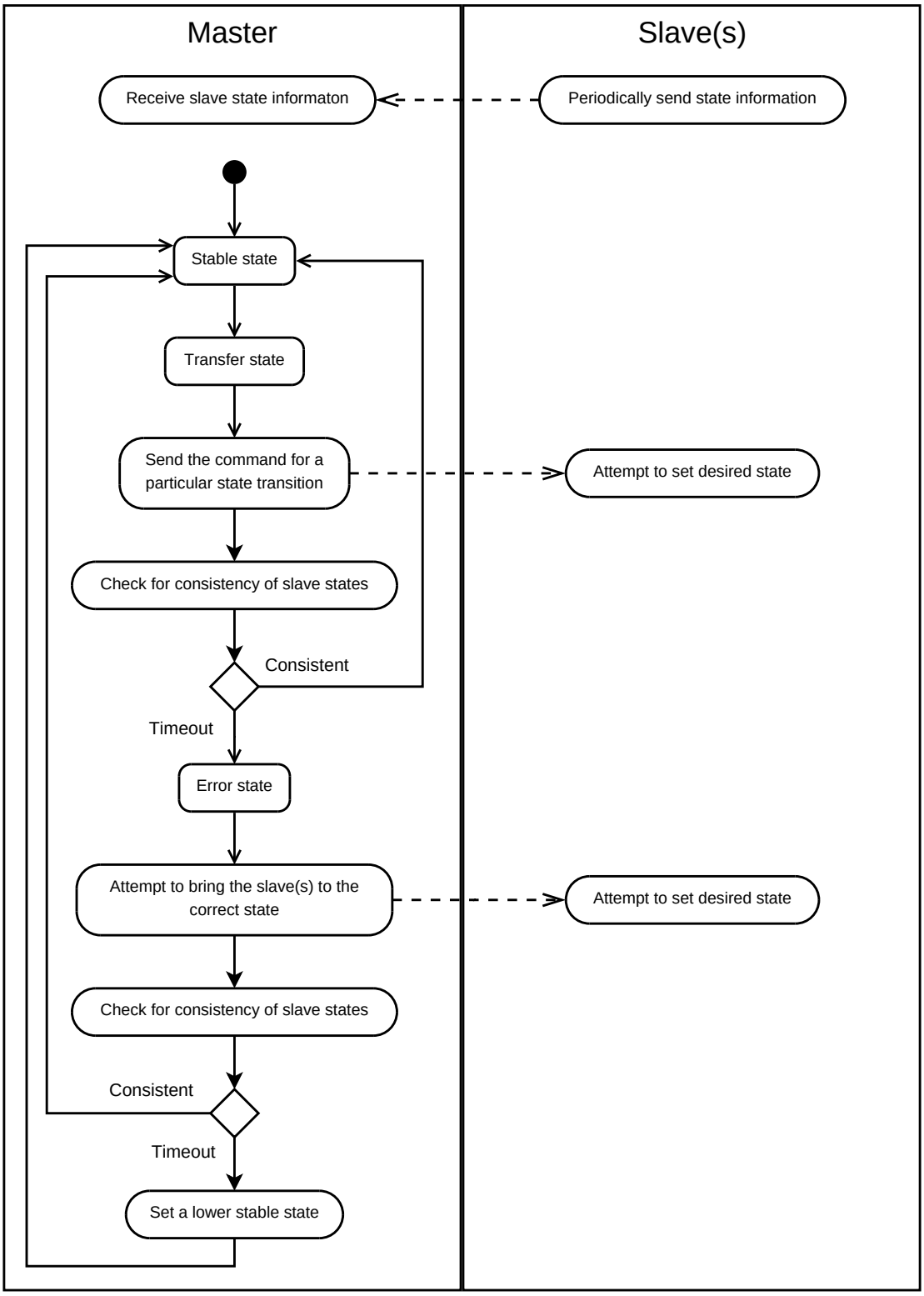


Figure 3.2: Handling of slave inconsistencies in Master's transfer states

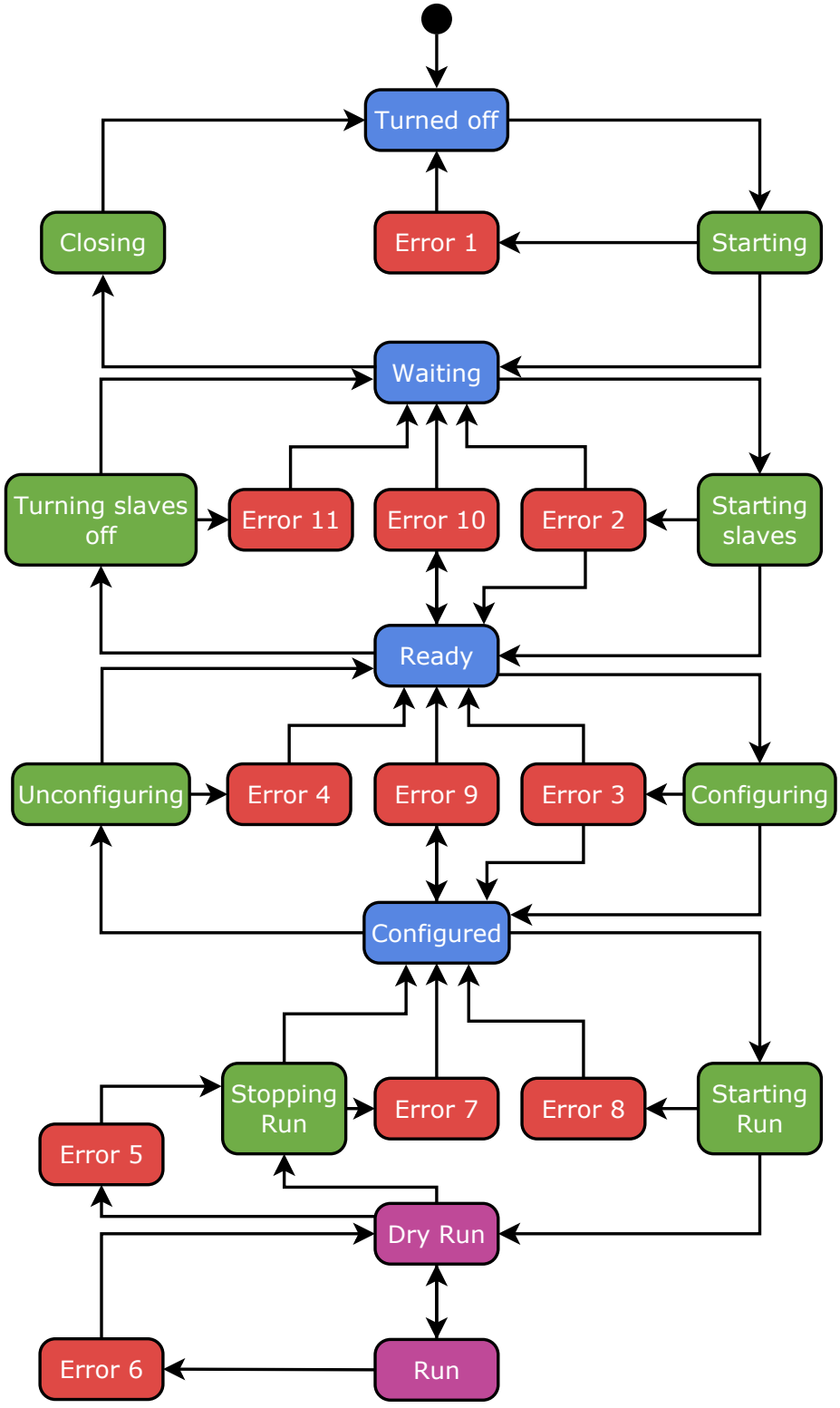


Figure 3.3: The new master state machine diagram

Chapter 4

Implementation of the error-handling algorithms

As the functionality provided by the code for Task 3 is also used in the code for Task 1, the implementation of the solution of Task 3 will be discussed first in this Chapter.

4.1 SlaveHunter process

The SlaveHunter is a very simple process used to kill (i.e., terminate) any given process running on the same machine. It accepts a single argument (the process name) and its primary purpose is to kill rogue slave processes described in Task 3. However, given its flexibility, other uses within the DAQ may be found for it in the future as the DAQ's development continues.

The SlaveHunter is implemented in C++, and it consists of a single class with two methods which are called in the main function. The first method, `Execute_cmd`, uses functions of the C Standard Input and Output Library to execute a system command and return the result in the QString type. In the main function, it is called exclusively with the `"ps -A aux"` argument. The second method, `Parse`, is then used to parse the string returned by the first method and find the process ID associated with the given process name.

Once the process ID is retrieved, the SlaveHunter uses the kill function of the C Standard Signal Library to kill the process. First, it attempts to do so using the Terminal interrupt signal (`SIGINT`), and after a second of sleep, it uses the Terminal kill signal (`SIGKILL`). As a safeguard, the program repeats the whole process (including calling the `Execute_cmd` and `Parse` methods) until it is positive that a process of the name passed as an argument to SlaveHunter does not exist.

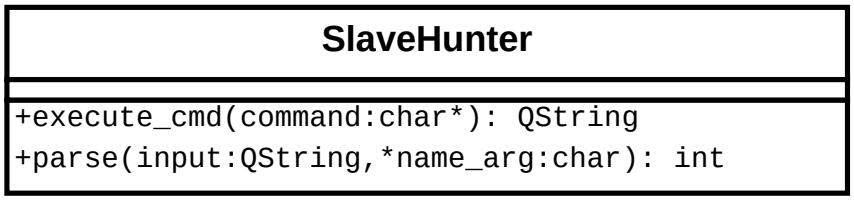


Figure 4.1: The `SlaveHunter` class diagram

4.2 IntegrityChecker class

The `IntegrityChecker` class is the class which implements the algorithm developed in Chapter 3. The class is situated within the Master and can be figuratively divided into three subsections which handle the aforementioned tasks. A new pair of files has been created within the Master to accommodate it – `integritychecker.cpp` and `integritychecker.h`. A description of its most important methods and how they interact follows. A complete listing of the methods can be found in Figure 4.2.

4.2.1 Handling of Task 3

This subsection consists of two methods, `Kill_slave_process` and `Kill_all_slaves`. The former accepts one argument, the slave index and is used to kill a slave process associated with that index. The latter accepts no arguments and kills all active slaves.

4.2.2 Handling of Task 1

The fundamental building block of the algorithm is the ability to induce a slave state shift to a state higher or lower than the previous one by one level. This behavior is implemented in the methods `Increase_slave_state` and `Decrease_slave_state`. These methods accept only one argument, the index of the slave whose state is to be changed, and return no value. Because of the given principle of the methods, as well as the fact the slave processes only accept commands which will change their state by only one level, it is necessary for the method to possess access to slave state information. This information is obtained from the global variable `Slaves`, which is an array of instances (each representing a single slave) of classes derived from the class `VirtualSlave` (either `SlaveControl` or `SlaveReadout`) that contain the state information in their public attribute State.

Another essential component of the algorithm is the ability to determine the directional relationship, i.e., whether the state needs to be increased or decreased in order to move from one node to another, between two given nodes of the slave state machine diagram; this task is handled by the `Direction_search` method. Two arguments of type unsigned int are accepted by this method, one defining the starting node (`From_state`) and one the final node (`To_state`). Error states are not accepted as the final node argument, as the notion of shifting to a higher or lower slave state is not consistent with the concept of transition

to an error state. While transfer states are accepted as the final node argument, this functionality is not used nor required in the current version of the algorithm and due to the cyclic nature of the state machine diagram, the result is only an approximation in some cases – a definition of additional return values would be required in order for this method to be completely accurate in this case. The method returns -1 if the direction corresponds to shifting to a lower state, 1 if the direction corresponds to shifting to a higher state and 0 if the two arguments are identical.

The method in which the previously mentioned methods come together is named `Case_error`. One argument of unsigned integer type is accepted by it and its purpose is to set all slaves to the state specified by the argument. A boolean value is returned, indicating whether the process has been successful or not.

The outernmost layer of the algorithm is made up by the methods `Restore_consistency_stm` and `Restore_consistency_update`. These methods are concerned with calling the the `Case_error` method with the appropriate argument and setting the `Status` variable. Both methods have no return value and accept a pointer to the `Status` variable as an argument. The `Restore_consistency_stm` method is called exclusively by the SMO when the Master enters an error state and the `Restore_consistency_update` method is called exclusively by the DCO (through a Qt signal, see Section 4.2.6) when a periodic consistency check fails during a stable state.

4.2.3 Handling of Task 2

This subsection consists of two simple methods, `Update_timeout` and `Is_timeout`. `Update_timeout` accepts the `Status` variable as an argument, returns no value and is used to keep track of how long the Master has been in a given state with the help of two private variables. It is called by the SMO in each iteration of the run loop, regardless of the state the Master is in.

`Is_timeout` is a method which returns a boolean value – true if the Master has not changed state for the number of milliseconds specified by its argument and false if otherwise. This method is called by the SMO in each iteration of the run loop as long as the Master is in a transfer state which deals with the slaves. If true is returned, an error message is emitted and the `Status` variable is changed to that of an appropriate error state. A `TIMEOUT` macro, which is used as the argument, is defined within the SMO.

The macro is set to 30 000 milliseconds, and is used in all transfer states except for `Starting run` and `Stopping run`, in which multiples of the macro are used. In the case of `Starting run`, the macro is multiplied by 2 – the reason for this is that when a run is being started, the Slave readout process remains in the `Starting run` state until it receives the first event of the run. As the events are received periodically (the period depends on the setup of the particle accelerator – presently up to 52 seconds), a time-out lower than the period would cause the first event to sometimes not be read out. In the case of stopping run, the macro is multiplied by 5 – one of the reasons for this is that data can pile up in the buffer and/or in the FIFO, and it is necessary to wait until they are saved on the hard drives. However, the main reason is that slaves of the Slave Readout type purposefully remain in the `Dry run` state even after the Master has entered the `Stopping run` state. This is because these slaves wait for the last event of the run before being allowed to transition into the `Stopping run` state. There is a slave-side time-out associated with this wait, because without it, the wait would be indefinite in the case the slaves miss the last event of the run. Naturally, the master-side time-out has to be long enough in order to allow for this process to happen.

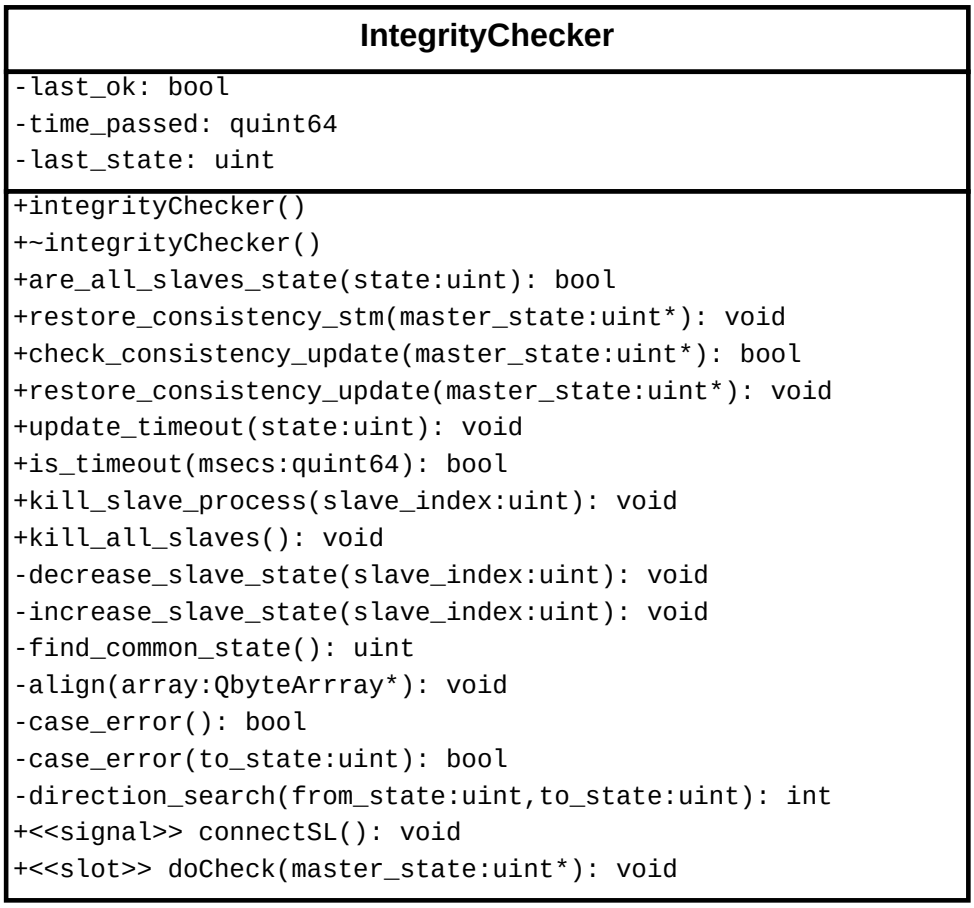


Figure 4.2: The `IntegrityChecker` class diagram

4.2.4 Detailed description of the methods

4.2.4.1 Kill_slave_process

This method uses the system function, which is part of the standard C++ library and is used to execute system commands. The command opens a Python script which connects to a remote host PC using SSH and starts an instance of SlaveHunter with the appropriate argument on it. The slave name and host PC name are accepted as arguments by the script. This information, along with the address of the Python script, is retrieved from attributes of the Slaves variable. The syntax of the system command is as follows: [Python Script Address] [Slave name] [Host pc Name].

```
import subprocess
import sys

slavename = sys.argv[1]
pcname = sys.argv[2]

subprocess.Popen(["ssh", pcname, "/online/CMAD/cmad/SlaveHunter/trunk/SlaveHunter",
    slavename])
```

Listing 4.1: The Python script used by the `Kill_slave_process method`

4.2.4.2 Increase_slave_state

In the case the slave is off and needs to be turned on, this method uses the system function and a python script in order to start a slave process on a host PC in the same manner the `Kill_slave_process` method starts an instance of SlaveHunter. The slave name, slave ID and host PC name are accepted as arguments by the script (slave processes take slave name and slave ID as a starting argument). The syntax of the system command is as follows: [Python script address] [Slave name] [Slave ID] [Host PC name]. In the case the slave is on and only needs to shift to a higher state, a command containing a message encapsulated in the transport protocol format is sent to the `RUN_CONTROL_[Slave ID]` DIM command. The body of the message is made up of four characters that specify which state to enter. In the case this method is called while the slave is in a transfer or error state, the main thread sleeps for one second.

4.2.4.3 Decrease_slave_state

This method is analogous to the `Increase_slave_state` method, with the exception that there is no need for the usage of a system command or a Python script. All interactions with the slaves are handled solely through DIM.

Slave state:	Increase state message :	Decrease state message :
`Ready`	`CONF`	`STOP`
`Configured`	`DRUN`	`UNCO`
`Dry run`	`SRUN`	`CONF`
`Run`	N/A	`DRUN`

Table 4.1: A listing of commands which can be sent to the `RUN_CONTROL_[Slave ID]` DIM command and their effects

4.2.4.4 Direction_search

This method contains two `switch` statements enclosed in `while` loops to iterate on a local variable of unsigned integer type called `Temp`, which is assigned the value of the initial state at the beginning of the method. The idea is to simulate a traversal of the state machine diagram, starting at the initial state node. The first loop traverses the slave state machine diagram upwards by assigning the temp variable the next

lower state while respecting directions of the edges of the state machine diagram. If the final node is encountered during the simulation, -1 is returned.

In the cases the initial node is a transfer state whose only possible next direction is downwards, the simulated traversal upwards first has to perform one shift downwards. If the final node is reached in this step, 1 is returned.

The second loop is an extension of the method and traverses the slave state machine diagram downwards in a similar fashion as the first loop, implementing the basis for the method's functionality when a transfer state is received as the final node argument. Should this functionality not be needed in the future, the entire loop can be replaced by a `"return 1;"` statement and the method will continue functioning normally for stable and working states. As this method is called relatively often in the consistency restoration algorithm, it has been decided that no error handling of its calls with incorrect arguments will be implemented. Therefore, the user of this method should pay increased attention when calling it, otherwise the process might break down.

4.2.4.5 Case_error

This method contains a `while` loop which repeats until all of the slaves are the state received in the argument, or until 30 seconds have passed. Within it, for every active slave, the `Direction_search` method is called. The value passed to the `From_state` argument is retrieved from the `Slaves` variable, and the value passed to the `To_state` argument is given by the value of the argument initially passed to the `Case_error` method. If 1 is returned, the `Increase_slave_state` method is called for the given slave. Similarly, if -1 is returned, the `Decrease_slave_state` method is called. If 0 is returned, neither of these methods is called. After this has been done for every slave, the signal `ConnectSL` (see Section 4.2.6) is emitted.

4.2.4.6 Restore_consistency_stm

A description of the actions this method performs in relation to which error state the Master is in can be found in Table 4.2.

Error state:	Actions to perform:
`Error 1`	None
`Error 2`	Call `Case_error(2)` and set `Status` to 2 if true is returned. If false is returned, call `Case_error(0)`. If false is returned again, call `Kill_all_slaves`. Afterwards, regardless of the value returned by the second call of `Case_error`, set `Status` to 1.
`Error 3`	Call `Case_error(3)` and set `Status` to 3 if true is returned. If false is returned, set `Status` to 2.
`Error 4`	Call `Case_error(2)`. Afterwards, set `Status` to 2 regardless of the returned value.
`Error 5`	Call `Case_error(3)`. Afterwards, set `Status` to 28 regardless of the returned value.
`Error 6`	Call `Case_error(3)`. Afterwards, set `Status` to 11 regardless of the returned value.
`Error 7`	Call `Case_error(3)`. Afterwards, set `Status` to 3 regardless of the returned value.
`Error 8`	Call `Case_error(3)`. Afterwards, set `Status` to 3 regardless of the returned value.
`Error 9`	Call `Case_error(3)` and set `Status` to 3 if true is returned. If false is returned, set `Status` to 2.
`Error 10`	Call `Case_error(2)` and set `Status` to 2 if true is returned. If false is returned, call `Case_error(0)`. If false is returned again, call `Kill_all_slaves`. Afterwards, regardless of the value returned by the second call of `Case_error`, set `Status` to 1.
`Error 11`	Call `Case_error(0)`. If false is returned, call `Kill_all_slaves`. Afterwards, regardless of the value returned, set `Status` to 1.

Table 4.2: Actions performed by the `Restore_consistency_stm` method upon entry into given error states

4.2.4.7 Restore_consistency_update

This method is concerned with setting the `Status` variable – its purpose is to make the Master enter the appropriate error state, in which the `Restore_consistency_stm` method will take over. It also includes error handling for the `Waiting` state (explained in Chapter 3).

Master state:	Actions to perform:
`Waiting`	Call `Case_error(0)`.
`Ready`	Set `Status` to 50.
`Configured`	Set `Status` to 49.
`Dry run`	Set `Status` to 45.
`Run`	Set `Status` to 46.

Table 4.3: Actions performed by the `Restore_consistency_update` method in given states when a consistency check fails

4.2.5 Additional functionalities of the IntegrityChecker class

This section describes the methods of the `IntegrityChecker` class whose usage is nearly or completely outside of the scope of the consistency restoration algorithm.

4.2.5.1 Find_common_state

This method is used to find the so-called "Lowest common state" (LCS) of the slaves, i.e., the closest stable state of the slave(s) with the lowest state (applicable regardless of whether the states of the slave processes are synchronized or not). This only applies to slaves which are turned on, which means the slave processes which are in a state which results explicitly in the `Turned off` state or in the `Turned off` state itself are ignored and not considered as being in the lowest state. The value returned by this method is the ID of the LCS. In the case of no slaves being connected or all slave processes being dead, 1000 or 999 are returned, respectively.

It is necessary to state that there are two unique cases of indeterminism when calling this method. This only occurs in extremely rare cases, i.e., when one or more of the slave processes are in the process of resolving `Error 3` or `Error 6`. Because these error states can result in more than one state, it is impossible to determine the resulting stable state without waiting for the slave error states to resolve. However, waiting for the slave error states to resolve is inconsistent with the intended use of this method, and therefore this method presumes that `Error 3` will always resolve in the `Ready` state and `Error 6` will always resolve in the `Dry Run` state.

Currently, this method only finds use in the `Case_error` method to check if any slaves are connected and in the method described below this one, but it may find additional uses in the future.

4.2.5.2 Case_error (Overloaded version)

This version of the `Case_error` method is used when no arguments are passed to it and attempts to set all slaves to the LCS in a similar fashion as the original version of the method. This method was made on request in the early stages of development of the consistency restoration algorithm and remains unused after a change of plans. It is nevertheless available, should a need for it arise in the future, however unlikely it may be.

4.2.5.3 Create_EM

This method is used to compose error messages which are then sent to MessageLogger using the DIM service `INFO_SERVICE_42`. The error messages contain information concerning the master state as well as the names and states of all connected slaves.

This method is called from the SMO every time the Master enters an error state which deals with the slaves (except for `Error 2`), but has been designed to be flexible and can be called from anywhere else in the Master process, provided an instance of the `IntegrityChecker` class is available.

4.2.6 Placement of the IntegrityChecker class within the Master's threading hierarchy

A single instance of `IntegrityChecker` has to be able to communicate with both the DCO and the SMO. The instance resides in the main thread and the following slot and signal are used within the `IntegrityChecker` class for the purpose of communication.

4.2.6.1 Slot DoCheck

This slot is connected to the `CheckNeeded` signal of the DCO. The only action performed within this slot is the calling of the `Restore_consistency_update` method. This is how DCO calls the method.

4.2.6.2 Signal ConnectSl

This signal is connected to the `reConnectToSlaves` slot of the DCO. The `reConnectToSlaves` slot handles Master's re-subscription to all DIM services of slaves which had to be restarted. This signal is emitted within the `Case_error` method after increasing a slave's state.

When the Master is being started, a single instance of `IntegrityChecker` is created within the main thread. Afterwards, when the DCO and SMO instances are being created, but not yet being divided to separate threads, a pointer to the `IntegrityChecker` instance is passed to their respective constructors and further stored within both of the objects. The `DoCheck` slot and the `CheckNeeded` signal are then connected within DCO's constructor. Finally, the `ConnectSL` signal and the `reConnectToSlaves` slot are connected, and both the DCO and the SMO are detached from the main thread.

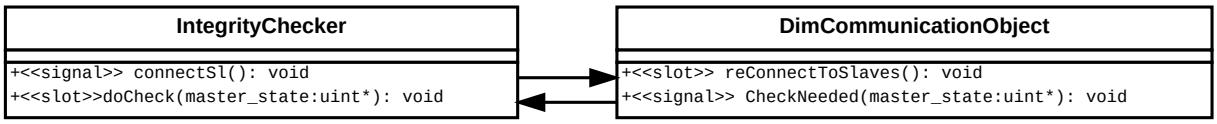


Figure 4.3: Signal-slot interaction of `IntegrityChecker` and the DCO

Chapter 5

Examples

This Chapter lists a few actual cases of the consistency restoration algorithm at work during several scenarios in order to present a comprehensive overview of the new error handling abilities of the DAQ to the reader. All of the scenarios listed here have occurred in practice and unfolded in the expected manner as described in this Chapter.

5.1 The temporarily unresponsive slave

In this scenario, the Master had sent the commands to start the slaves and is currently in the `Starting slaves` state. However, for an unknown reason, one of the slave host computers has failed to receive the command and did not start the slave process. Because one of the slave processes is not in the `Ready` state, the Master cannot exit the `Starting slaves` state and times out after 10 seconds. Following the time-out, the Master enters the `Error 2` state and attempts to start the dead slave. This time around, it is successful and enters the `Ready` state as soon as the slave has done the same.

5.2 The crash

In this scenario, the Master and the slaves are all stable in the `Configured` state, when one of the slaves suddenly crashes for an unknown reason. After detecting this state inconsistency, the Master enters the `Error 9` state, and proceeds with the following:

- Sending the command to turn on the slave and waiting until it has entered the `Ready` state.
- Sending the command to configure the slave.

Once the slave has entered the `Configured` state, the Master leaves the `Error 9` state and returns to the `Configured` state, becoming stable once again.

5.3 The fallthrough

This scenario is the most severe error situation of the examples given here. During a data taking run, one of the slave host computers has broken down (or has inadvertently been turned off by someone). Therefore, the connection to one of the slaves has been lost, and the slave will remain in the abstract `Turned off` state until the computer is replaced or turned back on again (i.e., for the whole duration of this scenario).

The Master and all of the unaffected (surviving) slaves are in the `Run` state. When the Master detects the state inconsistency, it enters the `Error 6` state and sends the command to stop the run to the surviving slaves, which then enter the `Dry run` state. Afterwards, it sends the command to stop the dry run, and the surviving slaves enter the `Stopping run` state. During this, it continuously keeps trying to turn the dead slave on, but fails. After 30 seconds in the `Error 6` state, the Master times out, enters the `Dry run` state, and because not only the inconsistency still exists, but also because the surviving slaves have already entered the `Stopping run` state, it enters the `Error 5` state, attempts to restore consistency for another 30 seconds, and promptly enters the `Stopping run` state. It remains in the `Stopping run` state for 150 seconds before timing out and entering the `Error 7` state. While in the `Error 7` state, it once again tries to revive the dead slave and times out after 30 seconds.

By now, all of the surviving slaves have almost certainly entered the `Configured` state, and after the last time-out, so did the Master. Due to the continued existence of the inconsistency, the Master enters the `Error 9` state, once again fails to restore the consistency and enters the `Ready` state.

While in the `Ready` state, the Master immediately detects the inconsistency: one of the slave processes is dead, and the surviving slaves still remain in the `Configured` state. It enters the `Error 10` state and sends the command for the surviving slaves to enter the `Ready` state, as well as the command for the dead slave to be turned on. While the former is successful and the slaves go through the `Unconfiguring` state into the `Ready` state, the latter is not and the first part of the error solving process times out after 30 seconds. During the second part of the error solving process, the Master sends the command for the surviving slaves to turn off. It is promptly successful, enters the `Waiting` state and finally becomes stable as all of the slaves are turned off. This scenario takes approximately 4 minutes and 30 seconds to resolve and its course is depicted in Figure 5.1.

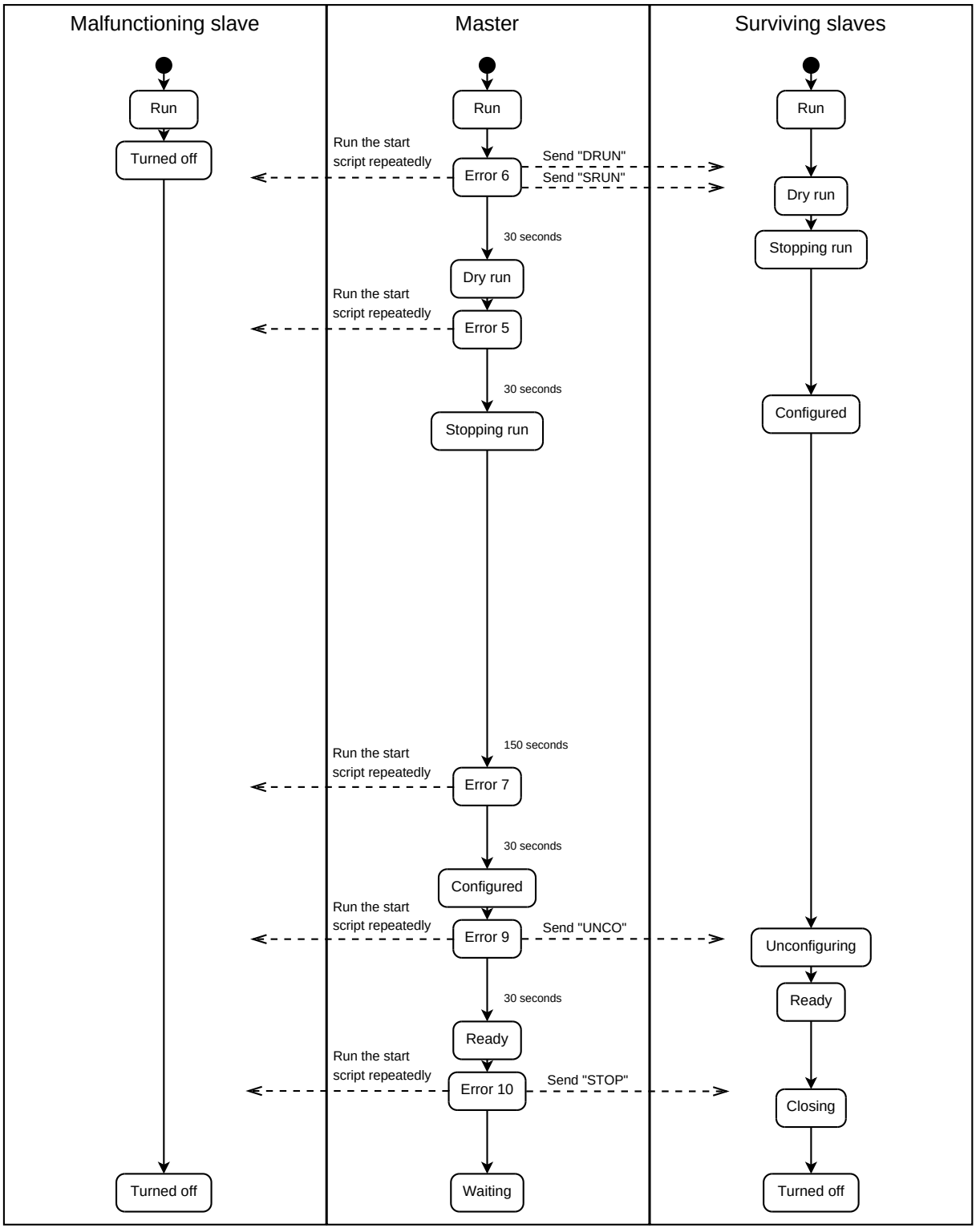


Figure 5.1: Illustration of the third example

Conclusion

The error-handling algorithms described in this Bachelor's Degree Project have been successfully implemented, tested and deployed as a part of the new COMPASS DAQ for the 2014 and 2015 Drell-Yan data-taking runs. Being a part of the master process which operates 24 hours a day, 7 days a week during a data-taking run, the algorithms find frequent use within the DAQ. The data taking started in August 2014, and so far, the algorithms have been working as expected and no defects have been reported.

These algorithms undoubtedly play an essential role within the DAQ, as without their functionality the DAQ software could be considered as unfinished – one of the most crucial aspects of the DAQ is user-friendliness and the automated error handling brought by these algorithms allows for a large amount of responsibility to be taken over by the system. The DAQ is ever-expanding and it can be assessed with an almost certain level of probability that changes to it will take place in the future. There have already been discussions concerning possible modifications of the state machines. This is where the adaptability of the algorithms will come into play – they can be adjusted to account for such changes.

Bibliography

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

[2] Ing. Josef Nový: **Processing of large quantity of data from the COMPASS experiment**; Written dissertation preparation, Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University, In preparation for release

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

[4] P. Abbon et al. (The Compass Collaboration): **The COMPASS experiment at CERN**, CERN-PH-EP/2007-001, January 2007

[5] **COMPASS: COmmon Muon Proton Apparatus for Structure and Spectroscopy** [online] Available at: http://wwwcompass.cern.ch [Accessed 3 February 2015]

[6] V. Yu. Alexakhin et al. (The Compass Collaboration): **COMPASS-II Proposal**, CERN-SPSC-2010-014; SPSC-P-340, May 2010

[7] **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]

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

[9] **iMUX/HGESICA module** [online] Available at: http://wwwcompass.cern.ch/twiki/pub/Detectors/FrontEndElectronics/imux_manual.pdf [Accessed 10 February 2015]

[10] **Electronic developments for COMPASS at Freiburg** [online] Available at: http://hpfr02.physik.uni-freiburg.de/projects/compass/electronics/catch.html [Accessed 10 February 2015]

[11] **The GANDALF Module** [online] Available at: http://wwwhad.physik.uni-freiburg.de/gandalf/pages/hardware/the-gandalf-module.php?lang=EN [Accessed 10 February 2015].

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

[13] CASTOR - **CERN Advanced Storage manager** [online] Available at : http://castor.web.cern.ch [Accessed 10 June 2015]

[14] **Qt developer network** [online] Available at: http://www.qt.io/developers/ [Accessed 20 April 2015]

[15] 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)

[16] **Distributed Information Management System** [online] Available at: http://dim.web.cern.ch/dim/ [Accessed 10 April 2015]

[17] **The IPbus Protocol** [online] Available at: http://ohm.bu.edu/ chill90/ipbus/ipbus_protocol_v2_0.pdf [Accessed 4 June 2015]

[18] **Linux at CERN** [online] Available at: http://linux.web.cern.ch/linux/scientific6/ [Accessed 4 June 2015]

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

Appendix A

CD contents

- **BP.pdf** : This Bachelor's Degree Project in PDF
- **Master** : A folder containing the Master's source files, including the IntegrityChecker class
- **SlaveHunter** : A folder containing the SlaveHunter's source files
- **Transport Protocol** : A folder containing source files of the Transport Protocol library