The CMS Pixel Luminosity Telescope Browser Interface

A Thesis Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Mark Miller Foerster
May 2015

Copyright © 2015 by Mark Miller Foerster
All rights reserved.

DEDICATION

To Dad, who has supported me through this entire adventure.

To investigation:
The process by which humans raise themselves up from their natural state of poverty to a state of greater wealth simply by learning how to solve their everyday problems.

ACKNOWLEDGEMENTS

My predecessor on the software of this project, Matt Hollingsworth, was responsible for all the heavy lifting involved in choosing a framework for web programming and the Python interfacing for it to the C++ code that does the real work with the PLT. He started this whole thing. I have been faithfully continuing its development.

I would like to thank my advisor, Dr. Stefan Spanier, for allowing me to participate in this part of the CMS collaboration.

I would also like to thank all the helpful people at CERN, UT (past and present), and our fellow collaborating institutions: Bob Stone, Will Johns, Dean Hidas, Andres Delannoy, Karen Andeen, Dmitry Hits, Alan James Bell, David Stickland, Anne Dabrowski, Elena Castro, Zongchang Yang, and Grant Riley.

Special thanks also to Danek Kotlinski for relaying a very helpful paper about the development of the pixel tracker's read out chip, used by the PLT.

Thank God for Particle Physics!

ABSTRACT

The search for and detailed study of new particles and forces with the Compact Muon Solenoid (CMS) detector at the Large Hadron Collider (LHC) of CERN is fundamentally dependent on the precise measurement of the rate at which proton-proton collisions produce any particles, the so-called luminosity. For the discovery of the Higgs candidate in 2012 the relative precision of this quantity was about 2.5%. To be able to observe deviations from Standard Model predictions for decay rates of this particle and others which can hint to new phenomena the targeted luminosity uncertainty is about 1%. Therefore, a new device, the pixel luminosity telescope has been built that counts charged collision products close to the LHC beam in real time. The development of the software interface dedicated to control its operation, monitor its activity, and store calibration and configuration information together with data into a database is the subject of this work. The interface was constrained by the need to be suitable for operation by expert as well as non-expert data taking personnel. In addition to operation at the detector level, the interface must also integrate into a framework that simultaneously collects luminosity information from other sub-detectors of CMS. Finally, it had to be built from common and well-tested software tools for reliability and the prospect of long-term availability.

TABLE OF CONTENTS

INTRODUCTION 1
The Standard Model of Particle Physics 1
The Importance of Knowing Luminosity 4
Measuring Luminosity 5
The Experimental Apparatus 6
Overview 6
The Large Hadron Collider 9
Particle Detectors at the LHC 12
The CMS Detector 13
The Silicon Tracker 14
The Electromagnetic Calorimeter 18
The Hadronic Calorimeter 21
The Muon Spectrometer 22
The Event Trigger 23
Level-1 Trigger 24
High-Level Trigger 24
Obtaining the Absolute Luminosity 24
Luminosity Calibration with Z-Boson Decays 25
Luminosity Calibration with Van der Meer Scans 27
References 31
CHAPTER I A System Dedicated to Measuring Luminosity: PLT 35
Abstract 36
Introduction 36
An additional benefit: bunch monitoring 38
PLT Implementation 39
Overview 39
The Sensors 40
Detector Readout 44
References 48
CHAPTER II Usage and Operation of the CMS PLT Browser Interface 49
Abstract 50
The Interface 50
Breakdown of Functionality 53
Data Collection 53
Running Calibrations 55
Job Work Loop Thread 62
Trigger Setup 64
Decoupled Data Run Monitoring 67
Managing Configuration Data (“Global Runs” and sub-structure) 69
References 75
CHAPTER III CMS PLT Browser Interface Database Structure 76
Abstract 77

Database Establishment 77
Data Structure Modeled by the Database 77
Storage of Bulk Data 80
References 85
CHAPTER IV Installing and building the browser interface 86
Abstract 87
Introduction 87
Installing and setting up the PLT Browser Interface on an SLC5/SLC6 system 87
The Additional Software 90
Supplementary: Chronic vs. One-Off Procedures 96
Supplement II: PLT Strip Chart Monitoring Software 99
CMSVol Repository 100
Building 101
Starting Up 104
Development and Testing Issues 105
Shutting Down 106
Further Information 107
References 108
CHAPTER V Navigating the Software for the PLT Browser Interface 109
Abstract 110
Introduction 110
The Files: What's where and how things are organized 111
The Connections: Chasing how things work from beginning to end 119
Connection to C++: Navigating Through the Python-to-C++ Interface 121
Organization of the C++ code 122
Extra: Implementation of the List of Available Calibrations for New Calibration Sets 124
Decoupled Data Run Monitoring 128
Trigger Setup Control 129
The Rest of the Interface Branch 131
Further Information 132
References 133
CHAPTER VI A Run-through of Using the Interface in Offline/Test Stand Mode 134
Abstract 135
Introduction 135
Creating and Running Data Taking Jobs 136
Creating and Running Calibration Jobs 157
Creating New Global Runs 169
References 176
CONCLUSION 177
VITA 178

LIST OF TABLES

Table 1: Record structure of Global_Run table 82
Table 2: Record structure of Configuration table 83
Table 3: Record structure of Calibration table 83
Table 4: Record structure of Calibrationset table 84
Table 5: Record structure of Calibrationassociation table 84
Table 6: Record structure of Datarun table 84
Table 7: Record structure for all calibration subclass tables 84
Table 8: C++ source directories and contents 123
Table 9: Calibration interface correspondences 1 of 4: checkbox names to value attributes 125
Table 10: Calibration interface correspondences 2 of 4: value attributes to model calibration classes 125
Table 11: Calibration interface correspondences 3 of 4: model calibration classes to class_name properties 126
Table 12: Calibration interface correspondences 4 of 4: class_name properties to C++ classes 126
Table 13: Calibration interface net correspondences: checkbox names to C++ classes 126

LIST OF FIGURES

Figure 1: The Fundamental Particles of the Standard Model 1
Figure 2: The CERN accelerator complex. Several machines are used to pre-accelerate the protons before their injection into the LHC 7
Figure 3: Beam pipe within the LHC tunnel, further enclosed by magnets and cryogenic cooling vessels (blue) 8
Figure 4: Damage caused by various 450 GeV beam intensities. The largest hole corresponds to being hit by 67 of the LHC's particle bunches (2808 is a full beam), and merely at the LHC injection energy (Schmidt, et al., 2006) 8
Figure 5: LHC's integrated luminosity (year-by-year comparison) seen at the CMS experiment 9
Figure 6: Comparison of various process production cross sections (Stirling, 2012). Discontinuities are due to switchover from computing collisions of protons vs. anti-protons to collisions of protons vs. protons at higher energies 10
Figure 7: Bunch structure at the LHC. Each bunch is 9 cm long 12
Figure 8: An overall view of the CMS detector complex. The detector is symmetrical around the beam line and designed to be hermetic 14
Figure 9: Transverse slice of the CMS detector with the tracks of an electron, a charged hadron (e.g. a pion), a neutral hadron (e.g. a neutron), a muon, and a cluster created by a photon. The electron and the photon deposit their entire energy in the electromagnetic calorimeter, generating an electromagnetic shower. A charged or neutral (no track) hadron reaches the hadron calorimeter where it is stopped. Only muons are able to traverse the entire detector. Their momentum is measured by the tracker and the muon system. 15
Figure 10: Radiation length thickness of the tracker vs. pseudorapidity (Caminada, 2012) 16
Figure 11: The silicon pixel detector (Hilbert, 2013) 17
Figure 12: 3-D and cross section views of the entire tracker, showing both layers of silicon strip detectors. Top plot: (Schael). Bottom plot: (Lenzi, 2013) 18
Figure 13: Schematic of the barrel and the end-caps of the CMS ECAL 19
Figure 14: Cross section of hadron calorimeter. Hadron Forward section not shown (Caminada, 2012) 22
Figure 15: Layout of one quarter of the CMS experiment. The Muon Barrel system extends r > 350 cm and |z| < 700 cm (chambers are shown in green color). In the endcap part CSC chambers (noted as ME) are clearly visible (blue color). RPCs are displayed in red color in the barrel and end-cap parts. 23
Figure 16: Schematic representation of the CMS L1 trigger system (Caminada, 2012) 25
Figure 17: The di-muon mass distributions for Z boson candidate events – the red lines indicate the selection interval. The variable χ shown in the lower

plot is defined as $(\boldsymbol{N} - \boldsymbol{N_{exp}})/\sqrt{\boldsymbol{N}}$, where N is the number of measured events and N_{exp} is the sum of signal and background events. 26
Figure 18: VdM scan seen by the PLT's diamond telescope. Data between the scan curves is due to beam optics set to nominal for collisions between scan iterations. The initial exponential declines in the latter is due to reduced charge collection via charge trapping in radiation-induced defects. 29
Figure 19: Location of the PLT inside the CMS detector (Atramentov, 2010) 37
Figure 20: Each half of the PLT, an octagon of tri-plane telescopes around the beam pipe 39
Figure 21: One PLT telescope, containing three planes of sensors mounted on read-out chips (ROCs). This is an earlier version of a telescope using diamond as the detector material, which does not completely cover the ROC. 40
Figure 22: Diagram of charge separation in detection medium due to charged particle transit at high energy 41
Figure 23: Composite sample of PSI46v2 pixel readout, using a case of only one hit pixel per readout, showing all possible pixel address levels by repeating the hit and readout over all possible pixels (Gabathuler, Kurt (Paul Scherrer Institut), 2005) 45
Figure 24: Pixel readout of one cartridge mediated by TBM chip. Separate TBM and ROC contributions to the signal are labeled (Atramentov, 2010) 46
Figure 25: Hardware data flow schematic for PLT 47
Figure 26: Information flow for Online operation 51
Figure 27: Information flow for Offline operation during calibration 52
Figure 28: Information flow for Offline operation during testing 53
Figure 29: DIP facility dynamics 54
Figure 30: Theory of operation of interface server 55
Figure 31: Online Data collection start flowchart (proposed) 56
Figure 32: Online Data collection stop flowchart (proposed) 57
Figure 33: Offline Data collection creation page flowchart 58
Figure 34: Offline Data collection control page flowchart 59
Figure 35: Calibration Set flowchart 60
Figure 36: Calibration Set iteration flowchart for loading up the workloop queue to run or re-run calibration jobs 61
Figure 37: Calibration process flowchart 62
Figure 38: Python Workloop thread flowchart 63
Figure 39: Trigger setup flowchart 66
Figure 40: Finite State Machine diagram for the XDAQ TTCciControl application. The application is in the *halted* state when it first starts up. 67
Figure 41: Flowchart for decoupled monitoring of data taking 68
Figure 42: Configuration data anchors 70
Figure 43: Configuration management flowchart: New configuration list 71
Figure 44: Configuration list and job usage diagram. See also Figure 49 73
Figure 45: Configuration management flowchart: Add to configuration list 74

Figure 46: Conceptual Goal for the Interface's Database 77
Figure 47: Common resource between calibrations and data runs 77
Figure 48: Necessary extra structure for configuration information 78
Figure 49: Global runs and configuration lists in full detail. See also Figure 44 .. 79
Figure 50: Extra structure for calibrations: organization into ordered sets 79
Figure 51: Full details of calibration data structure 80
Figure 52: PLT Browser Interface Database Diagram 81
Figure 53: Storage Pattern for Calibration and Configuration Data in the Filesystem 82
Figure 54: The home page of the PLT browser interface 135
Figure 55: The navigation bar with the Data link highlighted 136
Figure 56: Starting a new data run 136
Figure 57: Choosing a calibration set for a new data run 137
Figure 58: Create the data run 138
Figure 59: Initialize the data run 139
Figure 60: Start the data run 140
Figure 61: A running data run 141
Figure 62: A stalled data run 142
Figure 63: The data taking indicator is visible on the Home page 143
Figure 64: A paused data run 144
Figure 65: Resumed data run 145
Figure 66: After stopping a data run 146
Figure 67: Data taking Off 147
Figure 68: Gray Initializing... 147
Figure 69: Yellow Initializing... 147
Figure 70: Initialization Stalled! 147
Figure 71: Cyan Starting... 148
Figure 72: Yellow Starting... 148
Figure 73: Start Stalled! 148
Figure 74: Running 148
Figure 75: Stalling... 149
Figure 76: Stalled! 149
Figure 77: Dark gray Pausing... 149
Figure 78: Yellow Pausing... 149
Figure 79: Not Pausing! 150
Figure 80: Paused 150
Figure 81: Dark gray Resuming... 150
Figure 82: Yellow Resuming... 151
Figure 83: Resumption Stalled! 151
Figure 84: Dark gray Stopping... 151
Figure 85: Yellow Stopping... 152
Figure 86: Not Stopping! 152
Figure 87: Stopped 152
Figure 88: Data Disappeared! 153

Figure 89: Lost Name! 153
Figure 90: Menu link to view existing data runs 154
Figure 91: Listing of past data runs 155
Figure 92: A past data run 156
Figure 93: The navigation bar with the Calibration link highlighted 157
Figure 94: List of calibration sets 157
Figure 95: New calibration set: edit the name 158
Figure 96: New calibration set: name edited 159
Figure 97: New calibration set: Sample calibration unselected, Gain calibration in mid-drag 160
Figure 98: New calibration set: newly created 161
Figure 99: New calibration set: running all calibrations 162
Figure 100: New calibration set: all calibrations done 163
Figure 101: New calibration set: some calibrations selected for running 164
Figure 102: List of calibration sets: examine an existing set 165
Figure 103: Existing calibration set: examine an individual calibration 166
Figure 104: Individual calibration page 167
Figure 105: Submenu for the navigation bar's Calibrations link 168
Figure 106: The navigation bar with the Global Runs link highlighted 169
Figure 107: The list of global runs 169
Figure 108: New global run page 170
Figure 109: Initialized new global run 171
Figure 110: A previously finished global run, showing its configurations 172
Figure 111: Submenu for the navigation bar's Global Runs link Power and Trigger Utilities Page 173
Figure 112: Home page with link to Power and Trigger Utilities page highlighted 174
Figure 113: The Power and Trigger Utilities page 175

INTRODUCTION

The Standard Model of Particle Physics

The development of the modern view of subatomic matter has gone hand in hand with progress in particle accelerator technology to reach high particle energies and fluxes. The length scale that can be resolved with particles such as electrons and positrons when colliding at center-of-mass energies of 100 GeV is about 6 x 10^{-18} m according to the Heisenberg uncertainty relation (Heisenberg, 1927). This length is also the typical range of the weak force responsible e.g. for the β-decay of neutrons in nuclei; the strong force range is about 10^{-15} m while the electromagnetic force has an infinite range. The gravitational force, which is the 4th fundamental force, does not play a role up to that length scale but might at shorter distances (higher energies).

In the Standard Model of Particle Physics (SM), forces between particles are mediated by field particles with internal angular momentum (or spin) of one unit of the Planck constant. The weak force is mediated by the W^+, W^- and Z^0 bosons, the strong force by 8 types of gluons, and the electromagnetic force by photons. The particles composing ordinary matter have spin-1/2 (called fermions) and are 6 flavors of quarks (called up, down, strange, charm, bottom, top) and corresponding 6 flavors of leptons (electron, electron neutrino, muon, muon neutrino, tau, and tau neutrino) (see Figure 1).

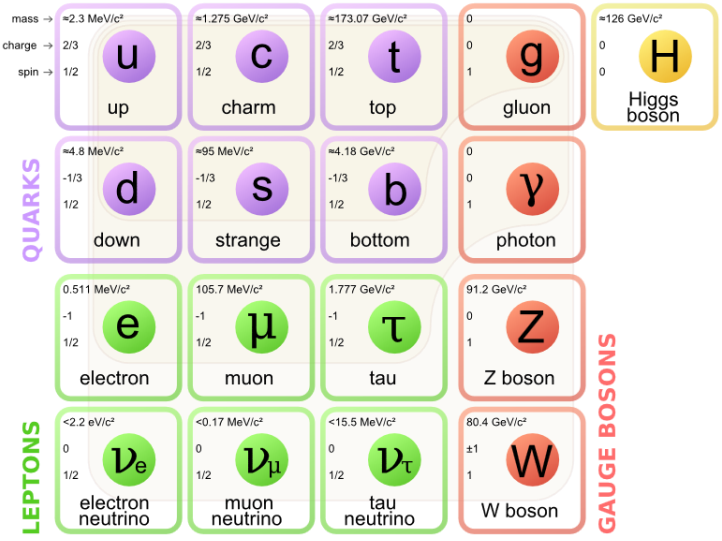


Figure 1: The Fundamental Particles of the Standard Model

The quarks and leptons are grouped into three generations with increasing mass (Figure 1 from left to right), with the top quarks in the 3rd generation having the same mass as a gold nucleus. In the SM the neutrinos are assumed to be massless, but there is experimental evidence (Fukuda, Y., et al (Super-Kamiokande Collaboration), 1998) that they have finite mass. For each particle there exists an anti-particle of opposite electrical charge, e.g. the positron is the anti-particle partner of the electron.

The SM is a quantum field theory (QFT) based on the symmetry group SU(3) x SU(2) x U(1), attributed to the three forces acting between particles, listed in order: strong, weak, and electromagnetic. The SM is renormalizable which refers to the fact that the model calculations of interaction probabilities predict finite values. In the classical view, the field for a force maps the force's strength and direction on an elementary charge at each point in space and time. The corresponding QFT treats particles as excited states of an underlying physical field; e.g. quantum electrodynamics has an electron and a photon field; the electromagnetic force between electrons is mediated by photons, the field quanta. A very important concept is gauge invariance, which is required in order to make a QFT renormalizable (t'Hooft, 1971). For example, the electric field is observable while the associated potential is not – adding a constant to the potential (applying a gauge transformation) does not change the electric field as it only depends on the potential difference. In other words, if the potential is a solution to the Maxwell equations, the gauge transformed potential is also a solution of these equations. Both solutions cannot be distinguished by experiment. The fundamental fields in the SM are not observable and change under gauge transformations, but quantities such as charges, energies etc. that are derived from the fields do not. This implies that all changes introduced by gauge transformations have to cancel each other. It turns out that just requiring the invariance under gauge transformations (also called gauge symmetry) already constraints the mathematical formulation of the underlying fundamental forces and makes the theory renormalizable.

The success of the unification of electric and magnetic force in 1865 by Maxwell (Maxwell, 1865) led to the effort to unify other forces. The Salam-Glashow-Weinberg model unifies the electromagnetic and weak force into a single electroweak force. The unification of these forces that appear rather different at low energies is expected to occur at energies above about 100 GeV, corresponding to a temperature of approximately 10^{15} K (as present shortly after the Big Bang). At even higher energies (greater than 10^{15} GeV corresponding to temperatures higher than 10^{27} K) Grand Unified Theories (GUTs) predict that the three SM forces become one electronuclear force. In 1999, G. t'Hooft and M. Veltman were awarded the Nobel prize in Physics for the proof that the electroweak theory is renormalizable.

Initially, the mathematical descriptions of the QFTs depend on massless fermions and mediator bosons. Only the photon, though, has zero mass, while the $W^{\pm}$ and Z^0 bosons are more than 80 times heavier than protons. This contradiction in the SM is resolved by generating these bosons in a gauge invariant way via spontaneous symmetry breaking caused by the Higgs mechanism. The three weak bosons obtain mass by interacting with the Higgs field that is assumed to permeate all space and is added to the SM. In more detail, the Higgs field induces the symmetry breaking of the electroweak field which leads to the generation of a massless scalar particle that in turn is re-absorbed creating a mass-term for the field. The particle associated with the Higgs field is the Higgs boson that acquires its own mass via self-interaction. The fermions in the SM also obtain mass as the result of their interaction with the Higgs field. Hence, it could be considered that the Higgs particle gives mass to other elementary particles (including itself) except the photon and the gluons.

The masses of particles are free parameters of the SM and must be determined experimentally. On July 4, 2012, the ATLAS and CMS experiments at the Large Hadron Collider (LHC) at CERN presented measurements consistent with the Higgs particle having a mass of about 125 GeV (CMS Collaboration, Higgs, 2012) (ATLAS Collaboration, 2012). But precision measurements of the Higgs particle production and relative decay rates are still needed to confirm its compliance with the SM. In 2013 P. Higgs and F. Englert were awarded the Nobel Prize in Physics, “for the theoretical discovery of a mechanism that contributes to our understanding of the origin of mass of subatomic particles”.

The SM provides predictions of particles and their interactions that have been in agreement with measurements to very high precision (Beringer, J. et al (PDG), ElectroWeak 2013 update) (Beringer, J. et al (PDG), QCD 2013 update). However, the SM has limitations and does not extrapolate reliably beyond the 100 GeV electroweak scale. A problem is that the mass of the Higgs particle is not a fundamental prediction of the SM and in principle can attain any value (Beringer, J. et al (PDG), Higgs 2013 update). Furthermore, the three fundamental forces seem not to converge exactly at higher energies (Beringer, J. et al (PDG), GUTs 2013 update). Evidence for the shortcomings of the SM are that the gravitational force and dark matter are not included. Furthermore, the assumption that neutrinos are massless contradicts experiment (Fukuda, Y., et al (Super-Kamiokande Collaboration), 1998) and the matter-antimatter asymmetry in the Universe cannot be explained without a more complete theory. There are many extensions to the SM, all of which predict new generations of particles and forces (mediator particles), including additional Higgs bosons (informally called New Physics). The ongoing program of the LHC is to conduct precision measurements of the Higgs boson candidate to potentially discover contradictions to SM predictions as indirect evidence for New Physics and to explicitly discover signals of new particle states. Therefore, the LHC is being

upgraded to reach center-of-mass energies of 13 TeV in the collision of protons and increased particle fluxes in the year 2015. The precision measurements and particle searches require the knowledge of the proton-proton collision rate resulting in particle production (luminosity) with relative uncertainties of less than 1%. For this purpose a new instrument, the pixel luminosity telescope (PLT), is being added to the CMS detector at the LHC.

The Importance of Knowing Luminosity

Particle detectors in contemporary accelerators directly measure rates of production of various kinds of particles that come out of beam particle collisions. The witnessed rates can be compared with rates predicted by theory. Theory, in turn, deals directly with interaction cross sections. The critical quantitative link between theory's cross sections and rates in detectors is luminosity.

$$R = \sigma \mathscr{L} \quad (1)$$

In equation 1, R represents a production rate (N per time), σ is a cross section (area) from theory for a particular reaction, and $\mathscr{L}$ is instantaneous luminosity. The instantaneous luminosity is dimensioned as a particle collision intensity, N per unit area per unit time, so the entire equation reads as: rate of output particles equals cross section times intensity of input (colliding) particles. The output particles are not necessarily of the same type as the input particles.

For colliders with bunched beams, the instantaneous luminosity has a basic definition of:

$$\mathscr{L} = f_{coll} \frac{n_1 n_2}{A_{eff}} \quad (2)$$

In equation 2, n_1 and n_2 are the numbers of particles in each of the two respective beam bunches, f_{coll} is the frequency at which opposing bunches collide with each other, and A_{eff} is the effective area through which the bunches overlap when they pass through each other.

A higher luminosity is favorable because it increases the production rate of rare processes that are of the greatest interest to physicists, such as the Higgs boson, possible supersymmetric particles, and other exotica. For previously established familiar processes, their production rate is also increased, allowing higher precision measurements via better statistical accuracy from larger samples. Luminosity and energy stand together as the two most important accelerator parameters.

Measuring Luminosity

Measuring an accelerator's luminosity, and measuring it with high precision, is vital to accurately measuring production cross sections from the observed rates. The luminosity determination is often the dominant systematic error.

Because precise measurement of cross-sections for various processes is an essential component of the LHC physics program, each experiment has its own subsystems for luminosity measurement. Stepping back a bit, the entire collider system manages a "delivered" luminosity to all of the experiments. The experiments themselves see a "recorded" luminosity that is somewhat lower than the delivered luminosity due to detector readiness and the settings of the beam optics that bracket each detector experiment. The measured luminosity has to be the individual experiment's recorded luminosity since that is the only one that matters for making collisions to analyze.

Luminosity measurement falls into two types of operation: online and offline. Online luminometers make measurements for each particle bunch crossing and report their integrated results to the rest of the experiment at some high frequency (from once per second to once per nearly half a minute). Offline luminometers record data over long periods of a day or more and may perform extensive analysis to provide their luminosity figures. The online luminometers produce uncalibrated relative luminosities (count rates) that (ideally) bear a linear relationship to the absolute instantaneous luminosity:

$$L_{Rel} = \frac{\mathscr{L}}{\mathscr{L}_0} \qquad (3)$$

The $\mathscr{L}_0$ above is the luminosity measured under well-known conditions. $\mathscr{L}$ itself is the absolute luminosity to recover by applying calibration. The offline luminometers may provide either the L_{Rel} or the finished $\mathscr{L}$, depending on the process.

The well-known conditions for a luminosity calibration should be time independent so that the luminosity at other times can be inferred from the uncalibrated luminometer signal. The canonical choice for calibration is the result of a Van der Meer scan (van der Meer, 1968). A Van der Meer (VdM) scan is a special procedure performed instead of data taking. The opposing beams' directions are incrementally adjusted both horizontally and vertically to "scan" them across each other in and out of coincidence. The positions of maximum coincidence are noted where the counting rate in the detector peaks. The scan process also reveals the horizontal and vertical sizes of the beams. Knowledge of the beam current can be combined with this to infer what the absolute

luminosity is, and the detector's response to the luminosity at max coincidence finally forms the basis of the luminosity calibration. The dominant limiting systematic for VdM scans is the measurement of the beam current.

Physics "standard candles" are reactions whose cross sections are well defined and understood by theory, and are also easy to distinguish experimentally.

$$\mathscr{L} = \frac{R}{\sigma} \qquad (4)$$

R is the rate of a chosen reaction seen in the detector, σ is the well-known cross section of the chosen reaction, and $\mathscr{L}$ is the inferred absolute luminosity. The luminosity for the standard candle process is determined only by extensive offline analysis to identify the particular reaction apart from the background. A notable standard candle, for example, is the decay of Z bosons into two muons. The dominant limiting systematic for that process at the LHC (due to colliding protons together) is the parton distribution function. The absolute luminosity uncertainty for $Z \rightarrow \mu^+\mu^-$ is currently limited to 4-5% (Stoynev, 2014).

The Experimental Apparatus

The following describes the Large Hadron Collider where protons collide at center-of-mass energies exceeding 7 TeV, and the Compact Muon Solenoid detector that measures the particles emanating from such collisions.

Overview

The Large Hadron Collider (Evans, Bryant, & (eds), 2008) is the most powerful particle accelerator and collider ever built and it is operational at CERN. It is housed in the former Large Electron-Positron Collider (LEP) circular tunnel with a circumference of about 27 km. It has unprecedented center of mass energy and design luminosity. Out of the possible eight interaction regions of the collider, four are equipped with underground caverns and detectors for physics research. ATLAS (Aad, G. et al (ATLAS Collaboration), 2008) and CMS (S. Chatrchyan, et al (CMS Collaboration), 2008), the two general purpose detectors, are located at the two high luminosity interaction points. The LHCb (A. Augusto Alves, et al (The LHCb Collaboration), 2008) detector, devoted to B-physics studies, and ALICE (K. Aamodt, et al (ALICE Collaboration), 2008), dedicated to the study of heavy ion collisions, are both designed to operate at a reduced luminosity. A schematic of the LHC together with accelerators feeding protons into it is shown in Figure 2.

Figure 3 shows a view inside the LHC tunnel with a dipole magnet in the foreground.

The LHC carries an unprecedented amount of beam current and stored beam energy. For the LHC's restart in 2015, each beam will have a current of 0.54 amperes and a stored energy of 313 MJ (Beringer, J; LBNL, 2013). The planned high luminosity upgrade (HL-LHC) starting beyond 2023 will have beam currents of 1.2 amperes and energies of 681 MJ (Beringer, J; LBNL, 2013). It takes 308 MJ to melt ½ metric ton of copper from room temperature, so there is quite a lot of energy to be dissipated in case the beams go astray. A bad accident could easily drill a hole through one's detector or through one or two steering magnets.

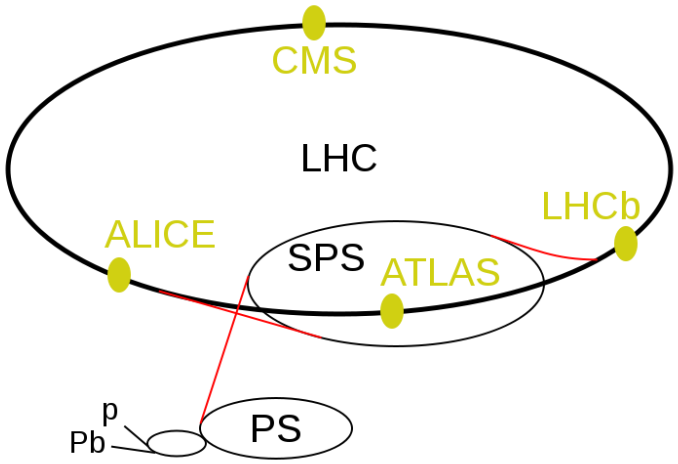


Figure 2: The CERN accelerator complex. Several machines are used to pre-accelerate the protons before their injection into the LHC

A test was conducted in 2005, well before the LHC started, to check the accuracy of beam damage simulations (Kain, Vorderwinkler, Ramillon, Schmidt, & Wenninger, 2005). A 450 GeV beam of various intensities was fired into a stack of 108 metal plates whose composition alternated regularly between zinc, copper, and two alloys of stainless steel. The damage done to a copper plate at ~1/3 deep in the stack is shown in Figure 4. The damage was done by a low-intensity beam of merely the LHC injection energy, nowhere near the multi-TeV energy or the high intensity used for LHC physics operation.



Figure 3: Beam pipe within the LHC tunnel, further enclosed by magnets and cryogenic cooling vessels (blue)

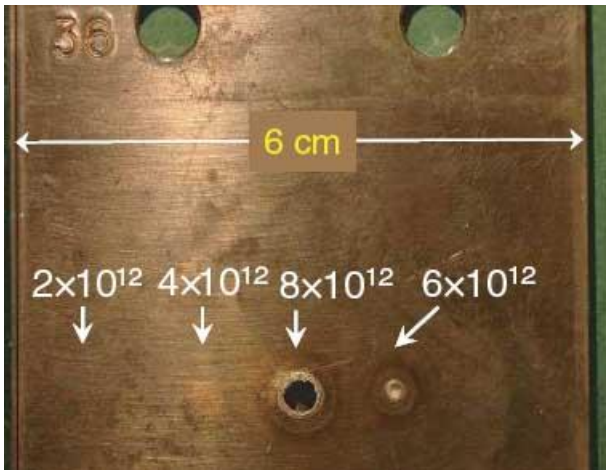


Figure 4: Damage caused by various 450 GeV beam intensities. The largest hole corresponds to being hit by 67 of the LHC's particle bunches (2808 is a full beam), and merely at the LHC injection energy (Schmidt, et al., 2006)

Aside from the worst case hole-drilling possibility, the more chronic average case hazard is to the cryogenic (1.85 K) magnets lining the beam pipe; a small deposition of beam energy would easily heat one above its critical temperature, causing it to resist the enormous current circulating within it (up to 11.85 kA), quenching the magnet. Without a rapid shutdown of the magnet, the potential energy of the magnetic field would then be changed into heat in the current bearing coils of the magnet. There is approximately 7.6 MJ in each magnet assembly, nearly 11 GJ throughout all the magnets of the LHC. That energy

would rapidly boil the liquid helium coolant around the magnet and risk overwhelming the pressure relief systems built into the helium enclosure. A breach of helium enclosure in 2008 by an electric arc caused extensive secondary damage, including breaking some magnet assemblies from their mountings and breaching the beam pipe (CERN/AT/Phl, 2008). Fortunately, there are already quench monitoring systems in place for all cryogenic magnets to safely cease the magnet current in time.

Even when the beam remains well under control throughout a data taking period and it becomes depleted due to normal and collisional losses, it still carries tremendous energy (~200 MJ) and must be disposed of safely. A specialized beam dump system diverts it out of the LHC, spreads it out, and aims it into a 7m deep by 0.7 m diameter carbon composite cylinder. The carbon cylinder is encased in a steel cylinder and sits in a cooling water jacket. The entire absorber system is further surrounded by 750 tons of concrete and steel. The beam is steered into a spiral pattern as it is being dumped into the absorber to prevent any part of the absorber getting much hotter than ~700° C.

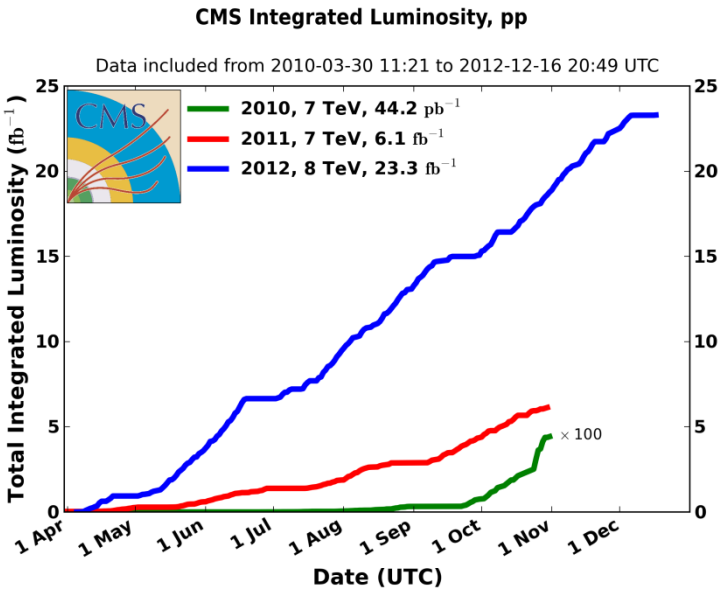


Figure 5: LHC's integrated luminosity (year-by-year comparison) seen at the CMS experiment

The Large Hadron Collider

The LHC is a proton-proton machine, designed to collide beams at the center of mass energy of $\sqrt{s}$ = 14 TeV, with a nominal luminosity of 10^{34} cm^{-2} s^{-1}. The

main motivation for choosing a hadron collider was to benefit from the highly variable center of mass energy in the interactions of the constituents of protons (quarks and gluons, combined called partons) to explore physics in the TeV region. The LHC can provide parton-parton collisions up to energies of about 1 TeV and with a very high collision rate, to favor the study of rare particle production. To carry out its physics program, the LHC was designed with demanding conditions that constitute a very ambitious technological challenge. Two different vacuum chambers are used to circulate the two proton beams in the 27 km circumference tunnel. It is filled with about 1200 superconducting dipoles each providing a magnetic field of 8.3 Tesla, which imposes the limiting factor to the achievable center-of-mass energy for the fixed curvature of the ring. In 2012 the LHC was running at the highest center of mass energy of $\sqrt{s}$ = 8 TeV with a maximum instantaneous luminosity equal to 7 x 10^{33} cm^{-2} s^{-1}. The time evolution of the total (time) integrated luminosity for the pp running periods since 2010 are shown in Figure 5. The LHC will resume operation at $\sqrt{s}$ = 13 TeV in Spring 2015.

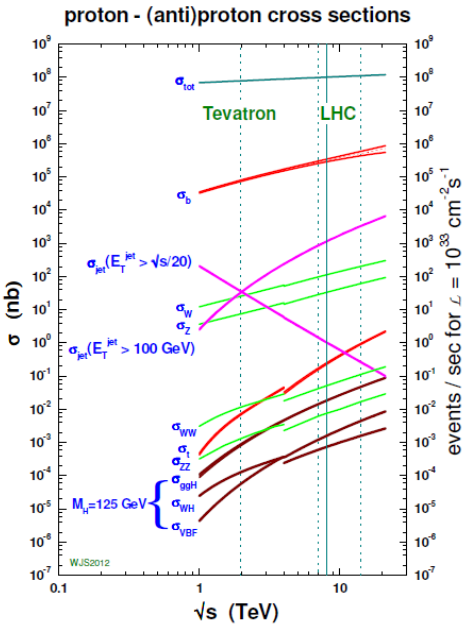


Figure 6: Comparison of various process production cross sections (Stirling, 2012). Discontinuities are due to switchover from computing collisions of protons vs. anti-protons to collisions of protons vs. protons at higher energies

Two main topologies of events are produced in pp collisions, depending on the energy transfer to the quarks and gluons inside the proton. The proton total

momentum is distributed among the partons. The center of mass energy of the parton scatterings is not known a priori but parameterized with parton distribution functions. Large-distance collisions between the two incoming protons lead to small momentum transfers ("soft collisions") and a suppression of particles that scatter at a large angle. The particles produced in the final state of such interactions have large longitudinal momentum, but small transverse momentum (pT) relative to the beamline and most of the collision energy escapes parallel to the beam pipe. The products of these soft interactions are called "minimum bias" events. They represent by far the majority of the pp collisions. This is also the population used for luminosity monitoring. In the cases when head-on collisions occur between two partons of the incoming protons, interacting at small distances, a large momentum can be transferred ("hard scattering"). In these conditions, final state particles can be produced at large angles with respect to the beam direction (high pT) with creation of massive particles. However, these are rare events compared to the soft interactions. The Higgs boson is dominantly produced when a gluon from each of the two colliding protons fuse together. A comparison of the cross sections of the typical processes at the LHC is shown in Figure 6.

The figure shows the predicted cross sections for several particle production processes and the expected event rate assuming an instantaneous luminosity of 10^{33} cm^{-2} s^{-1}. At $\sqrt{s}$ = 8 TeV the Higgs production rate is about 0.01 Hz, while the total event rate is about 9 orders of magnitude larger. For the observation of the Higgs in a particular decay one has to further take into account the relative decay probability (branching fraction). For example, the Higgs is expected to decay into two photons only 1% of the time so that the rate for such events is actually 10^{-4} Hz. This particular decay leads to two high energy photons in a transverse direction. This signature is distinct enough to separate the Higgs events from many other particle production events with fast event filters (triggers).
In a synchrotron ring accelerator, such as the LHC, particles are periodically accelerated in a radio-frequency (RF) electric field. The field is created in a sequence of field-regions and shielded regions (cavities). In time with their arrival at the RF cavity, the electric field is ramped to nearly max value to accelerate the charged beam particles. Once the particle is inside the cavity, the field can progress through its cycle to arrive at the opposite polarity and the particle exits and enters the next field zone between successive cavities. Hence the beam has to be segmented into bunches with length corresponding to the length of the cavities when traveling at the speed of light. The resultant bunch length for the LHC is 9 cm. The frequency of the LHC's RF is 400.8 MHz. During one oscillation period, a particle moves about 75 cm when traveling at (close to) the speed of light. Each bunch is filled with 1.2×10^{11} protons, and there are 2,808 bunches in the ring. Bunches are generally separated by 7.5 m spatially, or by 25 ns temporally, i.e. by 10 RF buckets. In addition there are 119 unfilled bunches corresponding to a travel time of 3 μs at the end of the sequence of bunches that

are unfilled. This is the beam abort gap; it leaves enough time for a fast kicker magnet to ramp up and redirect the beam toward the beam dump when requested or adverse beam conditions develop. Similar gaps exist for the PS and SPS which are used in sequence as pre-accelerators to feed the LHC. The bunch pattern is displayed in Figure 7.

The instantaneous luminosity can be estimated from beam parameters (Beringer, J; LBNL, 2013) using equation 2. Two bunches, each containing n = 1.15 x 10^{11} protons collide head-on at a frequency f_{coll} of 40 MHz. The effective area of overlap A_{eff} may be defined by $4\pi\sigma_x\sigma_y$, assuming that the transverse profile of the bunches is identical and is described by a Gaussian function everywhere along the bunch with RMS transverse beam sizes σ_x and σ_y in horizontal and vertical direction, respectively. With a $\sigma_x = \sigma_y = 20$ µm, an instantaneous luminosity of about $10^{34}/cm^2s$ can be achieved.

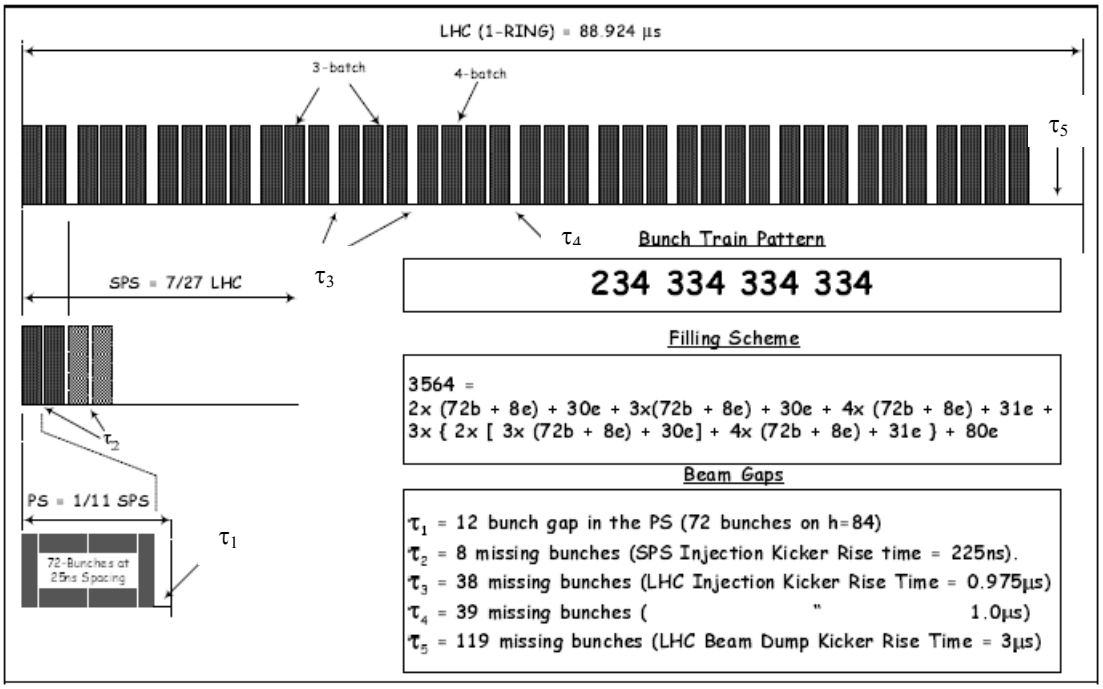


Figure 7: Bunch structure at the LHC. Each bunch is 9 cm long

Particle Detectors at the LHC

The LHC design value for proton-proton collisions at 40 MHz would provide ≈ 20 collisions per event, resulting in about 2000 charged particles every 25 ns. These reference values are indicative of the challenge to choose the detector technologies in order to perform in such a high radiation environment: excellent

time resolution, high granularity, and radiation hard materials are fundamental requirements. One of the most serious difficulties for the experimental operation at the LHC, with a major impact on the detector design, is the event "pile-up". Both "in-time pile-up", from multiple collisions produced in the same bunch crossing and filling the detector's acceptance, and "out-of-time pile-up", due to event overlapping between successive bunch crossings, are important contributions.

A fast response time is mandatory for the LHC detectors, in order to prevent the signal from becoming integrated over too many bunch crossings. The readout electronics have typical response times in the range of 20-50 ns (an integration spanning 1-2 bunch crossings). Residual pile-up in selected events must be subtracted in the offline analysis based on reconstructed particle tracks.

High readout granularity reduces the time the information needs to be collected from a detector since the occupancy rate per readout channel is reduced. Furthermore, it enhances the discrimination of particles originating from different pile-up locations. The LHC detectors have to be radiation resistant, primarily because of the high flux of particles originating from the pp collisions, but also from secondary particles created by lost beam particles. The particle flux in the innermost sensitive layer for charged particle tracking (closest around the beam pipe) exceeds 10^{15} particles/cm^2.

Detectors at colliders are designed to provide the best solid angle coverage around the interaction point. For the LHC, the natural reference frame consists of a right-handed cartesian system, whose origin is fixed in the interaction vertex, with the x axis pointing towards the LHC center, the z axis being parallel to the beam line and the y axis upward. More commonly used is the cylindrical coordinate triplet (r, ϕ, η): r is the distance from the z axis, ϕ is the azimuthal angle in the transverse plane and η is the pseudo-rapidity defined as $\eta = -\ln(\tan(\theta/2))$. Here, θ is the polar angle as measured from the z axis.

The CMS Detector

The Compact Muon Solenoid (CMS) is a general purpose detector that is installed at the interaction point number five along the LHC tunnel. The detector has a cylindrical shape, symmetric around the beam and is divided into a cylindrical central region (barrel) and circular forward/backward region (end-caps). The overall dimensions of CMS are a length of 21.6 m, a diameter of 14.6 m and a total mass of 12500 tonnes (metric tons). The CMS detector is built in several layers surrounding the beam pipe, where the innermost is a charged particle tracking device based on silicon, followed by an electromagnetic calorimeter (ECAL) for electron and photon detection, a hadronic calorimeter (HCAL) and an advanced muon detection system. The charged particle tracking

detectors are housed in a solenoidal superconducting magnet creating a magnetic field with strength of 3.8 Tesla in the inner part and 2 Tesla in the flux return housing the muon detectors. The overall layout of CMS is shown in Figure 8, and a slice of the detector is shown in Figure 9.

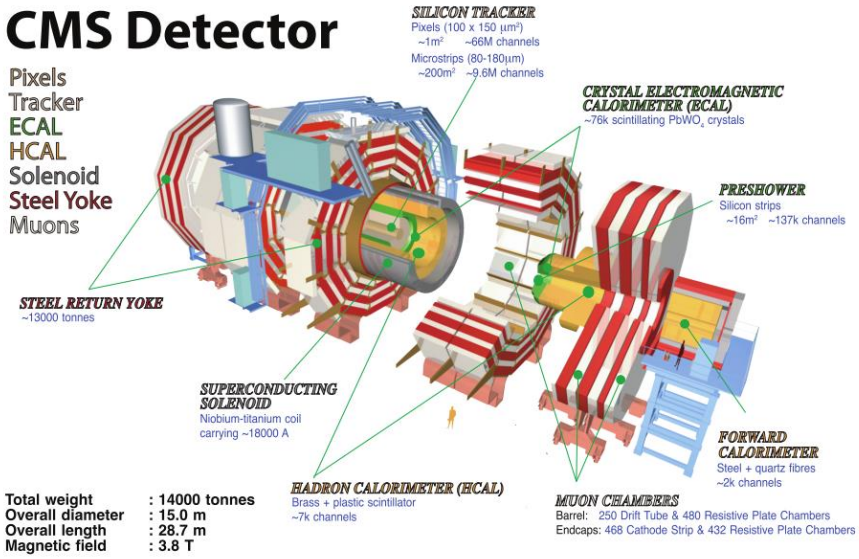


Figure 8: An overall view of the CMS detector complex. The detector is symmetrical around the beam line and designed to be hermetic

The Silicon Tracker

The CMS tracker (Karimäki, et al., 1997) (Karimäki, V. et al (CMS Collaboration), 2000) is the sub-detector closest to the interaction point and inside the 3.8 T magnetic field of the superconductive solenoid. It is designed to determine the primary interaction vertex, measure with good accuracy the momentum of charged particles, and identify secondary vertices associated with the decay of longer-lived particles. The tracker must operate in the harsh radiation environment of LHC without performance degradation over several years. Also, in order to maintain the excellent energy resolution of the EMC (ElectroMagnetic Calorimeter, the next detector layer enclosing the tracker), it has a stringent material budget. This means it must be constructed with a minimal amount of material to reduce its interference with the next detector layer(s). The tracker must not cast too great a shadow upon the EMC, especially with respect to the energy carried by light leptons (e^+ and e^-) and energetic photons. Figure 10 shows the thickness of the entire tracker system computed by simulation, quantified in radiation lengths, and plotted vs. pseudorapidity η. Near η of 0, the

thickness has an average low of 0.4 radiation lengths, favorable for the most transverse particle products from the IP.

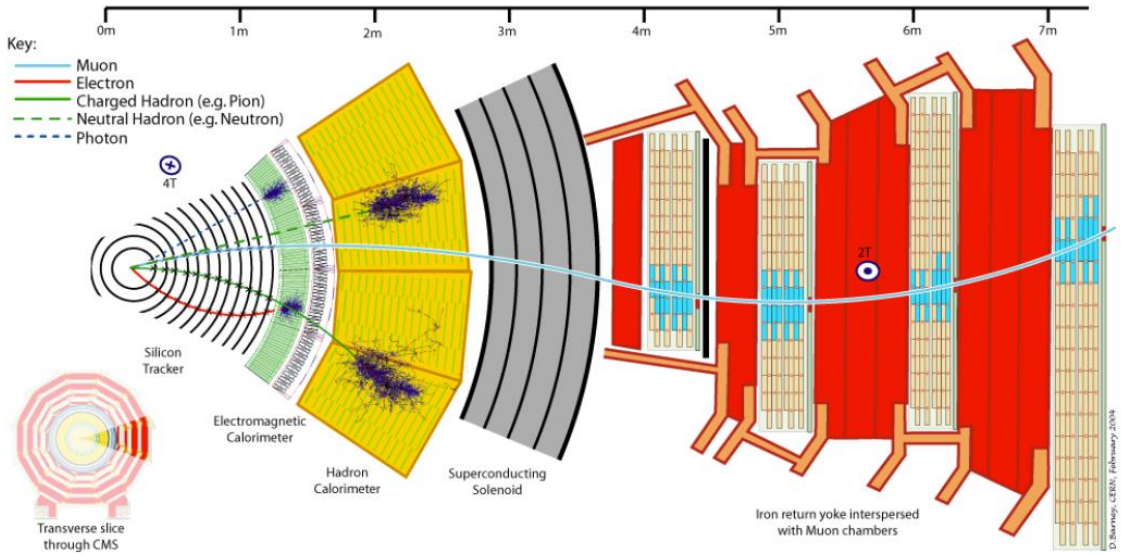


Figure 9: Transverse slice of the CMS detector with the tracks of an electron, a charged hadron (e.g. a pion), a neutral hadron (e.g. a neutron), a muon, and a cluster created by a photon. The electron and the photon deposit their entire energy in the electromagnetic calorimeter, generating an electromagnetic shower. A charged or neutral (no track) hadron reaches the hadron calorimeter where it is stopped. Only muons are able to traverse the entire detector. Their momentum is measured by the tracker and the muon system.

One radiation length (denoted X_0) is the distance over which an electron (very susceptible to energy loss) has its energy and momentum reduced to 1/e of its original value. For photons, one radiation length is 7/9 of its mean free path. Traversing 1.29 (9/7) radiation lengths converts 63% (1 – 1/e) of high energy photons into e^+ e^- pairs, with the attendant lossiness of those particles. A tiny fraction of one radiation length is best for preserving the original energy and momentum (E-p) of the most strongly affected particles, reducing the fraction of missing E-p due to tracker scattering and absorption. There is already missing E-p to look for due to neutrinos or more exotic theoretical interactions. Hadrons are far less affected by the tracker's material: the peaks around 1.8 radiation lengths in the plot correspond to only 0.56 hadronic interaction lengths when that quantity is plotted.

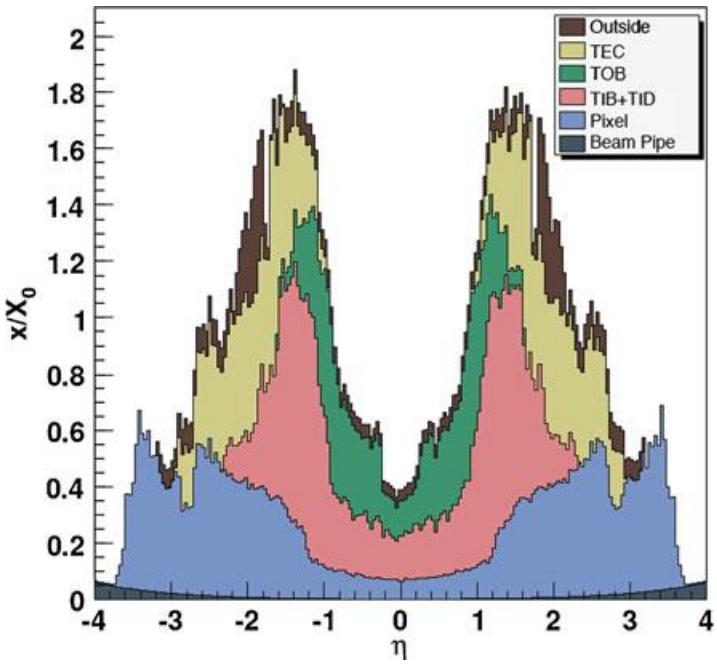


Figure 10: Radiation length thickness of the tracker vs. pseudorapidity (Caminada, 2012)

The pixel sub-detector is the part of the tracking system closest to the interaction point and is exposed to the highest particle flux. It consists of three barrel layers and two end-cap disks at each side of CMS (Figure 11). The barrel layers are located radially at 4.4 cm, 7.3 cm and 10.2 cm and they are 53 cm long. The two end-cap disks, extending from 6 cm to 15 cm in radius, are placed on each side at |z|= 34.5 cm and |z|= 46.5 cm. This design ensures that, within the $|\eta| < 2.2$ coverage region of the detector, at least two locations can be measured for charged particle tracks that originate close to the center of CMS. The total number of channels (pixels) is about 66 million, organized in about 16000 modules of 52 columns and 80 rows of pixels. The total active area is close to 0.92 m^2. The presence of high magnetic field causes a noticeable drift of the electrons (and a smaller drift for the holes) from the ionization point along the track. This leads to charge sharing between pixels which adjoin each other. Since the pixel detector uses an analog readout this is used to considerably improve the position resolution to better than 15 μm for tracks traversing the detector perpendicularly. In the end-caps the modules of the detector are arranged in a turbine-like shape with a 20 degree tilt in order to achieve charge sharing between neighboring pixels.

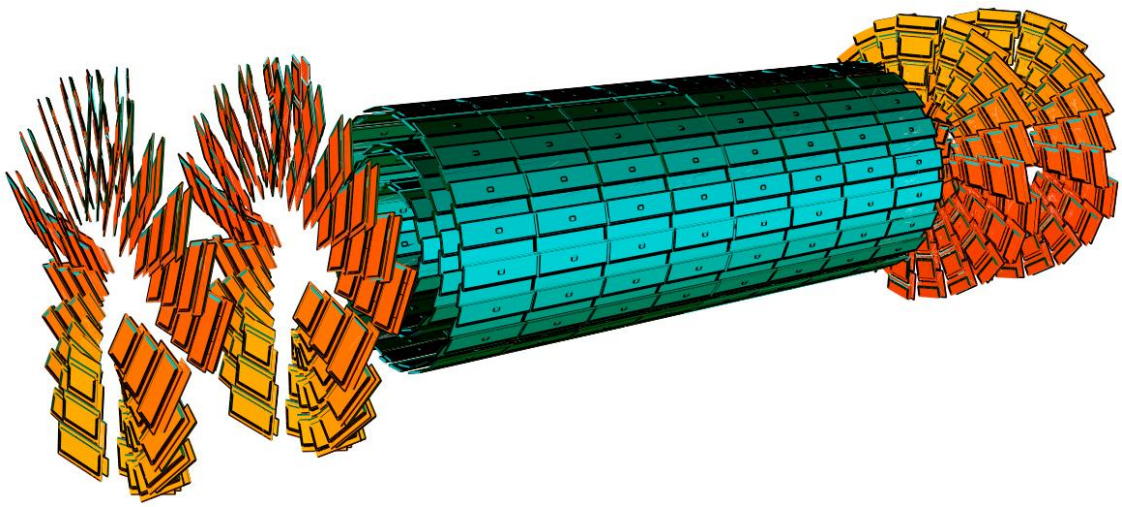


Figure 11: The silicon pixel detector (Hilbert, 2013)

The silicon pixel detector is surrounded by the silicon strip detector. The silicon strips are p+ doped silicon strips inserted into n-type silicon bulk whose thickness is close to 300 and 500 μm in the inner and outer tracker, respectively. In the barrel the strips are oriented parallel to the beam axis while for the end-caps they have a radial orientation. The inner tracker consists of 4 barrel layers and the two innermost layers use double sided detectors. The end-caps have 3 disks each. The outer tracker consists of 6 layers in the barrel (the two innermost are double sided) while the end-caps have 9 layers (the first, the second and the fifth are double sided). In total the silicon tracker has about 10 million channels and covers an active area close to 198 m^2 (Figure 12Figure 12).

The reconstruction efficiency of the tracker for isolated muons with transverse momentum greater than 1 GeV/c in the $|\eta| < 2$ region is close to 100%. High energy electrons are reconstructed with an efficiency of greater than 90%. The relative transverse momentum resolution for muons with transverse momenta between 1 and 100 GeV/c is 1-2% at $|\eta| < 1$ and remains below 3% up to $|\eta| = 2.2$. For particles with momenta greater than 10 GeV/c the transverse impact parameter is measured with a precision of about 20 μm or better.

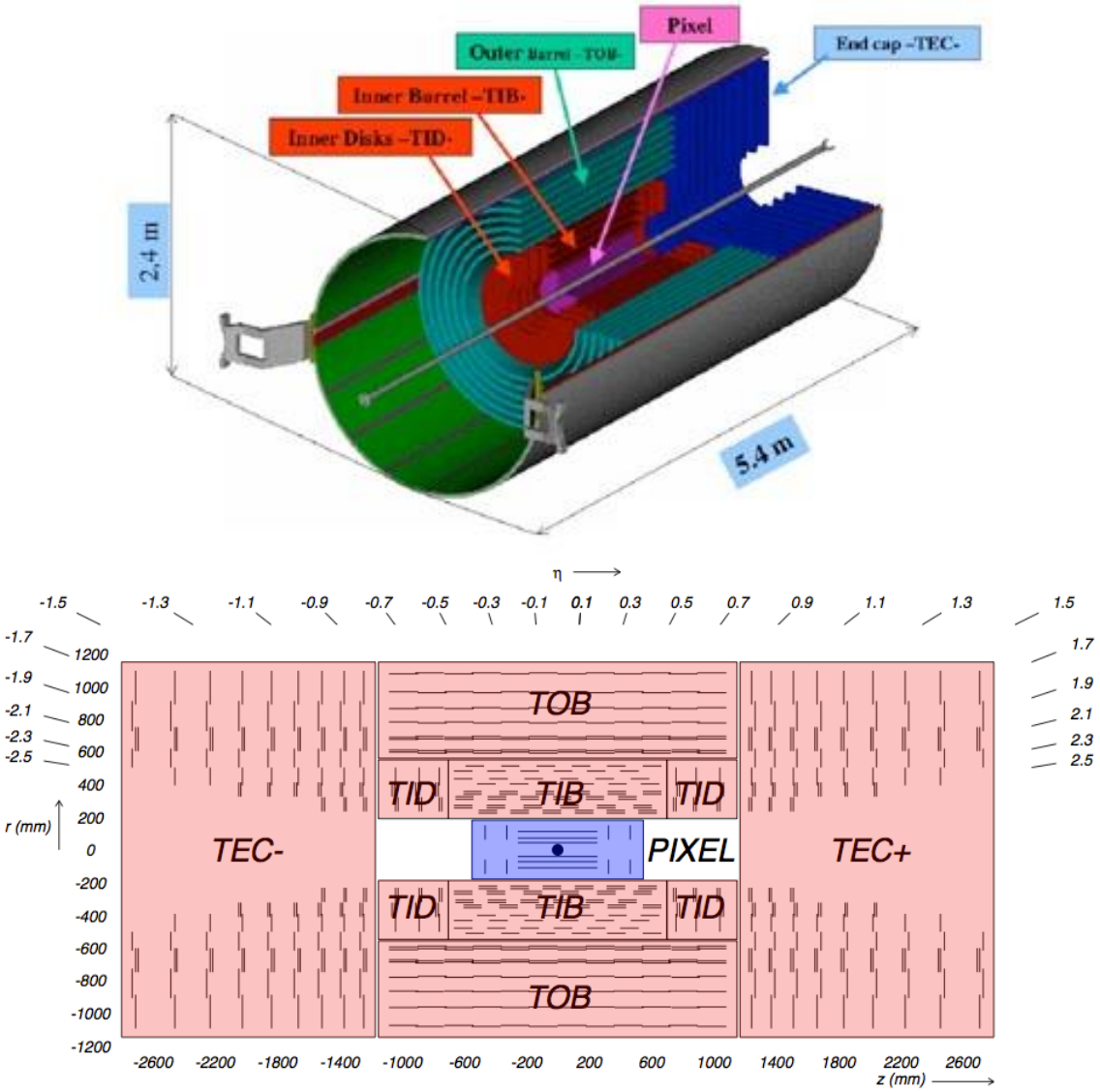


Figure 12: 3-D and cross section views of the entire tracker, showing both layers of silicon strip detectors. Top plot: (Schael). Bottom plot: (Lenzi, 2013)

The Electromagnetic Calorimeter

The electromagnetic calorimeter (ECAL) provides precise identification, position and energy measurements of electrons and photons (Pauss, F. et al (CMS Collaboration, ECAL Project), 1997) (Bloch, Brown, Lecoq, & Rykaczewski, 2002). It plays an essential role in the study of the electroweak symmetry

breaking physics as e.g. it detects and measures the two-photons from the decay mode $H \rightarrow \gamma\gamma$, and the electrons from $H \rightarrow ZZ^* \rightarrow 4e$ ($\rightarrow 2\mu 2e$).

Since the ECAL is placed inside the solenoid a compact design is required. ECAL is a hermetic, homogeneous calorimeter made of lead tungstate ($PbWO_4$) crystals; 61,200 crystals are mounted in the central barrel part, and 7,324 crystals in each end-cap (see Figure 13). The lead tungstate scintillating crystals were chosen because they have a short radiation length (X_0 = 0.89 cm) and a small Moliere radius (R_M = 2.2 cm); they are fast - 80% of the scintillation light is emitted within 25 ns. Furthermore, they are radiation hard. However, the relative low light yield (~30 photons/MeV) requires the use of photo detectors with high intrinsic gain that can also operate in a strong magnetic field. In the barrel, silicon avalanche photodiodes (APDs) are used, while vacuum photo triodes (VPTs) have been chosen for the end-caps, particularly because of their high radiation hardness. A water cooling system guarantees the long term stability within 0.1^{o} C in order to preserve the ECAL energy resolution performance, as crystals and APDs are sensitive to temperature changes. The desired ECAL performance requires that the front end signals are digitized to almost 16-bit accuracy. Instead, the pre-amplifier uses multiple gain ranges, digitizing and transmitting the signals to the off-detector electronics only for the highest non-saturated range so that a 12-bit ADC is sufficient.

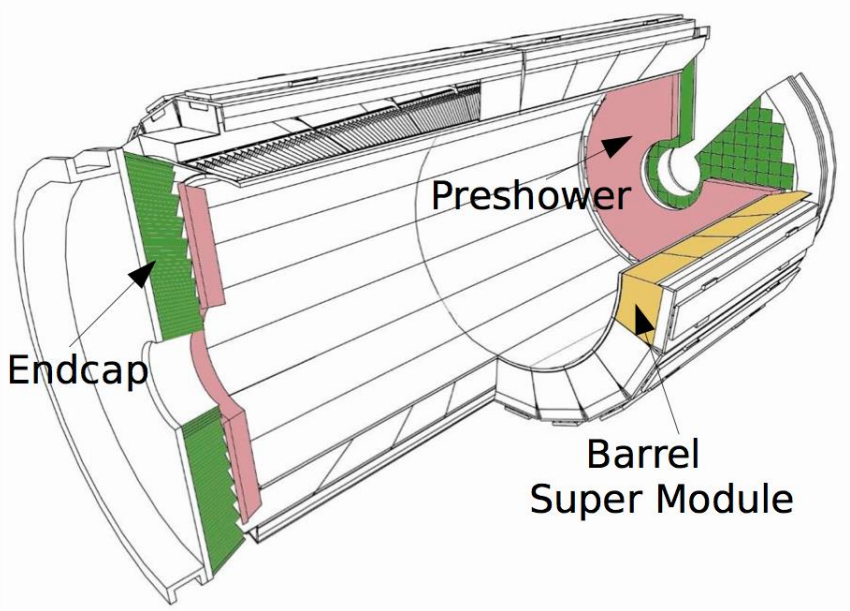


Figure 13: Schematic of the barrel and the end-caps of the CMS ECAL

A schematic of the ECAL geometry is displayed in Figure 13. The barrel region has pseudo rapidity coverage of $|\eta| < 1.479$. It has an inner radius of 129 cm and is structured in 36 super modules, containing 1700 crystals each, covering half the barrel length and subtending a 20 degree angle in ϕ. Each super module is divided along η into four modules which themselves are made of sub modules, the basic assembling alveolar units, containing 5 x 2 crystals each. The barrel crystals have a front face cross-section of about 22 x 22 mm^2 and have a length of 230 mm, corresponding to 25.8 X_0. The crystal axes are oriented with a 3 degree tilt with respect to the pointing geometry to prevent any particles escaping undetected into the regions between the crystals. The end-caps cover the pseudo rapidity region $1.48 < |\eta| < 3.0$, ensuring precision measurements up to $\eta = 2.5$. The end-cap crystals have dimensions of 28.6 x 28.6 x 220 mm^3. Each end-cap consists of a semi-circular aluminum plate from which structural units of 5 x 5 crystals, called super crystals, are cantilevered. A preshower device, whose purpose is to identify neutral pions in the end-caps within $1.653 < |\eta| < 2.6$, is placed in front of the crystal calorimeter. The active elements are two planes of silicon strip detectors placed behind disks of lead absorber at depths of 2 X_0 and 3 X_0.

A typical electromagnetic cascade releases 98% of its energy in a 5 x 5 block of crystals and 81% is contained in the crystal of impact. The detector relative energy resolution can be parameterized as:

$$\frac{\sigma E}{E} = \frac{S}{\sqrt{E}} \oplus \frac{N}{E} \oplus C \qquad (5)$$

where S is the stochastic term, taking into account the fluctuations in the shower lateral containment as well as contributions from photo statistics, and N is the noise term accounting for the overall electronic noise.

The error on the ECAL calibration contributes, together with light collection non uniformity, to the constant term C. The most stringent requirements on the ECAL energy resolution are imposed by the necessity to measure the two photons of the SM Higgs decay. Such a measurement requires excellent performance for energies up to 100 GeV, where the constant C contribution dominates. The relative energy resolution is expected to be better than 2% for electrons with energies above 10 GeV.

A light monitoring system continuously monitors (at 20 min intervals) the optical transmission of each crystal near the scintillation spectrum peak (~ 500 nm) and cross-checks it at a longer wavelength (~ 700 nm). Although the scintillation light yield of $PbWO_4$ crystals is not affected by radiation exposure, the crystals transparency is proven to decrease with the amount of radiation absorbed per

unit of time: at the nominal luminosity the transparency changes a few percent at the time scale of hours.

The Hadronic Calorimeter

The hadron calorimeter (HCAL) is placed just outside the electromagnetic calorimeter, and within the magnet (Green, D. et al (CMS Collaboration, HCAL Project), 1997). It plays a major role in the reconstruction of particle jets and missing energy in a collision event. It must allow the reconstruction of the di-jets invariant mass with high precision and efficient measurements of the missing energy which represent an effective signature in many channels of physics beyond the Standard Model. Similarly to the other sub-detectors, the HCAL has to provide good hermeticity, which is critical for determining the missing energy, and fine granularity for a wide separation of di-jets from resonance decays, and for the measurement of the invariant mass of the di-jets. To contain the energetic particles from high transverse momentum jets it has to constitute several hadronic interaction lengths. Its dynamic range has to be large: from detecting a single minimum ionizing muon to particle energy deposits of up to 3 TeV.

The barrel and endcap portions of the HCAL are shown in Figure 14. The barrel section of the HCAL covers the pseudo rapidity region $|\eta| < 1.74$, and the two end-caps extend the range to $|\eta| = 3$. The HCAL uses brass layers as absorbers that are interleaved by plastic scintillators used as active medium. The absorber layers' thickness is between 60 mm in the barrel and 80 mm in the end-caps, while the scintillator's layers are 4 mm thick. The barrel hadronic interaction length λ ranges from 5.46 λ at $|\eta| = 0$ to 10.82 λ at $|\eta| = 1.3$; the end-cap has 11 λ on average. The scintillator in each layer is divided into tiles with a granularity matching the granularity of the ECAL. The scintillation light is collected via wavelength shifters. The two hadronic forward (HF) calorimeters improve the HCAL hermeticity, covering the pseudo rapidity region $3 < |\eta| < 5$. They are placed at 11.15 m from the interaction point outside the magnetic field. Due to the extremely harsh radiation environment a different detection technique is used: a grid of quartz (radiation hard) fibers is embedded in an iron absorber.

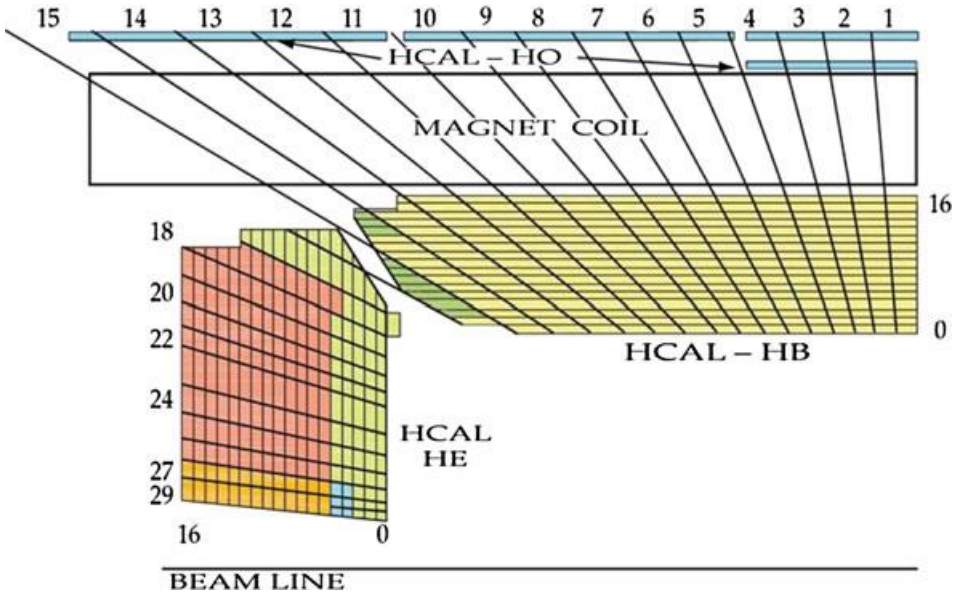


Figure 14: Cross section of hadron calorimeter. Hadron Forward section not shown (Caminada, 2012)

The Muon Spectrometer

In CMS the muon system consists of four active stations interleaved by the iron absorber layers which constitute the return yoke for the magnetic field (Gasparini, F., Mitselmakher, G. et al (CMS Collaboration, Muon Project), 1997). The return field is large enough to saturate 1.5 m of iron, allowing the muon stations to be integrated to ensure robustness and full geometric coverage. Each muon station consists of several layers of aluminum drift tubes (DT) in the barrel region, where the neutron induced background is small, the muon rate is low and the residual magnetic field in the chambers is low, and cathode strip chambers (CSCs) in the end-cap region, where the muon rate as well as the neutron induced background rate is high, and the magnetic field is also high, complemented by resistive plate chambers (RPCs) in both the barrel and the end-cap regions. The DT and CSC detectors are used to obtain a precise measurement of the position and thus the momentum of the muons, whereas the RPC chambers are dedicated to providing fast information for the Level-1 trigger. The layout of one quarter of the CMS muon system for initial low luminosity running is shown in Figure 15. In the Muon Barrel (MB) region, 4 stations of detectors are arranged in cylinders interleaved with the iron yoke. The segmentation along the beam direction follows the 5 wheels of the yoke. In each of the end-caps, the CSCs and RPCs are arranged in 4 disks perpendicular to the beam, and in concentric rings, 3 rings in the innermost station, and 2 in the others. In total, the muon system contains about 2500 m^2 of active detection planes, and nearly 1 Million electronic channels. For

muons with transverse momenta above 10 GeV/c and $|\eta| < 2.4$ the muon system information combined with the silicon tracker information ensures that the relative transverse momentum resolution stays well below 10%.

The Event Trigger

At the nominal LHC luminosity, the expected event rate is about 10^9 Hz. Given the typical size of a raw event (~1 MB) it is not possible to record all of the information for all of the events. The event rate is largely dominated by soft p-p interactions with particles of low transverse momentum. The triggering system must have a large reduction factor and maintain at the same time a high efficiency on the potentially interesting events, reducing the rate to 100 Hz, which is the maximum sustainable rate for storing events. The trigger system (Smith, W. H. et al, (CMS Collaboration, TriDAS Project), 2000) consists of two main steps: a Level-1 Trigger based on custom designed, largely programmable electronics, and a High Level Trigger (HLT) software system implemented in a computer filter farm of about one thousand commercial processors. The basic concepts will be described in the following subsections.

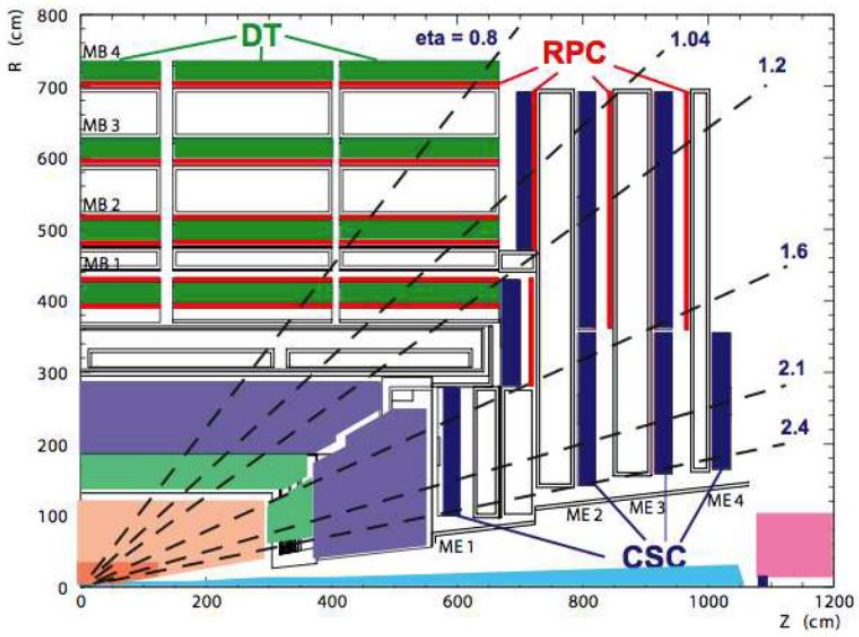


Figure 15: Layout of one quarter of the CMS experiment. The Muon Barrel system extends r > 350 cm and |z| < 700 cm (chambers are shown in green color). In the endcap part CSC chambers (noted as ME) are clearly visible (blue color). RPCs are displayed in red color in the barrel and end-cap parts.

Level-1 Trigger

The Level-1 trigger reduces the rate of selected events to 50 (100) kHz for the low (high) luminosity running. The full information collected during a particular bunch crossing is stored in pipelines of processing elements, while awaiting the trigger decision to keep or discard this event. The decision is made within 3.2 μs (latency). If the L1 accepts the event, the data is moved to be processed by the HLT. The L1 trigger has to make its decision in a time too short for reading information from all sub-detectors. It employs both calorimeter and muon chamber information only, while the data acquisition in the silicon tracker is too slow for this purpose.

The Calorimeter Trigger is based on arrays of 5 crystals in ECAL which match the granularity of the HCAL towers. These so-called trigger towers are grouped into calorimetric regions of 4 x 4 trigger towers. From the calorimetric region information the Calorimeter Trigger identifies the best four candidates of each of the following classes: electrons and photons, central jets, forward jets and τ-jets identified from the distribution of the deposited energy. The information of these objects is passed to the Global Trigger, together with the measured missing energy (MET). The Muon trigger is applied separately to each muon detector. The information is then merged and the best four muon candidates are transferred to the Global Trigger. The Global Trigger makes the accept-reject decision exploiting both the characteristics of the single objects and the combinations of them. The schematic of the L1 trigger is shown in Figure 16.

High-Level Trigger

The High Level Trigger reduces the output rate to 300 Hz. The HLT software performs the regional reconstruction on demand and uninteresting events are rejected as soon as possible. This is implemented in three “virtual trigger” levels: at the first level only the full information of the muon system and of the calorimeters is used, at the second level the information of the tracker pixels is added, and at the third and final level the full event information is available. The use of a processor farm for all selections beyond Level-1 makes maximal use of the evolution of computing technology since there is complete freedom in the selection of the data access, as well as in the sophistication of the algorithms.

Obtaining the Absolute Luminosity

The luminosity monitors of the CMS detector provide rate measurements and require an absolute calibration to convert the rate into a luminosity value. This number can be obtained from comparison of the production of Z-bosons that is predicted to a very high accuracy. Another approach uses the accelerator beams

and measures particle production as function of their separation near the nominal collision point. This is the Van der Meer scan method (van der Meer, 1968). It presents a hardware based determination of the luminosity calibration and therefore is complementary to the first method.

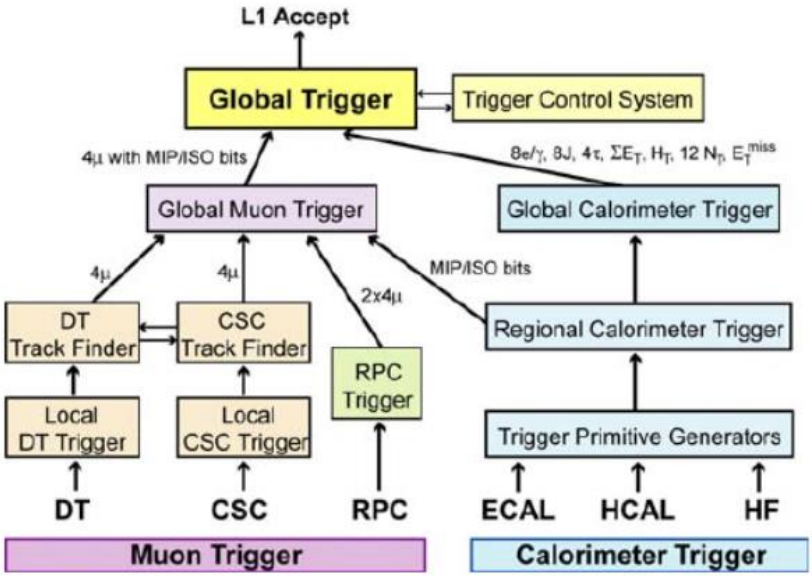


Figure 16: Schematic representation of the CMS L1 trigger system (Caminada, 2012)

Luminosity Calibration with Z-Boson Decays

The Z-boson is abundantly produced in proton-proton collisions at the LHC and can be reconstructed with the CMS detector from its decays into pairs of oppositely charged muons. Charged muons from Z decays have a very high identification and charged particle tracking efficiency, each exceeding 95% (The CMS Collaboration, 2012). Furthermore, the decay muons associated with tracks measured in the silicon tracker typically have a relative transverse momentum resolution of 2%. During data taking collision events containing Z-boson decays are enhanced with a fast event filter (trigger) that requires the presence of two isolated muons with transverse momentum greater than 24 GeV/c and pseudo-rapidity well within the central part of the detector ($|\eta| < 2.1$). In the muon-pair invariant mass M the Z-signal with an expected mass of 91 GeV/c^2 region is defined as 66 GeV/c^2 < M < 116 GeV/c^2 (see Figure 17). An absolute value for the luminosity is derived from the observed Z-boson yield N_{obs} in a given time interval (integrated luminosity L_{int}) and a theoretical value for the cross section, σ.

The time interval typically extends over a fill period of the LHC which lasts several hours. The cross section is calculated to a relative accuracy of about 4-5% (Hamberg, van Neerven, & Matsuura, 1991), (S. Catani, et al., 2009), (Melnikov & Petriello, 2006).

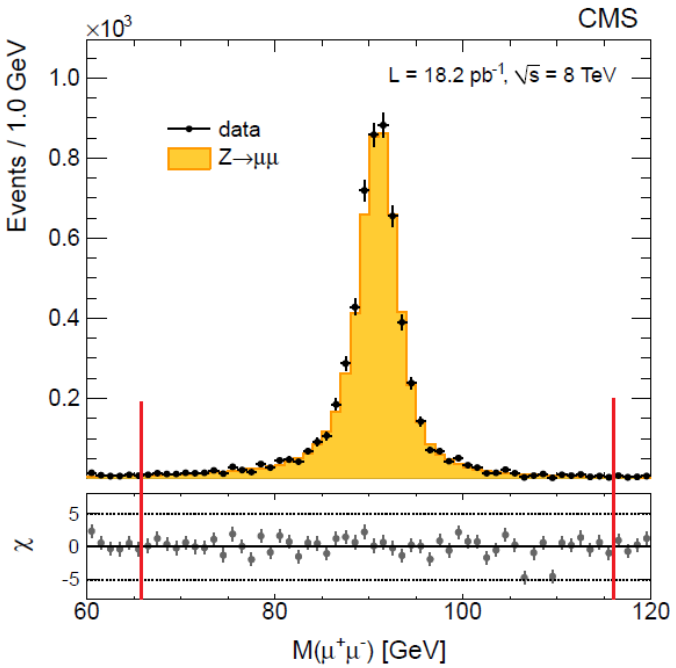


Figure 17: The di-muon mass distributions for Z boson candidate events – the red lines indicate the selection interval. The variable χ shown in the lower plot is defined as $(N - N_{exp})/\sqrt{N}$, where N is the number of measured events and N_{exp} is the sum of signal and background events.

The shaded area is the expected $Z \rightarrow \mu^+\mu^-$ signal as predicted from simulations (S. Chatrchyan et al. (CMS Collaboration), 2014). The main source of uncertainty is the level of understanding of parton distribution functions (PDFs) (Stoynev, 2014). This will improve as the PDFs can be adjusted as cross section measurements become available. The dominant physics processes that result in similar signatures for the Z-signal and need to be suppressed for comparison are Z-decays into tau-leptons, multi-jet events, and di-boson (WW, WZ, ZZ) production. The relative contribution of these backgrounds to the event sample can be far less than 1%. The estimate is obtained with full detector simulations of such processes.

The number of observed signal events is given as:

$$N_{obs} = A\varepsilon C L_{int}\sigma + N_{bkg} \qquad (6)$$

Here, A is the acceptance defined by geometrical and kinematic requirements on the decay muons, and obtained from detector simulations of the Z-boson production, and ε is the efficiency for triggering and reconstructing the event. The C accounts for systematic differences in efficiencies between simulation and data as determined from data control samples. The N_{bkg} is the number of background events. With the well-known cross section σ the relation can now be used to determine the luminosity L_{int}:

$$L_{int} = \frac{(N_{obs} - N_{bkg})}{(A\varepsilon C\sigma)} \qquad (7)$$

This number is then compared to luminosity values obtained from the HF, the pixel detector, and in future from the PLT. A Z-production cross section of 1110pb was used to calculate L_{int} at a center-of-mass energy of 8 TeV arriving at good agreement with the measurements (Stoynev, 2014).

Luminosity Calibration with Van der Meer Scans

As described earlier in **The Importance of Knowing Luminosity** subsection, the instantaneous luminosity L can be calculated from the knowledge of the number of protons in each of the bunches in a pair of colliding beams, n_1 and n_2, the frequency f_{coll} at which the bunches meet, and the effective area of overlap between the beams, A_{eff}. The latter is determined by moving the beams across each other in the (x,y) plane transversal to the beam direction (along z). The measurement of the particle production rate L(x,y) as function of their systematic displacement in x and y with respect to each other can be translated into effective area width and height (White, 2010), (The ATLAS Collaboration, 2011). In the ideal case that the beam profiles are Gaussian shaped and have the same widths σ_x and σ_y in x and y, respectively, the effective area is given as $A_{eff} = 4\ \pi\ \sigma_x\ \sigma_y$. As long as the profiles in both coordinates are uncorrelated, the effective area is given as the product of an effective width W and height H: A_{eff} = H x W. Both quantities are obtained from independent scans of L(x,y) as:

$$W = \frac{\int L(x,0)dx}{L(0,0)} \qquad H = \frac{\int L(0,y)dy}{L(0,0)} \qquad (8)$$

For Gaussian-shaped beams, the scan curves L(x,0) and L(0,y) will also be Gaussian distributed with RMS widths $\sqrt{2}\ \sigma_x$ and $\sqrt{2}\ \sigma_y$, respectively. The LHC beams typically cross at a small angle of 2 x 10^{-4} in the horizontal plane which

causes a small shift of the luminosity by about 1% with respect to head-on collisions. The beams can be tuned for head-on collisions during a special VdM scan. At typical LHC running conditions the transverse RMS widths are typically 50 μm, the RMS width along the beam's z is about 7 cm.

Corrections and uncertainties to the luminosity during the scan have several origins (The CMS Collaboration, 2014):

- The size of the beam separation. It is introduced with magnets and hence depends on the knowledge of the magnet hysteresis and imperfections in the overall beam steering around the accelerator as closed orbits are still maintained.
- The actual fill pattern. There are potential bunch slots every 2.5 ns along the beam, but only every 10^{th} is filled resulting in the spacing of 25 ns for collisions. Nevertheless, some charges can leak into the nominally empty slots. For the evaluation of the luminosity according to equation 2 (above) only charges in the active slots must be counted for n_1 and n_2. The actual fill of a bunch can be obtained from fast beam current measurements. The current of individual bunches circulating in the accelerator can vary by up to 30%.
- Growth of transverse beam dimension with time. Typically the time between scans in x and y is up to 30 minutes. During this time the transverse dimension of the LHC beam continues to grow slowly with time. In an ideal case, x and y scan should be done at the same time. Therefore, the effective RMS widths are evaluated under the assumption that they increase linearly over the time of measurement. If the growth in both dimensions occurs at the same rate the relative change in the widths will cancel. The residual uncertainty is due to differences in the growth rates.
- Beam-beam interactions. Colliding bunches when passing through each other exert mutual electromagnetic forces. One effect is that this increases their separation relative to nominal values. The relative change in the observed luminosity can be calculated from beam parameters and is about 1.5%. A small uncertainty of about 0.2% remains due to focusing or defocusing of the beams.
- Change in the nominal beam orbit. Small variations in the beam orbit influence the luminosity measurement as it impacts other beam parameters. It is used in the calculation of the beam separation above assuming stable conditions. Beam position monitors deployed around the LHC accelerator keep this luminosity uncertainty at about 0.2%.

- Extraction of the widths from measurements. The widths are extracted assuming certain distribution functions, in the simplest case single Gaussian functions. The distribution model assumption and the parameter uncertainties lead to uncertainties in the effective area respective luminosity. Furthermore the assumption that x and y measurements are uncorrelated might not be true, particular when the width and height distribution exhibit tails. A partial remedy is the use of double Gaussian functions. A better solution though is an effort to obtain highly Gaussian-shaped beam profiles due to beam optimization.
- Variation of beam conditions and luminosity detector calibration during the time of the scan.

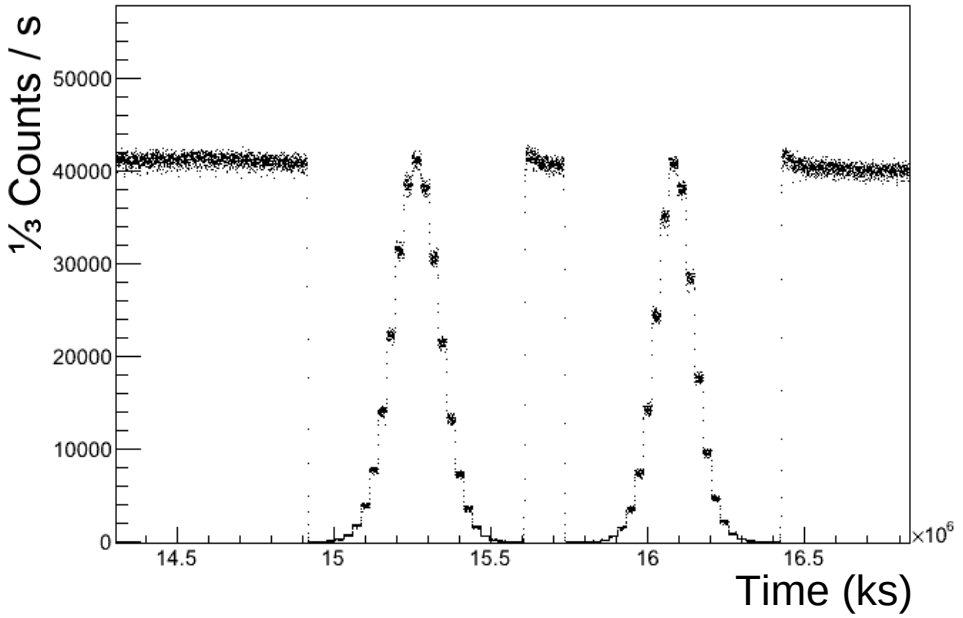


Figure 18: VdM scan seen by the PLT's diamond telescope. Data between the scan curves is due to beam optics set to nominal for collisions between scan iterations. The initial exponential declines in the latter is due to reduced charge collection via charge trapping in radiation-induced defects.

The PLT will simultaneously measure the luminosity during the VdM scan independently of LHC on a bunch-by-bunch basis. This allows an absolute comparison between PLT measurements and LHC luminosity calculations. Figure 18 shows rate measurements during a VdM scan with a prototype PLT mounted in front of the CMS detector during the 2012 data taking. The prototype

consisted of one telescope and the pixel detector material was single-crystalline diamond.

References

A. Augusto Alves, et al (The LHCb Collaboration). (2008). The LHCb Detector at the LHC. *Journal of Instrumentation, 3*(8), S08005. doi:10.1088/1748-0221/3/08/S08005

Aad, G. et al (ATLAS Collaboration). (2008). The ATLAS Experiment at the CERN Large Hadron Collider. *Journal of Instrumentation, 3*(8), S08003. doi:10.1088/1748-0221/3/08/S08003

ATLAS Collaboration. (2012). Observation of a New Particle in the Search for the Standard Model Higgs Boson with the ATLAS Detector at the LHC. *Physics Letters B, 716*(1), 1-29. doi:10.1016/j.physletb.2012.08.020

Beringer, J. et al (PDG). (ElectroWeak 2013 update). Electroweak Model and Constraints on New Physics. *Phys. Rev. D, 86*(010001). Retrieved from http://pdg.lbl.gov/2013/reviews/rpp2013-rev-standard-model.pdf

Beringer, J. et al (PDG). (GUTs 2013 update). Grand Unified Theories. *Phys. Rev. D, 86*(010001). Retrieved from http://pdg.lbl.gov/2013/reviews/rpp2013-rev-guts.pdf

Beringer, J. et al (PDG). (Higgs 2013 update). Status of Higgs Boson Physics. *Phys. Rev. D, 86*(010001). Retrieved from http://pdg.lbl.gov/2013/reviews/rpp2013-rev-higgs-boson.pdf

Beringer, J. et al (PDG). (QCD 2013 update). Quantum Chromodynamics. *Phys. Rev. D, 86*(010001). Retrieved from http://pdg.lbl.gov/2013/reviews/rpp2013-rev-qcd.pdf

Beringer, J; LBNL. (2013, September). *High-Energy Collider Parameters.* Retrieved from Particle Data Group: http://pdg.lbl.gov/2013/reviews/rpp2013-rev-hep-collider-params.pdf

Bloch, P., Brown, R., Lecoq, P., & Rykaczewski, H. (2002). *Changes to CMS ECAL electronics : addendum to the Technical Design Report.* Geneva: CERN. Retrieved from http://cds.cern.ch/record/581342

Caminada, L. (2012). Study of the Inclusive Beauty Production at CMS and Construction and Commissioning of the CMS Pixel Barrel Detector. In L. Caminada, *Study of the Inclusive Beauty Production at CMS and Construction and Commissioning of the CMS Pixel Barrel Detector* (Vol. XIV, p. 158 p.). Springer. Retrieved from http://www.springer.com/978-3-642-24561-9

CERN/AT/Phl. (2008). *Interim Summary Report in the Analysis of the 19 September 2008 Incident at the LHC.* CERN, AT. Geneva: CERN. Retrieved from https://edms.cern.ch/file/973073/1/Report_on_080919_incident_at_LHC__2_.pdf

CMS Collaboration, Higgs. (2012). Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC. *Physics Letters B, 716*(1), 30-61. doi:10.1016/j.physletb.2012.08.021

Evans, L., Bryant, P., & (eds). (2008). LHC Machine. *Journal of Instrumentation, 3*(8), S08001. doi:10.1088/1748-0221/3/08/S08001

Fukuda, Y., et al (Super-Kamiokande Collaboration). (1998). Evidence for oscillation of atmospheric neutrinos. *Physical Review Letters, 81*(8), 1562-1567. doi:10.1103/PhysRevLett.81.1562

Gasparini, F., Mitselmakher, G. et al (CMS Collaboration, Muon Project). (1997). *The CMS muon project : Technical Design Report.* Geneva: CERN. Retrieved from http://cds.cern.ch/record/343814

Green, D. et al (CMS Collaboration, HCAL Project). (1997). *The Hadron Calorimeter Technical Design Report.* Geneva: CERN. Retrieved from https://cds.cern.ch/record/357153/files/CMS_HCAL_TDR.pdf

Hamberg, R., van Neerven, W. L., & Matsuura, ,. T. (1991). *A complete calculation of the order αs^2 correction to the Drell-Yan K-factor.* Leiden, Hamburg: Nuclear Physics B 359 (1991) 343. doi:10.1016/0550-3213(91)90064-5

Heisenberg, W. (1927). *The actual content of quantum theoretical kinematics and mechanics.* Retrieved from NASA Technical Reports Server (English translation in 1983): http://ntrs.nasa.gov/search.jsp?R=19840008978

Hilbert, K. (2013). *CMS Pixel Detector Noise Calibrations.* Retrieved from Quarknet: http://quarknet.fnal.gov/fnal-uc/quarknet-summer-research/QNET2013/project_files/CMSPixelDetector.pdf

K. Aamodt, et al (ALICE Collaboration). (2008). The ALICE Experiment at the CERN LHC. *Journal of Instrumentation, 3*(8), S08002. doi:10.1088/1748-0221/3/08/S08002

Kain, V., Vorderwinkler, K., Ramillon, J., Schmidt, R., & Wenninger, J. (2005). Material Damage Test With 450 GeV LHC-Type Beam. *Proceeding of 2005 Particle Accelerator Conference* (pp. 1607-1609). Knoxville: CERN. Retrieved from https://accelconf.web.cern.ch/accelconf/p05/PAPERS/RPPE018.PDF

Karimäki, V. et al (CMS Collaboration). (2000). *The CMS tracker : addendum to the Technical Design Report.* Geneva: CERN. Retrieved from https://cds.cern.ch/record/490194/

Karimäki, V., Manelli, M., Siegrist, P., Breuker, H., Caner, A., Castaldi, R., . . . Cattai, A. (1997). *The CMS tracker system project : Technical Design Report.* Geneva: CERN. Retrieved from https://cds.cern.ch/record/368412/

Lenzi, T. (2013). *Development and Study of Different Muon Track Reconstruction Algorithms for the Level-1 Trigger for the CMS Muon Upgrade with GEM Detectors.* The Free University of Bussels, Physics. Geneva: CERN. Retrieved from http://cds.cern.ch/record/1551944/files/CERN-THESIS-2013-042.pdf

Maxwell, J. C. (1865). A dynamical theory of the electromagnetic field. *Philosophical Transactions of the Royal Society of London, 155*, 459-512.

Melnikov, K., & Petriello, F. (2006). *Electroweak guage boson production at hadron colliders through O(αs^2).* Honolulu, Madison: Physical Review D 74 (2006) 114017. doi:10.1103/PhysRevD.74.114017

Pauss, F. et al (CMS Collaboration, ECAL Project). (1997). *The Electromagnetic Calorimeter : Technical Design Report.* Geneva: CERN. Retrieved from https://cms-docdb.cern.ch/cgi-bin/PublicDocDB/RetrieveFile?docid=2713&version=2&filename=ECAL_TDR.pdf

S. Catani, et al. (2009). *Vector Boson Production at Hadron Colliders: A Fully Exclusive QCD Calculation at Next-to-Next-to-Leading Order.* Florence: Physical Review Letters 103 (2009) 082001. doi:10.1103/PhysRevLett.103.082001

S. Chatrchyan et al. (CMS Collaboration). (2014). *Measurement of Inclusive W and Z Boson Production Cross Sections in pp Collisions at √s = 8 TeV.* Geneva: Physical Review Letters 112 (2014) 191802. doi:10.1103/PhysRevLett.112.191802

S. Chatrchyan, et al (CMS Collaboration). (2008, 08 14). The CMS Experiment at the CERN LHC. *Journal of Instrumentation, 3*(8), S08004. doi:10.1088/1748-0221/3/08/S08004

Schael, S. (n.d.). *CMS Tracker*. Retrieved August 21, 2014, from RWTH-Aachen Physics Institute B: http://www1b.physik.rwth-aachen.de/~schael/CMS_Tracker.html

Schmidt, R., Assman, R., Carlier, E., Dehning, B., Denz, R., Goddard, B., . . . Zerlauth, M. (2006). Protection of the CERN Large Hadron Collider. *New Journal of Physics, 8*(290). doi:10.1088/1367-2630/8/11/290

Smith, W. H. et al, (CMS Collaboration, TriDAS Project). (2000). *CMS The TriDAS Project : Technical Design Report, Volume 1: The Trigger Systems.* Geneva: CERN. Retrieved from http://cmsdoc.cern.ch/cms/TDR/TRIGGER-public/CMSTrigTDR.pdf

Stirling, W. J. (2012). *Parton Luminosity and Cross Section Plots.* Retrieved August 7, 2014, from Imperial College HEP Group Research Web Pages: http://www.hep.ph.ic.ac.uk/~wstirlin/plots/crosssections2012_v5.pdf

Stoynev, S. (2014). *Luminosity Monitoring and Determination by Z Decays to Muons in CMS.* CMS Physics Analysis Summary CMS-PAS-LUM-13-003, CERN, Geneva.

The ATLAS Collaboration. (2011). *Luminosity Determination in pp Collisions at sqrt(s) = 7 TeV Using the ATLAS Detector at the LHC.* Geneva: The European Physical Journal C (2011) 71: 1630. doi:10.1140/epjc/s10052-011-1630-5

The CMS Collaboration. (2012). *Performance of CMS muon reconstruction in pp collision events at √s = 7 TeV.* Geneva: JINST 7 (2012) P10002. doi:10.1088/1748-0221/7/10/P10002

The CMS Collaboration. (2014). *Luminosity measurement for the CMS experiment.* Geneva: CERN. Retrieved from

http://cms.cern.ch/iCMS/analysisadmin/get?analysis=LUM-12-002-paper-v0.pdf

t'Hooft, G. (1971). Renormalizable Lagrangians for massive Yang-Mills fields. *Nuclear Physics B, 35*, 167-188. doi:10.1016/0550-3213(71)90139-8

van der Meer, S. (1968). *Calibration of the effective beam height in the ISR.* Technical Report CERN-ISR-PO-68-31, CERN, Geneva.

White, S. M. (2010). *Determination of the Absolute Luminosity at the LHC.* Orsay: PhD thesis, Université Paris-Sud 11, 2010. CERN-THESIS-2010-139; LAL-10-154. Retrieved from http://cds.cern.ch/record/1308187

CHAPTER I
A SYSTEM DEDICATED TO MEASURING LUMINOSITY: PLT

Abstract

The new detector subsystem for CMS is outlined and then parts are discussed in depth. The phenomenology of charged particles passing through bulk matter and the side effects of charge drift are explained, followed by device setup to sense the latter and convey the information all the way to data recording computers.

Introduction

The Pixel Luminosity Telescope (PLT) is a new subsystem being developed for the CMS detector at the Large Hadron Collider. It measures the bunch-by-bunch relative luminosity at the CMS interaction point on a time scale of a few seconds. The luminosity measurements integrated over some periods of data taking are then made available for the data analysis.

The PLT consists of three layers of charged particle trackers with an active area of about 5 x 5 mm^2, spaced equally through a depth of 7.5 cm. It can provide fast particle counting within steps of one bunch crossing (every 25 ns) and provide particle tracking at a larger time interval, for a subset of particles.

The PLT is placed a short distance (1.75 m) from the interaction point (IP) at an angle of 1.56 degrees ($\eta = 4.3$) with respect to the beam axis (see Figure 19 for the location inside CMS). It measures the charged particle flux by using the technology of the pixel tracker. The bunch monitoring capability is supplied by using a fast charged particle multiplicity counter feature of the pixel ROC that is not used by the pixel tracker. It will be joining two other luminometers already present in the CMS experiment: the Hadron Forward (HF) calorimeter and the pixel tracker.

The HF calorimeter was intended to supply relative luminosity measurements in real time with uncertainty at or below one percent, but actual usage uncovered multiple undesirable effects that increase its uncertainties to much larger values (CMS Collaboration, Lumi, 2012). The revised uncertainty for HF is 3.6% (Stoynev, 2014). The required correction to raw luminosity data was so complex and laden with additional uncertainties that the pixel tracker was used instead to make the corrections to the integrated luminosity for 2011 and 2012 data. After correction, a precision of 2.2% was established for 2011 integrated luminosity (CMS Collaboration, Lumi, 2012), and a precision of 2.5% was established for 2012 integrated luminosity (CMS Collaboration, 2013). The effort for building another luminometer is driven by the increased need for high precision monitoring on a bunch-by-bunch base. The two other monitors offer precision at the level of two to three percent. The best theoretical cross sections are known to a level of only one percent, so it would be desirable to have a luminometer

whose measurement is comparable with that level of precision, or possibly exceed it. The CMS experiment also remains in need of an online luminometer that works at the higher luminosities of the future. Therefore, the present PLT serves also to test detection technologies that can withstand higher irradiation expected for a higher luminosity LHC.

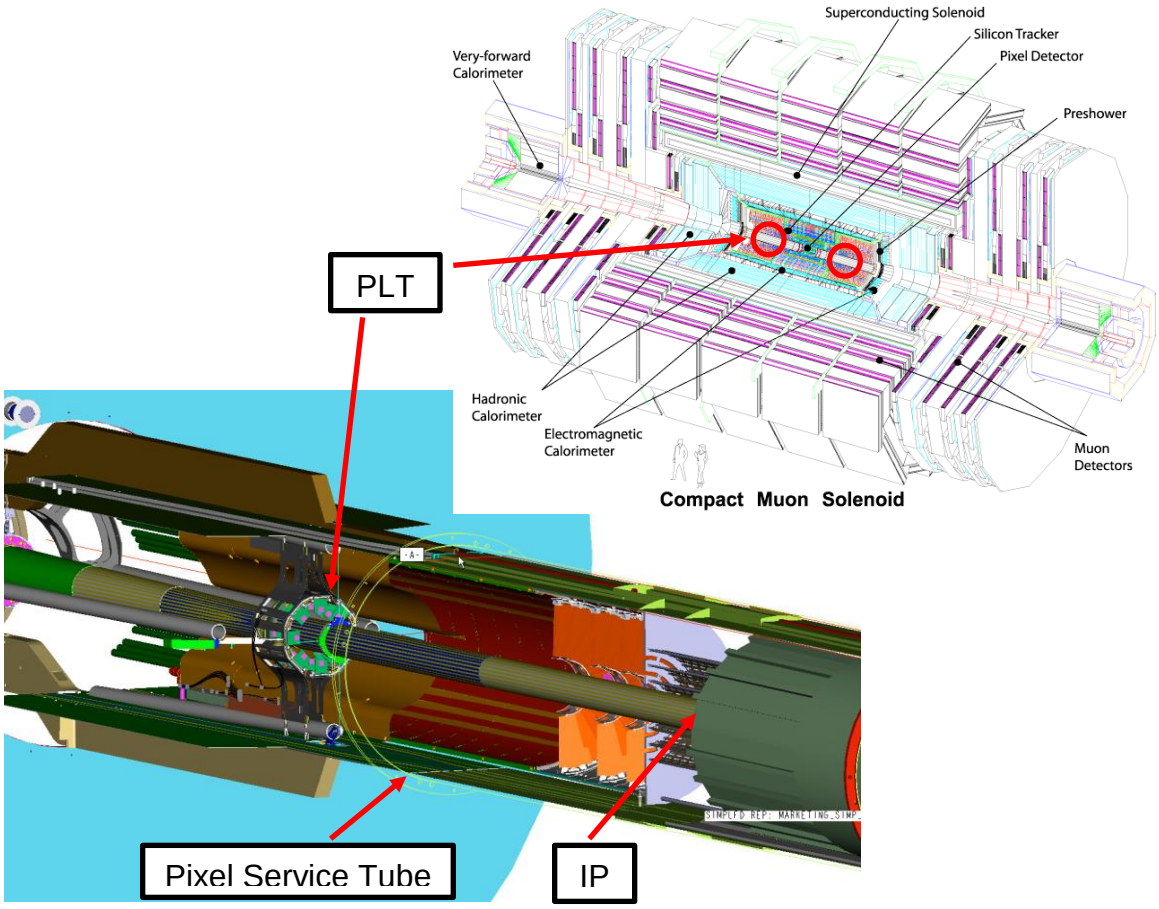


Figure 19: Location of the PLT inside the CMS detector (Atramentov, 2010)

Comprehensive simulations have shown that the dedicated PLT can achieve a statistical and systematic measurement accuracy of 1% for relative luminosity (Bartz, et al., 2009). This is the result after correcting for effects such as track overlap, coincidences that look like hits (the systematic effects), and drift in the location of the interaction point. Based on simulations, linearity in the measurement of the relative luminosity has also been shown for LHC luminosities over the next 10 years.

The precision of the final absolute luminosity is still limited by the precision of the luminosity calibration. The most reliable method is the Van der Meer (VdM) scan (van der Meer, 1968), which incrementally changes the vertical and horizontal aiming of the particle beams (scanning) to search for the aim settings that result in the greatest coincidence of the beams. The PLT (and all other sub-detectors) measures the variation of the radiation flux from the IP as the opposing beams are moving across each other. The maximum radiation from the IP corresponds to the maximum achievable overlap between the beams. The precision of the corresponding luminosity value is still limited by the beam current measurement provided by LHC (about 2% relative precision). Improvement of this measurement will further the approach to the larger goal of luminosity measurement precision meeting the best theoretical cross section precisions.

An additional benefit: bunch monitoring

In addition to measurements of luminosity, the monitoring of the beam structure of the LHC with the PLT and reporting this information in real time to the central LHC control will improve the overall beam conditions and luminosity performance.

The bunch pattern in the LHC is the result of the complex link between several stages of acceleration: there are four stages of injection linking four pre-accelerators with the LHC being the final stage. The third pre-accelerator, PS, first imposes the bunch structure seen in the LHC upon the particle beam. The rest of the structure of the LHC beams (gaps in bunch trains) results from accommodating injector rise times from PS to the next pre-accelerator, SPS, and then the rise times of injectors into LHC. The only other gap in the LHC bunch train is the beam abort gap, mentioned earlier. The bunch structure for the LHC is shown in Figure 7.

With the PLT a bunch-by-bunch determination of the luminosity can be obtained. The instantaneous luminosity measurement for each bunch, accumulated over 4096 beam orbits and published at 2.75 Hz, allows the observation of whether a bunch is fully filled with its required number of particles, is in deficit, or even completely absent. The converse condition, for bunch slots that should be empty, is also made apparent. The entire pattern of bunches circulating in the LHC is published in form of histograms. This allows verification of the pattern that is actually in circulation, offering feedback to LHC control about possible beam optics or RF problems.

PLT Implementation

Overview

The luminosity measurement is based on the measurement of the rate of charged particles through a well-defined charged-particle detector area. The charged particles of greatest interest originate from the interaction point (IP) at the center of the CMS detector, 1.75 meters in front of the PLT detector.

The PLT consists of several telescopes of three planes of pixel detectors, equally spaced over the length of 7.5 cm. The telescopes are placed in an octagonal arrangement around the beam pipe on both sides of the interaction point (each side has its own octagon, Figure 20, Figure 21). The planes are mounted with one edge up against the outside of the vacuum beam pipe with the normal of each plane parallel to the beam pipe axis. The positioning of the entire PLT within CMS is shown in Figure 19.

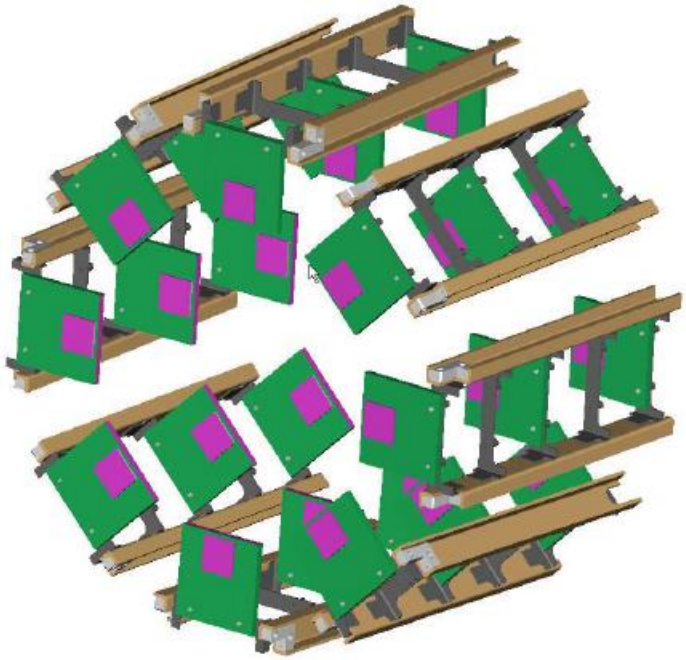


Figure 20: Each half of the PLT, an octagon of tri-plane telescopes around the beam pipe

The charged particle flux experienced by the planes falls into three categories: particles from proton-proton collisions at the IP, particles moving parallel to the beam pipe axis (beam halo), and random scattering events at any angle. The

luminosity is evaluated from the rate of triple coincidences in the telescopes that can be determined in a time interval well within 25 ns, the shortest separation of sub-sequent proton bunches in the LHC. Beam halo interactions and random scatterings will give a false elevation for the measured luminosity, so they must be distinguished from particles originating at the IP. The actual viewing angle of the telescope can be changed by setting pixels to inactive (masking). Furthermore, at a lower sampling rate, the trajectories of charged particles are completely reconstructed. The latter also provides monitoring of the IP position.

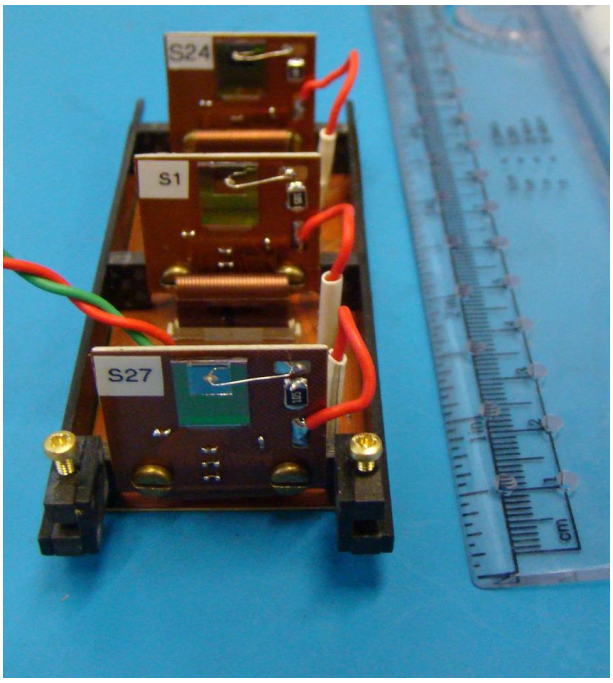


Figure 21: One PLT telescope, containing three planes of sensors mounted on read-out chips (ROCs). This is an earlier version of a telescope using diamond as the detector material, which does not completely cover the ROC.

The Sensors

The sensor medium is a thin layer of a semiconductor or an insulator material. High energy charged particles passing through the medium undergo Coulomb interactions with the atoms along their path. Two major categories for these interactions are elastic scattering on nuclei and inelastic scattering with atomic electrons. The latter category creates excited and ionized atoms. The ionization result is more useful, boosting bound electrons into a conduction band (free)

state, and consequently also creating “holes” where the freed electrons used to be bound. The electrons and holes recombine quickly unless they are acted upon by a strong ambient electric field, which causes them to migrate away from each other. Such an electric field is introduced into the sensor material by placing it between bias electrodes. Drifting charges induce a current in the electrodes (Shockley, 1938) (Ramo, 1939). The electron signal is read out on one side with a microelectronic sensing device that is connected to positive voltage or relative ground potential with negative voltage applied to the other electrode (Figure 22).

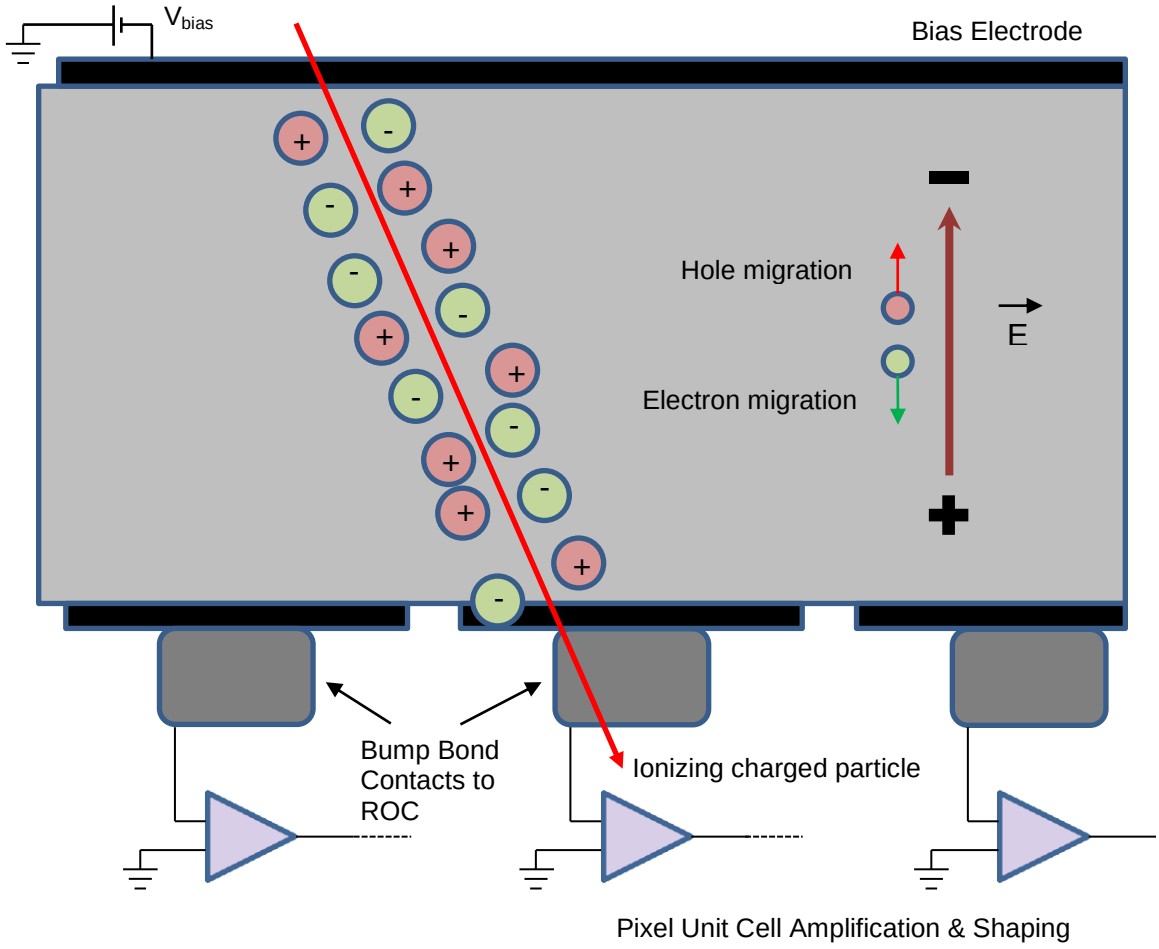


Figure 22: Diagram of charge separation in detection medium due to charged particle transit at high energy

The readout electrode can be segmented to define the location of the drifting charges within one segment. The current continues until electrons and holes within the sensor medium are stopped, either by reaching an electrode, by

electrons re-combining with holes, or by trapping charges somewhere within the material. The sensing device is a complex of circuits that integrate the current from the sensor. After a defined time period the charge signal is kept in a capacitor (sample-and-hold-device).

For the PLT two types of sensor material are considered: silicon, which is a semiconductor, and single-crystal chemical vapor deposited (CVD) diamond, which is an insulator. The choice is defined by the charge signal height and charge collection time in a detector of limited thickness, the radiation hardness, and operation conditions such as cooling and required voltage supply. The semiconductor choice is silicon doped forming a p-n junction. A reverse bias voltage of about 100V applied over a 285 μm thick sensor causes nearly the entire depth of the silicon to form an electric field region depleted of free charges. In case of the diamond a potential of 500V has typically been applied to 500 μm thick sensors.

- Charge yield

  The signal height for a minimum ionizing charged particle depends on the band gap of the material. The ionization is more extensive in the depleted p-n silicon medium due to the lower energy required to create an electron-hole pair (3.6 eV) as compared to diamond (13 eV). A minimum ionizing particle (MIP) in diamond creates in average 36 electron-hole pairs along every μm of the ionization path. A MIP in depleted silicon produces in average 67 to 87 such pairs per μm. This difference in pair production is compensated for by using a greater thickness for the less ionizable sensor material (diamond), accompanied by a higher bias potential to maintain electric field strength through the greater thickness. The thickness used for diamond allows 18,000 electrons to be collected from a normally incident MIP, comparable to the 25,000 electrons collected from a normally incident MIP for the p-n silicon. The time to collect charge deposited through the depth of the sensor is about 2.5 ns for diamond, and 5.9 ns for p-n silicon.

- Radiation hardness

  The particle fluxes expected for the next years of operation at the LHC exceed 10^{15} particles/cm^2. Charged particles are dominantly pions, with some protons, with the majority originating from the interaction point. Neutral particles are neutral pions and neutrons. These particles can dislocate atoms in the crystal lattice of the detector material, creating potential minima that can trap the secondary drifting charges. A charge removed from the drifting charge cloud is lost for the signal. Those

charges can be released after some time and therefore contribute to random noise. This requires a higher charge threshold for the signal, and hence reduces the sensitivity of the detector. When noise and signal become comparable, the detector becomes inoperable and must be replaced. Silicon detectors are widely used in the charged particle trackers. The energy to dislocate an atom in Si is about 15-20 eV, while for diamond, this energy is more than twice that value. Therefore, a priori diamond is considered more radiation hard than silicon. From simulations of the radiation during normal operation it is expected that silicon detectors will work for 3 years at lower beam intensity. The time is not yet established for diamond detectors.

- Operational Stability

  Due to the large band gap in diamond, this material does not need cooling during operation. In contrast, Si detectors need to be held at a low and constant temperature to reduce thermal noise and the dependence of the signal on temperature variation. This requires additional infrastructure for cooling Si detectors.

  Nevertheless, it has been observed that single crystalline diamond detectors become dependent on the particle flux after irradiation with neutrons to fluences of several 10^{13} n/cm^2 that are expected during one year of LHC operation. They trap charges in such a way that the external electric field is effectively reduced over short time intervals (minutes) when exposed to radiation from the LHC resulting in a decrease of the charge signal. Therefore, for the run in 2015, Si as the more established material was chosen and cooling pipes and thermally conductive detector mounts are installed.

The sensing electrode is divided into rectangular pixels of size 100μm x 150μm. Each pixel is connected to a channel in a readout chip (ROC) via a small pillar junction (“bump bond”) (see Figure 22). This structure of separated devices and bond points forces the mobilized charges to move directly toward the nearest sensing device. The arrival locations of secondary charges and their amounts indicate where the primary charged particle passed through the material. If the primary particle crosses more than one pixel volume, the charge collected from neighboring pixels can be used to determine a charge weighted position. This charge sharing improves the precision of the reconstructed location as compared to charge collected only from a single pixel.

Detector Readout

The pixel readout chip used by the PLT is the PSI46v2, designed by Roland Horisberger at the Paul Scherrer Institut and manufactured by IBM in its 0.25 µm feature size process (Hörmann, 2004) (Horisberger, 2008). The same pixel ROC is used by the well-established pixel tracker of CMS. Each ROC consists of 52 columns x 80 rows of pixels. The columns and rows are regularly separated at the same dimensions as the sensing electrode, 150 µm apart for the columns and 100 µm apart for the rows. The total active area is 7.85 mm x 8.00 mm. The actual used area may be a subset of this due to the size of the detection medium, bias electrode coverage, and bump bonds actually joining the detection medium to the ROC pixels. Individual pixels may also be set to active vs. inactive.

The chip supports two data acquisition modes: a fast sum of number of hits in the detector planes at high speed, and a full readout of the pixel addresses and stored charges used for reconstructing particle trajectories at a lower sampling rate.

The high-speed mode is a self-triggered mode called Fast-OR that communicates how many pairs of pixel columns (double-columns) had one or more charge deposits above a pre-defined threshold. The fast sum of such hits is generated anew on every clock cycle that drives the chip's operation (at 40 MHz, matching the frequency of beam bunch crossings) (Barbero, 2003). The ROC output has one of two possible meanings on each bunch crossing interval: it is either an analog signal whose level is proportional to the number of ***clusters*** of double-columns that had pixel hits, or it is a digital signal whose "true" level conveys that a threshold ***total number*** of double-columns had pixel hits. The threshold for driving the signal one way or the other can be set on an individual chip basis.

The digital "true" signal can be distinguished from the analog signal by the fact that it drives the signal to its maximum value. Any signal less than the maximum value is the analog signal. The signal magnitude is a current difference between two signal lines. Figure 8.14 in (Barbero, 2003) shows this for three different supply voltages. The number of pixel hits required to signal that their double-column has been hit can also be set. This affects all the double-columns of the chip uniformly.

For the present PLT operation, the thresholds are set so that:

- only one pixel must be hit in a double-column to signal that the double-column has been hit, and
- the digital double-column trigger fires at five or more total double columns, leaving the analog information to indicate up to four double-column clusters.

The lower speed mode is externally triggered at a few kHz and causes the chip to serially report a list of hit pixel addresses and charge pulse heights. Figure 23 shows the format transmitted: a three cycle (at 40 MHz) header followed by zero or more six cycle long pixel reports. The pixel reports are five cycles of base six digits for the column (2) and the row (3), and the sixth cycle is for the charge level. A special purpose communications management chip, a Token Bit Manager (TBM), receives the lower frequency readout trigger signals. Upon receiving each signal, the TBM makes the rounds of telling each of the three ROCs within a telescope cartridge to list their hit pixels on a common analog signal line. An oscilloscope recording of a (minimal) TBM mediated pixel readout is shown in Figure 24.

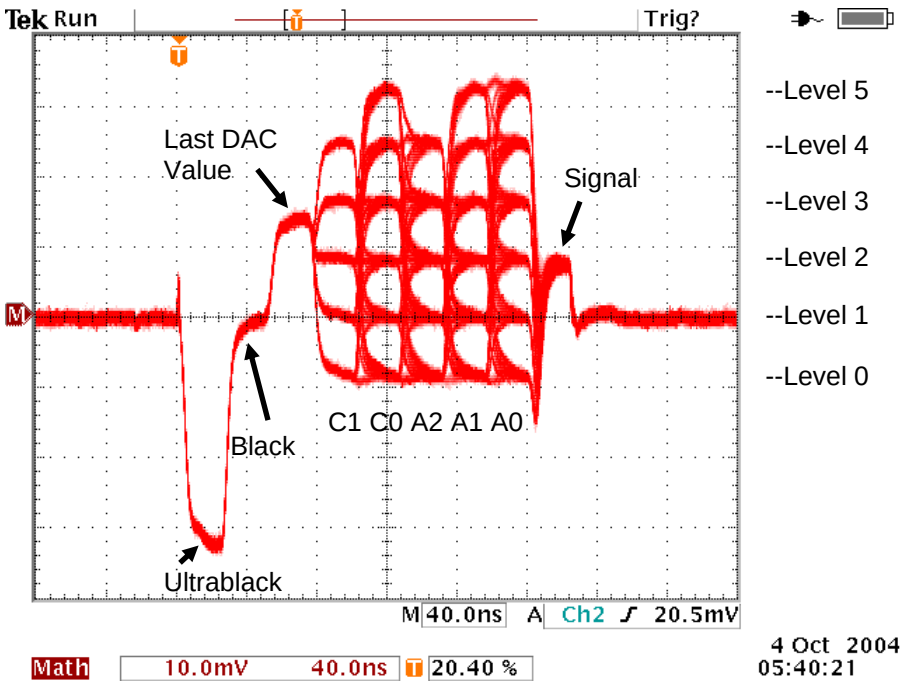


Figure 23: Composite sample of PSI46v2 pixel readout, using a case of only one hit pixel per readout, showing all possible pixel address levels by repeating the hit and readout over all possible pixels (Gabathuler, Kurt (Paul Scherrer Institut), 2005)

For the following description of the treatment of the ROC signals, refer to Figure 25.

The analog signals from the Fast-Or outputs of each individual ROC, and the analog pixel hit signals from each telescope are converted to optical analog

signals and sent into the service cavern via optical fibers. They arrive at the front end digitizers (FEDs) where specialized firmware turns them into digital information. The Fast-Or information is stored into histograms, one histogram per telescope. The triplet of Fast-Or numbers for each telescope is analyzed for a triple coincidence. Zero, one, or two is created from the minimum value of the triplet and added to the data bin that corresponds to the reading's individual bunch crossing. There are 3,564 bunch crossings in each one full orbit of the LHC's particle bunches (only 2,808 are of actually filled particle bunches), and the coincidence count histograms accumulate counts for 4,096 orbits, slightly more than 1/3 second. There are sixteen histograms in total, one for each telescope. The board that receives the ROCs' Fast-Or information and maintains the histograms is by turns referred to as the Custom FED and as the PLT FED. At the end of each accumulation period, the histograms are published by a Data Acquisition system (DAQ) on a dedicated PLT computer, and then cleared and reset for another accumulation period. The raw Fast-Or information is also provided at a higher frequency of a few kHz. In addition to providing the raw luminosity information, the Fast-Or histograms also enable the monitoring of beam bunch quality as described in the luminosity introduction.

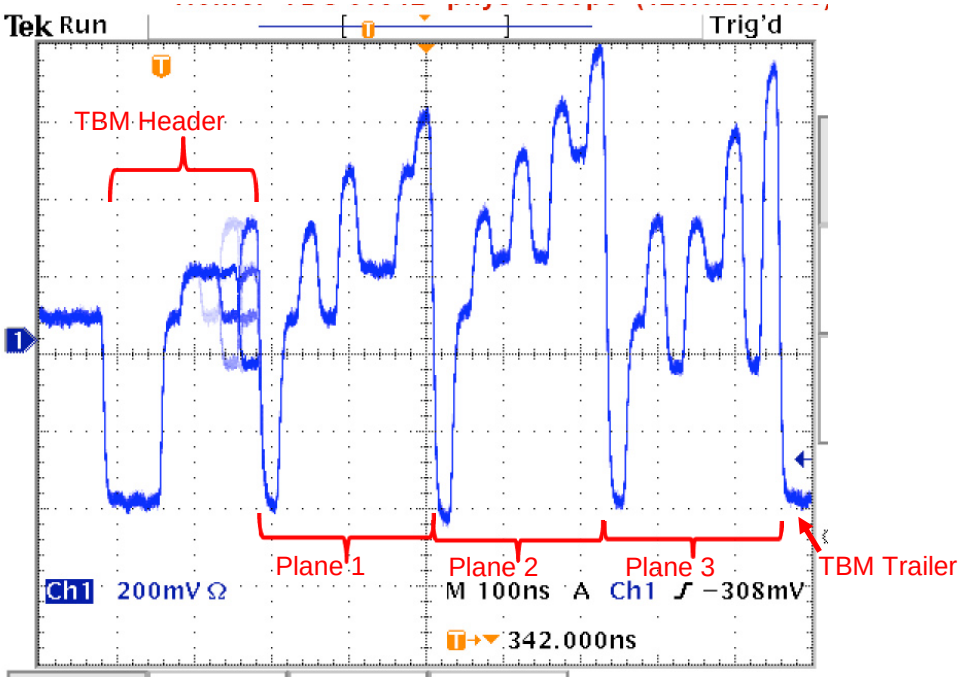


Figure 24: Pixel readout of one cartridge mediated by TBM chip. Separate TBM and ROC contributions to the signal are labeled (Atramentov, 2010)

The pixel readout signals are processed in a separate FED, referred to as the Pixel FED, which has the exact same firmware as the FEDs used by the pixel tracker sub-detector. The pixel hits are packaged into events and made available to a DAQ by the same dedicated PLT computer.

The actual track reconstruction from the pixel hit information occurs on the PLT computer. Via pattern recognition tracks originating from the IP are distinguished from beam halo tracks and tracks that were built from random scattering hits. Overlapping tracks are identified and hits are corrected for this. The results of the track analysis are then used to correct the luminosity derived from the Fast-OR information, monitor the location of the IP centroid, monitor the intensity of the beam halo, and to look for beam hot spots.

In the next chapter the browser interface software is introduced, which manages the execution of the data taking, the readout trigger control, and the running of detector calibration jobs. Web browsing, which came from CERN, now comes full circle in monitoring and (sometimes) controlling particle physics data acquisition.

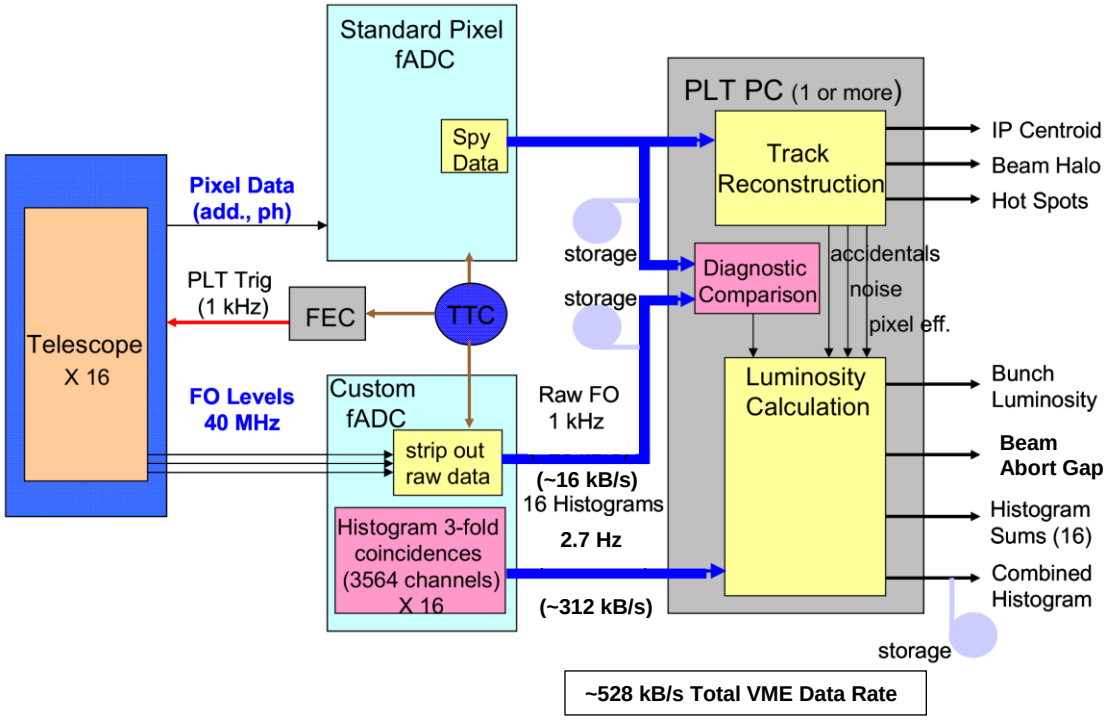


Figure 25: Hardware data flow schematic for PLT

References

Atramentov, O. (2010, April 12). *Prototype CMS Pixel Luminosity Telescope (PLT) at MTest.* Retrieved from Fermi National Accelerator Laboratory Directorate: https://www.fnal.gov/directorate/program_planning/all_experimenters_meetings/special_reports/Atramentov_T932_04_12_10.pdf

Barbero, M. (2003, December). *Development of a Radiation-Hard Pixel Read Out Chip with Trigger Capability.* Universität Basel, Institut für Physik. Geneva: PSI, CERN. Retrieved from http://cms.web.psi.ch/figures_and_presentations/thesis/Marlon-PhD.pdf

Bartz, E., Doroshenko, J., Halyo, V., Harrop, B., Hits, D., Macpherson, A., . . . Stone, R. (2009). A CMS Luminosity Monitor Using Single-Crystal CVD Diamond Pixel Detectors. *Journal of Instrumentation (JINST)*, 4-6.

CMS Collaboration. (2013). *CMS Luminosity Based on Pixel Cluster Counting - Summer 2013 Update.* CMS Physics Analysis Summary CMS-PAS-LUM-13-001, CERN, Geneva.

CMS Collaboration, Lumi. (2012). *Absolute Calibration of the Luminosity Measurement at CMS: Winter 2012 Update.* CMS Physics Analysis Summary CMS-PAS-SMP-12-008, CERN, Geneva.

Gabathuler, Kurt (Paul Scherrer Institut). (2005, October 24). *PSI46 Pixel Chip - External Specification.* Retrieved from ETHZ, IPP, CMS pixel group: http://cmspixel.phys.ethz.ch/docs/psi46-pixel-readout-chip/psi46v2.pdf

Horisberger, R. (2008, September 9). *A new ASICX design center at PSI.* Retrieved from CERN Indico: https://indico.cern.ch/event/36032/session/7/contribution/15/material/slides/0.ppt

Hörmann, C. (2004, October 7). *The Read Out Chip of the Compact Muon Solenoid Pixel Detector.* Retrieved from NIKHEF: http://www.nikhef.nl/~d90/gossip/choermann_phd_seminar_2004.pdf

Ramo, S. (1939, September). Currents Induced by Electron Motion. *Proceedings of the Institute of Radio Engineers (I.R.E.), 27*, 584-585. doi:10.1109/JRPROC.1939.228757

Shockley, W. (1938, October). Currents to Conductors Induced by a Moving Point Charge. *Journal of Applied Physics, 9*(10), 635-636. doi:10.1063/1.1710367

Stoynev, S. (2014). *Luminosity Monitoring and Determination by Z Decays to Muons in CMS.* CMS Physics Analysis Summary CMS-PAS-LUM-13-003, CERN, Geneva.

van der Meer, S. (1968). *Calibration of the effective beam height in the ISR.* Technical Report CERN-ISR-PO-68-31, CERN, Geneva.

CHAPTER II
USAGE AND OPERATION OF THE CMS PLT BROWSER INTERFACE

Abstract

The separate processes within the browser interface are enumerated and explained.

The Interface

A browser interface has been developed for use in both the CMS control room and for test stands. It combines control operation with processing and data quality monitoring and storage management. Its top level mode of operation is referred to as online or offline. When online, CMS's luminosity DAQ system (LumiDAQ) controls starts and stops of data runs, and the browser interface merely stays informed and monitors data collection for shift operator supervision and stores the data in its database. In offline, the browser controls are used to manually perform calibration runs and data runs. In the CMS control room, the dominant operating mode will be online (Figure 26). Offline mode is used to take data independent of other systems, calibrate the detector, and perform maintenance (Figure 27). For test stands, all operation is offline (Figure 28).

The Online state of the interface requires the presence of two systemic services. The first is the aforementioned CMS LumiDAQ, which is labeled as itself in Figure 26 and Figure 27. The CMS LumiDAQ retrieves information from all luminometers in the CMS detector, controlling start and stop of the data collection. The second systemic service, labeled as the "LHC Publication" object in Figure 26, is an LHC-wide service called the Data Interchange Protocol (DIP) cache (LHC Data Integration Working Group (LDIWG), 2004). It was invented for the exchange of information between the LHC machine and the individual experiments. The idea behind its operation is diagrammed in Figure 29. In terms of that diagram, the PLT is one of many data sources that inserts data into the cache.

Both the online and the offline mode of the interface depend on a third persistent service that controls the detector trigger. The interface communicates with a server process that operates the trigger hardware asynchronously via XML SOAP messages (Orsini, L. et al (Devel Man), 2014), (Hartl, 2013). The two triggering modes are one for taking data, and one for running calibrations. The precise details for each mode are set up ahead of time using configuration files for the served triggering application.

The browser interface itself is a server-powered process that persists on the PLT computer. The user operates a browser program on the same computer or on a different computer that connects to the PLT computer via internet. The browser communicates any user actions asynchronously to the server program. The

server reports back the results of the user actions, either to view a different system component, or to view any updates of system components that were addressed by the user. In the online mode, instead the LumiDAQ connects to the PLT interface server, in the same fashion, to switch data taking on and off.

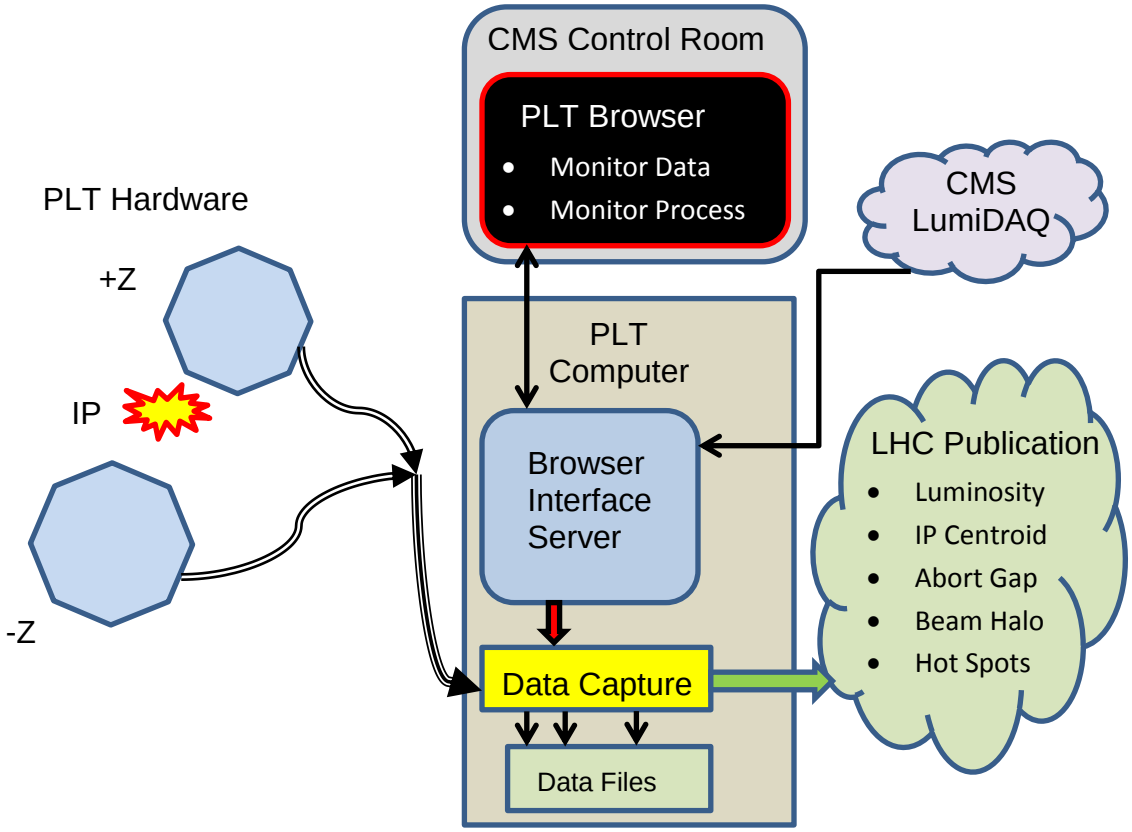


Figure 26: Information flow for Online operation

The information transmitted from PLT to the LumiDAQ connection is limited to success/failure indication, using a text message instead of an HTML page. A diagram of the browser- (and LumiDAQ-) to-server relationship and the server's actions is shown in Figure 30. For each request thread the interface server interacts with the database, starts the data taking or calibrations, and monitors the status of ongoing activity. During either data taking or calibration, only one process at a time is allowed to access the hardware. This is enforced by two sets of restrictions. At the command level, each job that accesses the hardware has to transition through a well-defined set of states. This is the concept of a state machine (Mealy, 1955), (Moore, 1956). The transitions are defined by the interface server and are recorded using the database. Keeping a record of states

and related information guarantees that only one process is running and other clients connecting to the server interface are brought up-to-date on the running process, but cannot bypass it. Even a second copy of the interface server running will not bypass the controls because both servers will still be referring to the same database and its records.

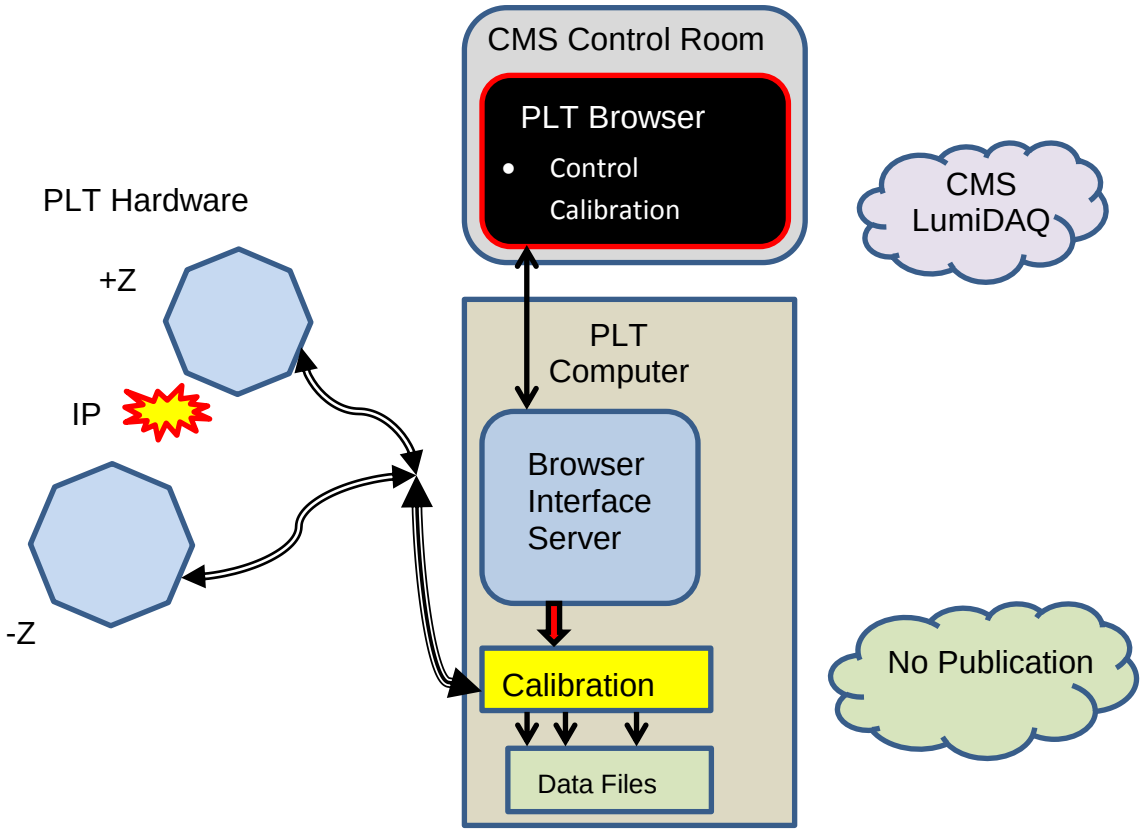


Figure 27: Information flow for Offline operation during calibration

The second restriction is at a lower level of running the job itself. A single job queue is used for all jobs. It serializes the order in which jobs are run, and it has only one slot for running jobs concurrently (only one job is given the CPU at a time). The database keeps track of configurations for data taking and the data files generated by jobs. Output data are tracked using records to represent running, past, and queued jobs. The database and the single job queue with a single slot keep hardware operations deterministic.

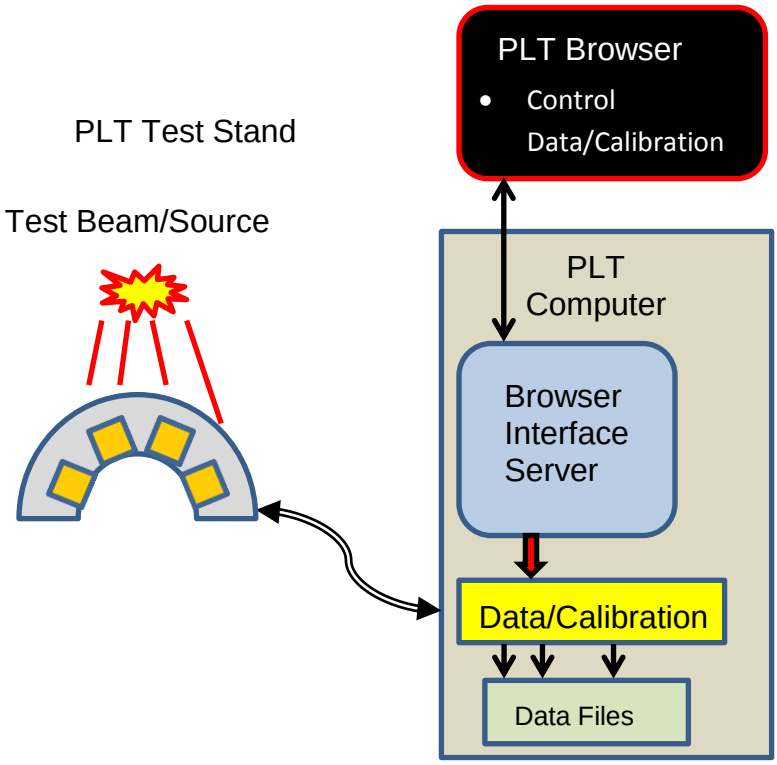


Figure 28: Information flow for Offline operation during testing

Breakdown of Functionality

Data Collection

During online operation, both data taking start (Figure 31) and stop (Figure 32) operations immediately coordinate the information between records, trigger control, and job queuing without any further user interaction. In the online mode a set of calibrations is automatically associated with the data run based on specifications about the type of data run as sent by LumiDAQ. In the offline mode, choices can be made via the user interface among available calibrations. The offline interface is implemented with two web pages with a separate flow chart each for creation of a job (Figure 33) and for data control (Figure 34). Note that the offline job control also offers pause and resume functionality.

Notice from the flowchart for job creation that the presence of an active record for data collection prevents the user from initiating a new data run. The user instead is directed to the control page.

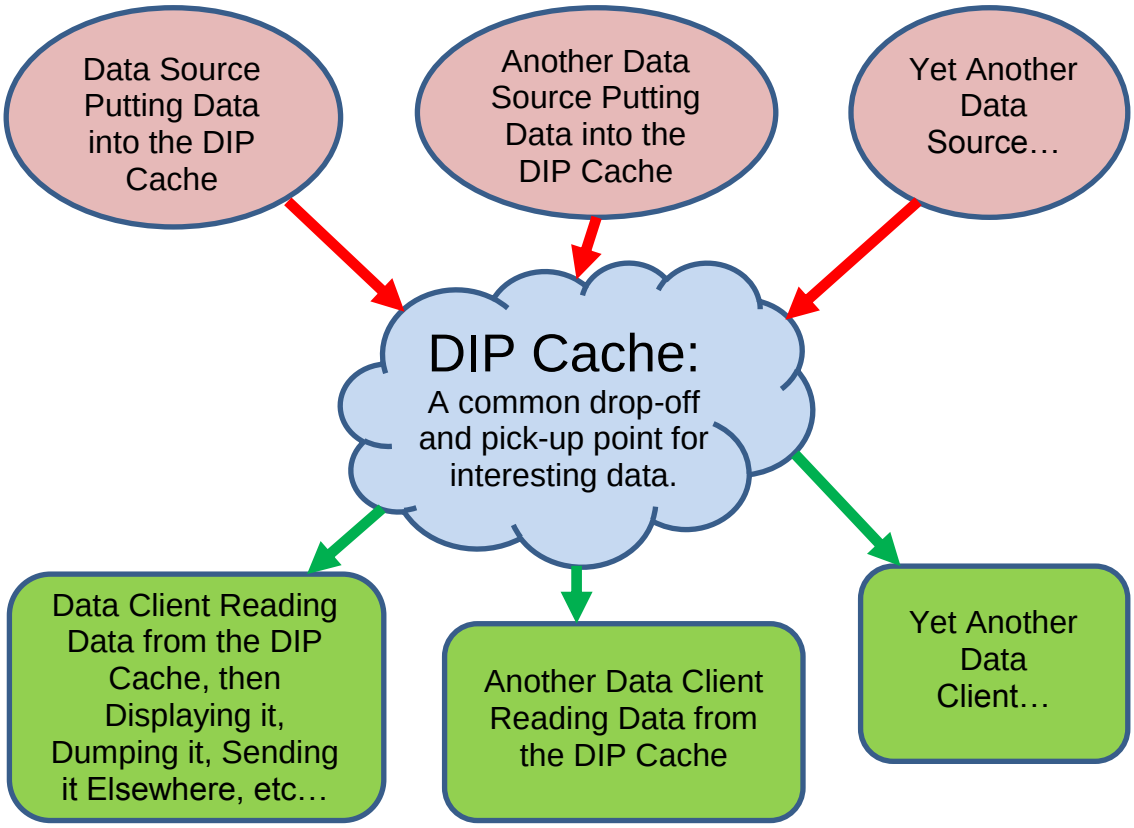


Figure 29: DIP facility dynamics

The control page is also dynamically adapted to the current state of the data job: The only buttons that are accessible are the ones that enable transition to valid states from its current state. This design is called finite state machine (FSM). Otherwise, in case that there is no active data record, the user is directed to the creation page to fill out a form with run information. Here the user chooses a calibration set to be associated with data collection and a beam type before committing to data run creation. Once the record is created, the user is automatically sent to the control page as above.

For both, online and offline data taking mode, the association of a calibration set with a data run serves two purposes: Reason one is that it is needed for later reference by the offline data analysis when processing the recorded luminosity data. The second reason is to ensure the consistency of configuration and calibration tables for a given data run: the same configuration for a given calibration has to be used for the data taking.

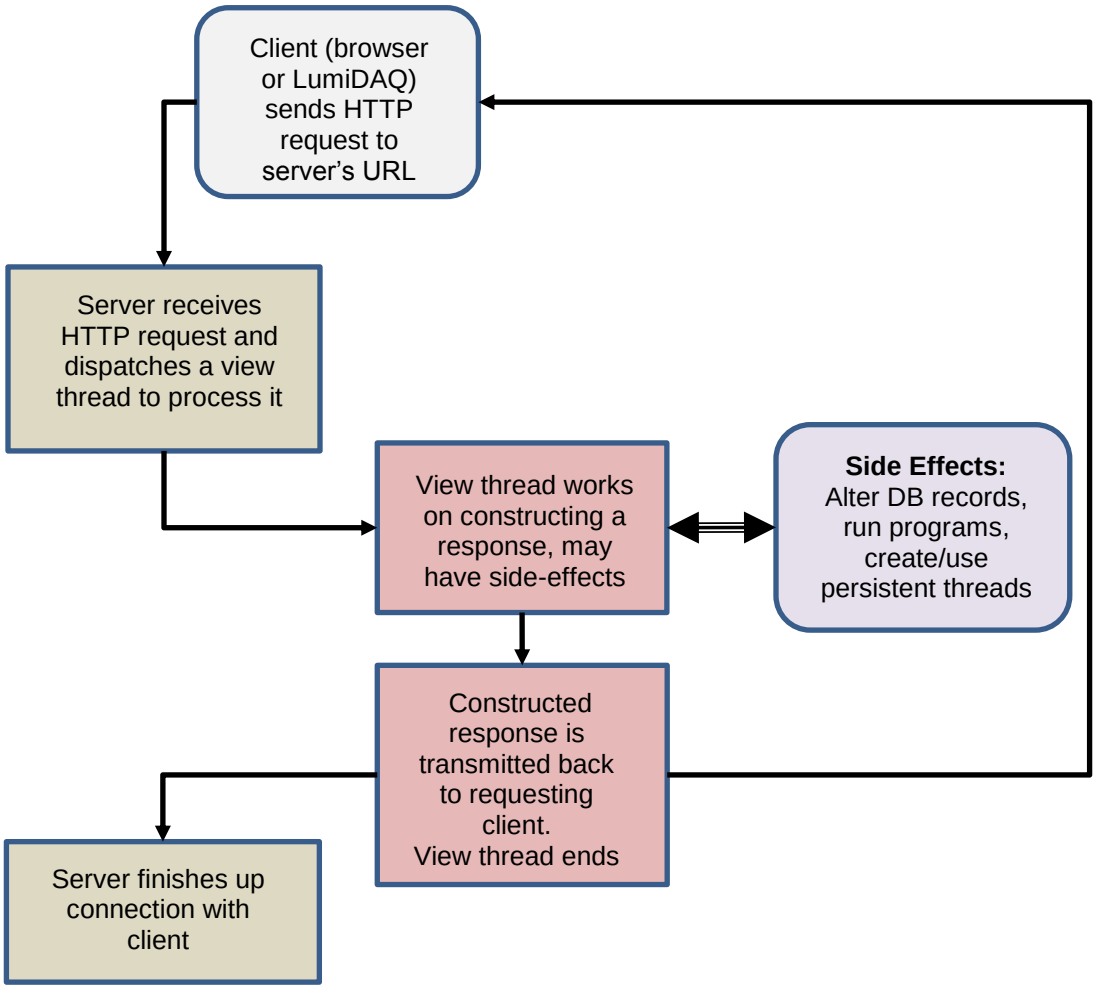


Figure 30: Theory of operation of interface server

Running Calibrations

Running calibrations is performed only during offline mode, and is available only to PLT expert operators instead of regular data shifters. Calibrations are organized into sets of one or more individual calibrations. The individual calibrations sometimes depend on other calibrations to be done first. As in case of data runs, calibration runs have a separate creation web page and control web page. The creation and control page flowcharts are displayed in Figure 35. Figure 36 continues the control page flowchart to show details for running individual calibrations. Each time calibration sets are selected, they are shown as a list of links to existing sets together with a button for creating a new one. Pressing the latter starts the creation page. Selecting one of the existing links leads the user to the control page for that set. There the user can run the set of calibrations for the first time, or re-run the set or parts of it.

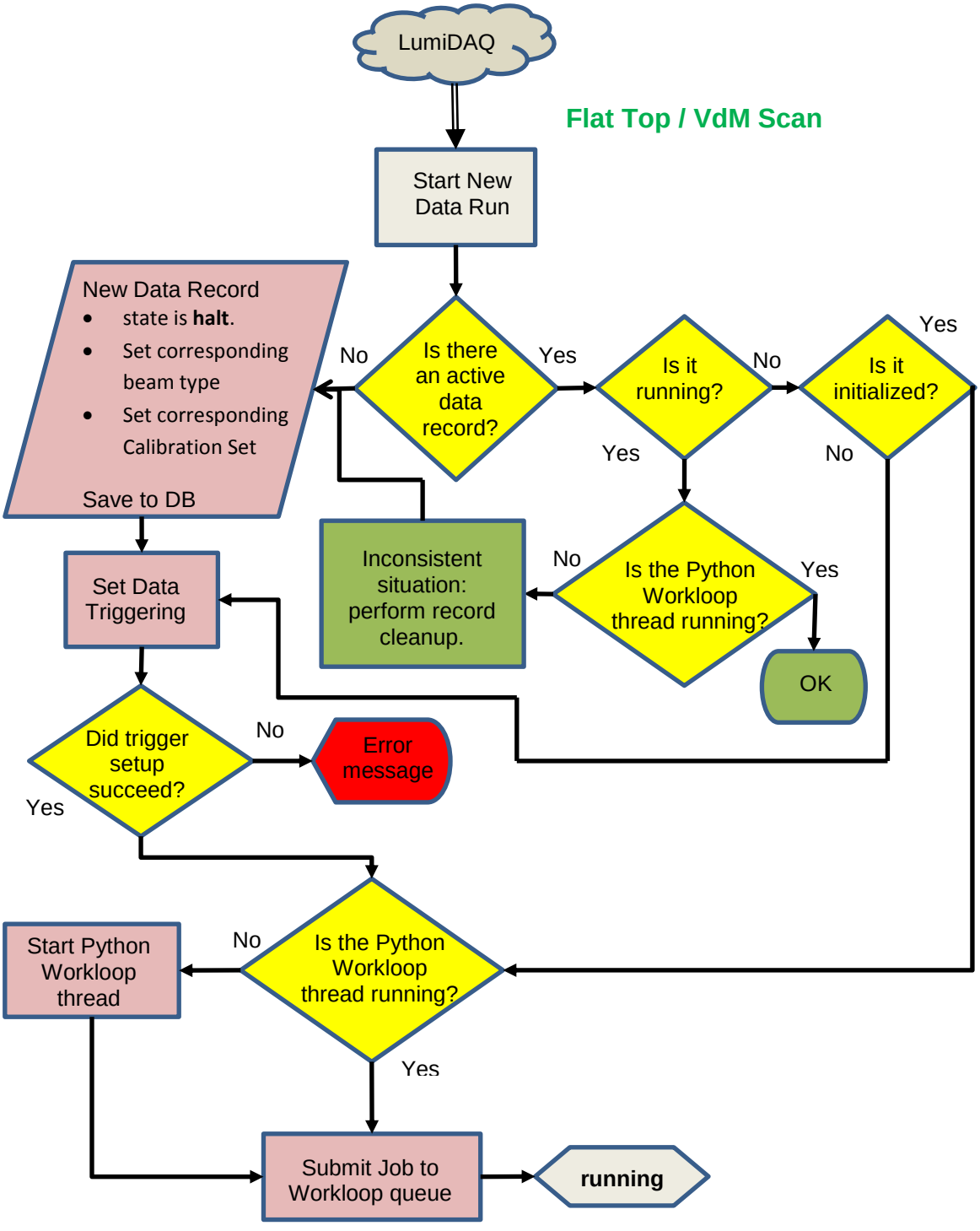


Figure 31: Online Data collection start flowchart (proposed)

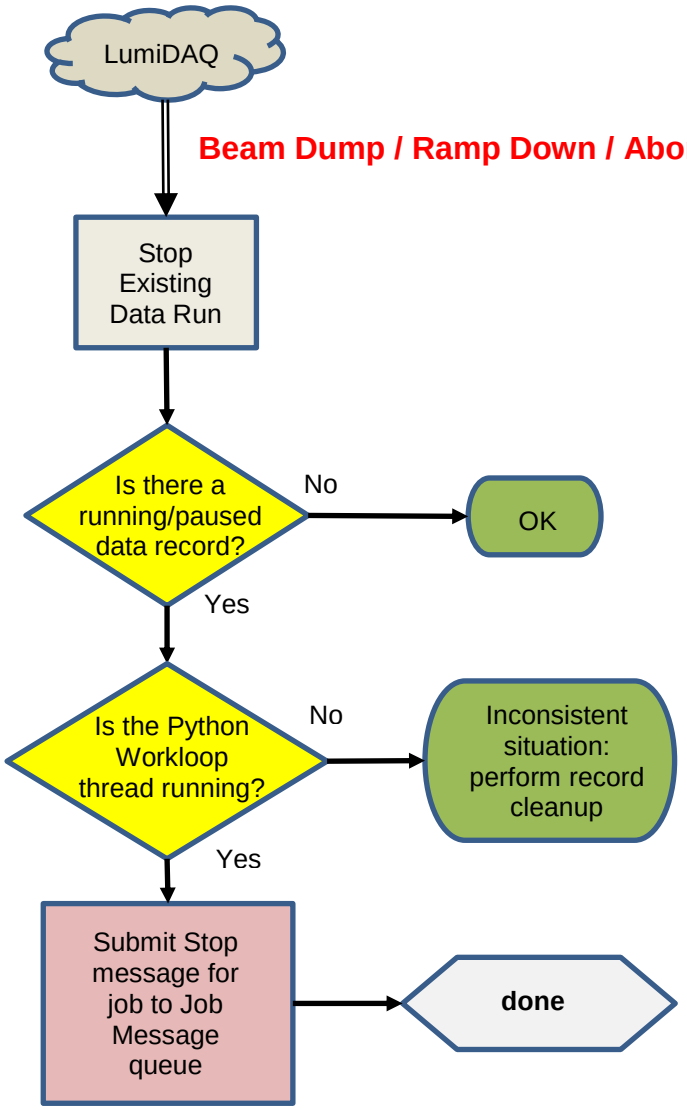


Figure 32: Online Data collection stop flowchart (proposed)

For the creation of a set, the user selects individual calibrations from a set of supported calibration types. The execution order can be changed by drag and drop action. After committing to creating the set, the user is directed to a control page. Either all calibrations in a set can run at once, or only some of them can be selected to run at that time.

Figure 37 outlines details of operation of individual calibration jobs. Only some calibrations alter configuration data as part of their calibration work, which is to alter some constants for the ROCs and for the optical communication hardware that transmits data back to the service cavern.

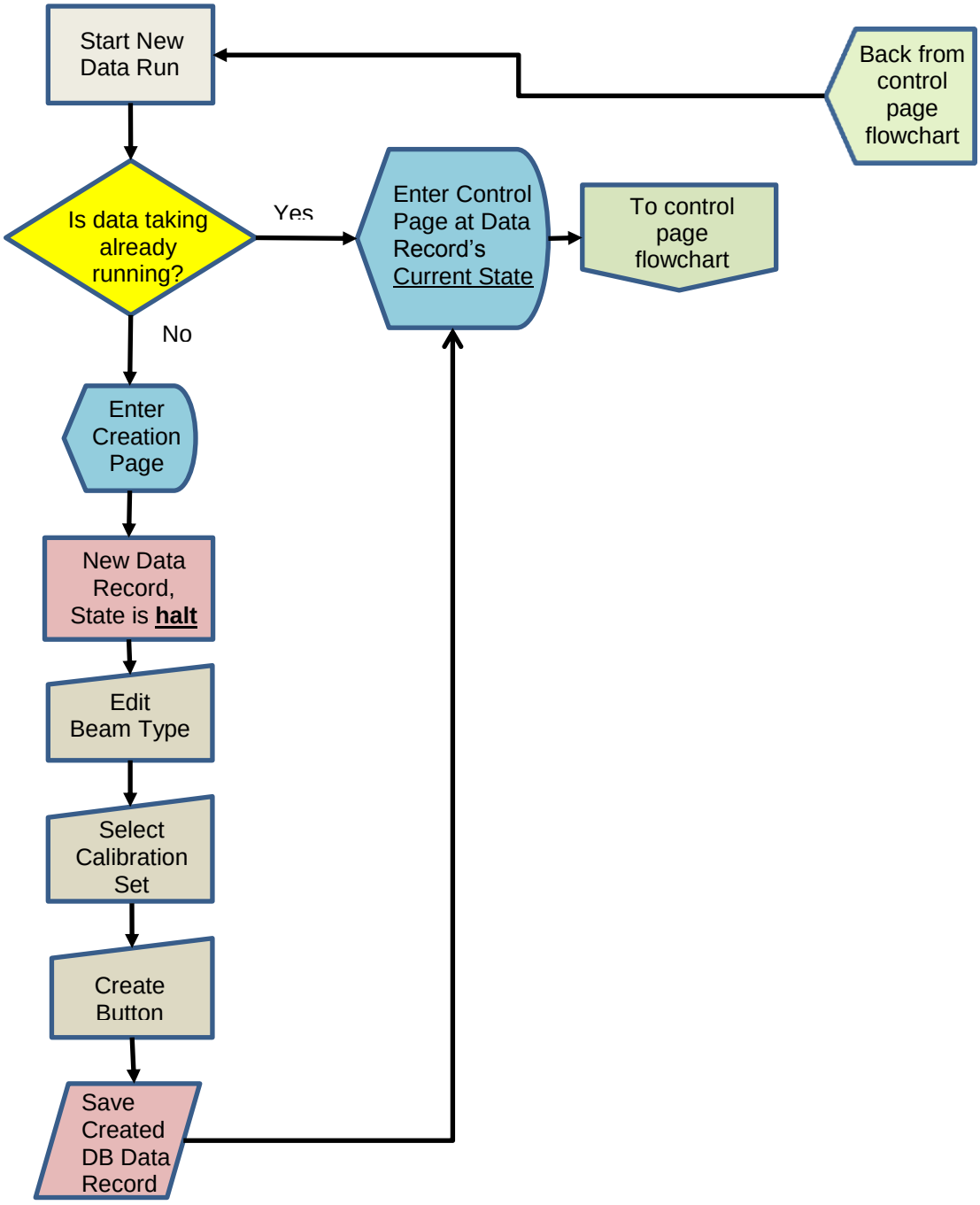


Figure 33: Offline Data collection creation page flowchart

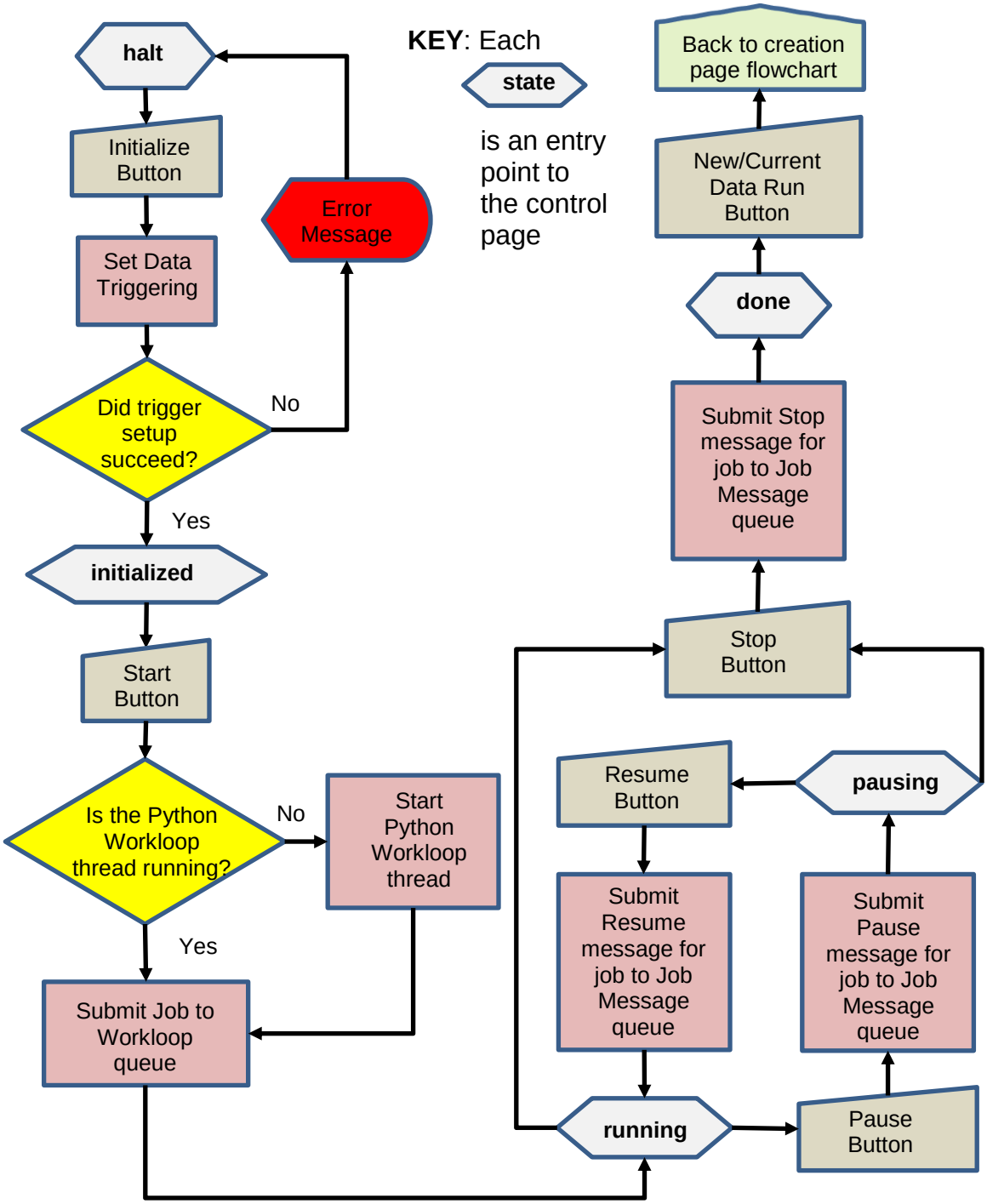


Figure 34: Offline Data collection control page flowchart

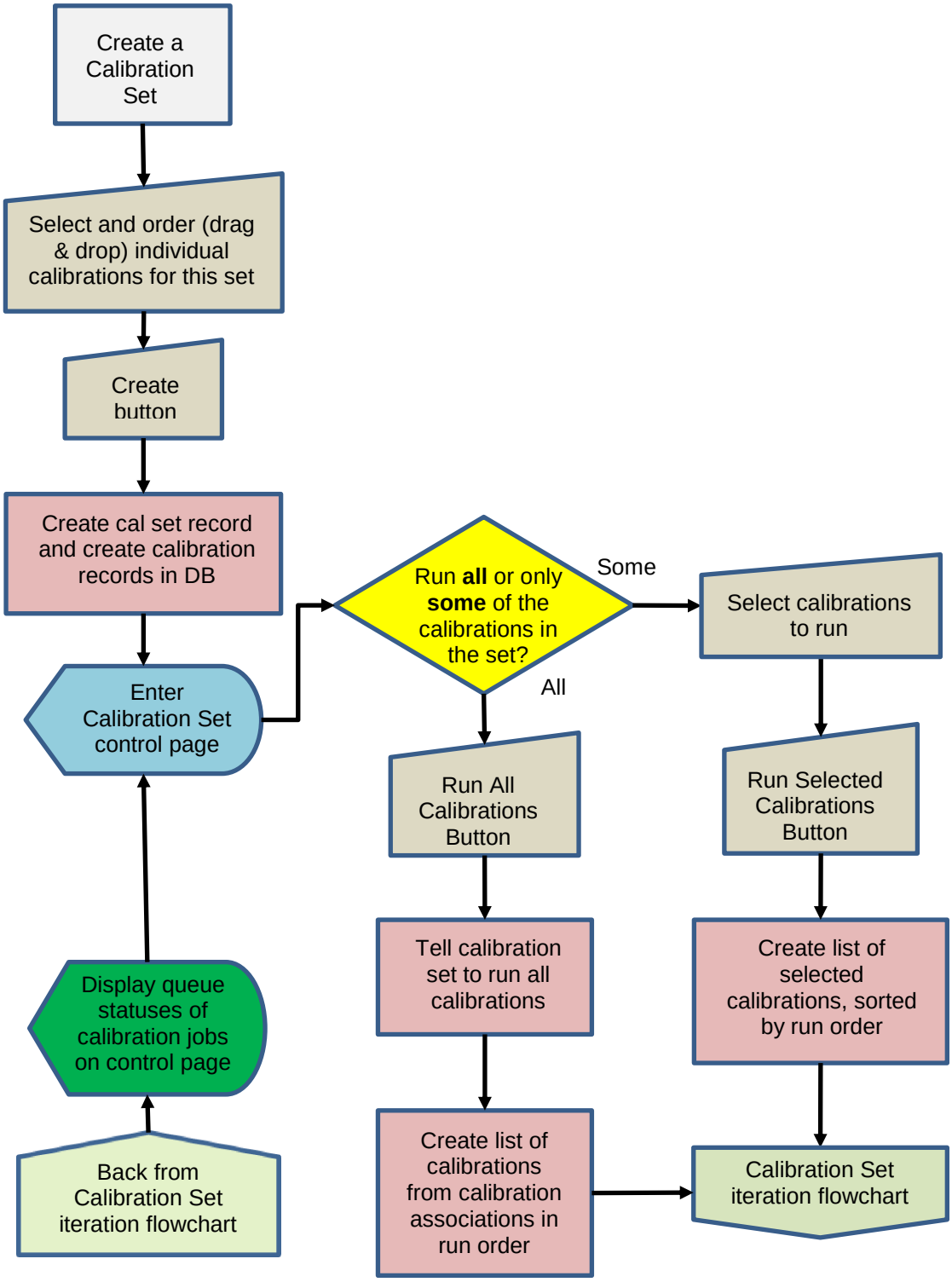


Figure 35: Calibration Set flowchart

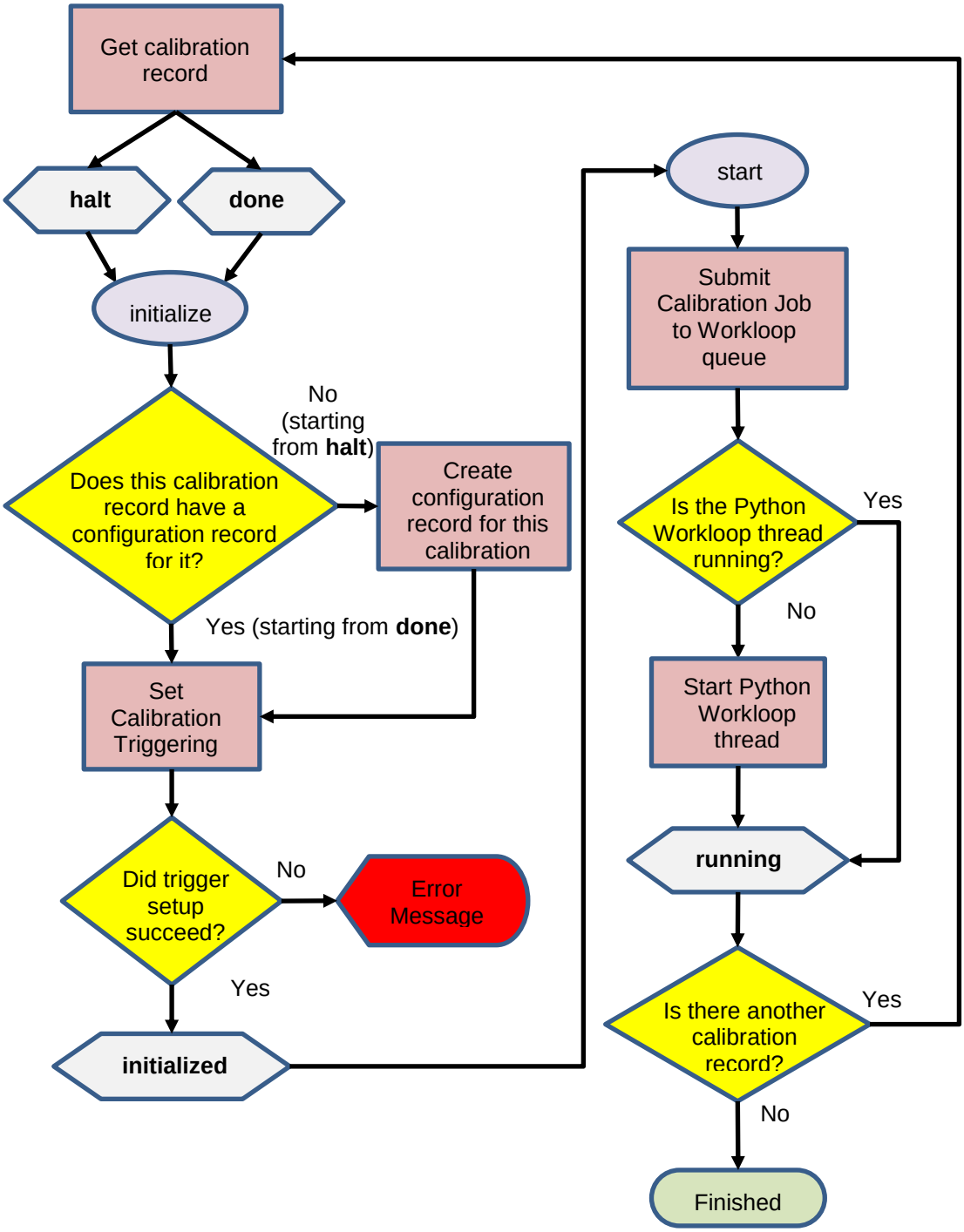


Figure 36: Calibration Set iteration flowchart for loading up the workloop queue to run or re-run calibration jobs

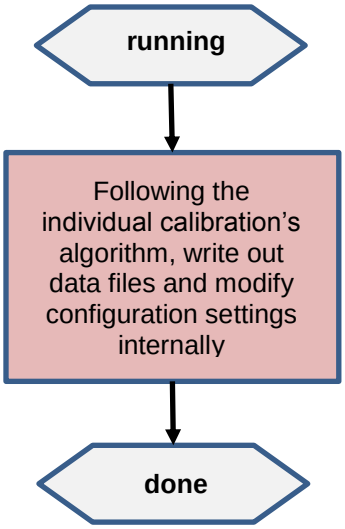


Figure 37: Calibration process flowchart

Job Work Loop Thread

The previous descriptions of data taking and calibration flowcharts refer to an important software process, the so-called Python work loop thread. It is also referred to generically as a persistent thread. This thread exists because the thread of the interface server is transient and only exists long enough to answer the browser's request. It cannot be used to support any jobs using the PLT hardware for their full duration.

The Python work loop thread does offer that support by continuing to maintain a handle on another object that keeps the job running, which is an XDAQ workloop. XDAQ is part of a large suite of software for the CMS detector in general, and is also the server software responsible for making the trigger control software available, mentioned earlier.

The Python workloop is initially spawned by the first view thread of the interface server that wants to run a job (lazy startup). It persists indefinitely while checking for any new jobs added by a view thread to the job queue (mentioned above, the job queue is visible to both the workloop thread and all view threads). There is also a job message queue used to communicate control messages (stop, pause, or resume) to the currently running job. In between any of these activities, the workloop thread simply sleeps to avoid unnecessary consumption of processor time. The full flowchart for the Python workloop is given in Figure 38.

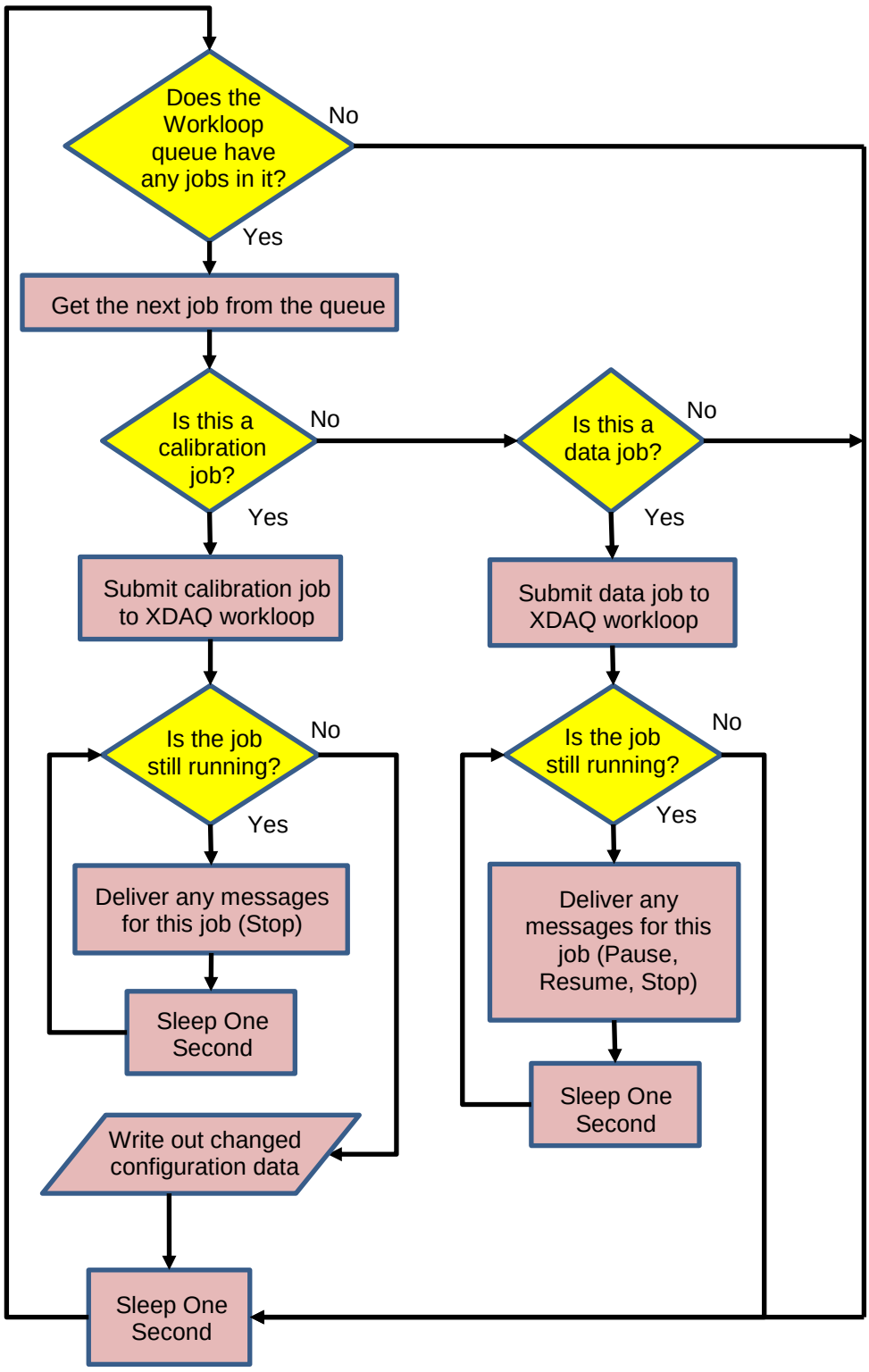


Figure 38: Python Workloop thread flowchart

Trigger Setup

The other modular block frequently mentioned in the data and calibration flowcharts is the one for trigger setup. This is powered by a co-resident server on the PLT computer, also mentioned in the introduction above. There are two pieces of software involved apart from the server: a trigger application called **TTCciControl** included with XDAQ that runs under the server, and a trigger setup client program that knows how to communicate with the trigger app via the server. The client program, called **xdaq_dialogue_to_enable.py**, belongs to the PLT browser interface.

Communicating with the trigger application in the server is a multi-step process of exchanging XML SOAP requests and replies over an http connection. The communication always starts with our trigger setup client program asking the server's trigger application what its current FSM (Finite State Machine) state is and what its current configuration is. The set of allowed states and the permitted order of transitioning between them is fixed and standardized for all trigger control applications. The configuration defines macro-procedures of low-level actions to perform when transitioning between allowed states: the configuration is used to, "fill in the blanks", of the fixed form for the state transitions. The configuration can also define extra named macro-procedures to call that are not associated with any state transitions. The configuration files are also supplied by the PLT interface.

When preparing triggers for other programs' communication with the PLT, the trigger setup client program works to ensure that the triggering mode matches what has been requested by the user interface. The interface typically requests either just a calibration mode or a data mode. The interface request is one piece of information that the client program internally associates with a state and configuration pair of properties to compare with the server app's replies when determining what to tell the server app to do. The latter is a variable amount of action:

- If the server app state and configuration already matches the interface request, then no further interaction with the server is required: the requested triggering mode is already active and the PLT is ready to use. For example, if the interface asked for data-taking triggering to be in effect, and the reply from the server app indicates that the triggers are enabled (the server app is in the **enabled** state) and that the server app's current configuration is the data-taking configuration, then the client program is already finished.
- If the server app configuration matches the interface request, but the state does not match, then more communication from the client to the server app is needed to change the state to what is needed. For example, the

request specifies data-taking triggering to be in effect, and the initial replies say the state is halted while the configuration is already set to the one for data-taking. The client will tell the server app to change its state from **halted** to **enabled**. Any intermediate states are transitioned-through along the way.

- If the configuration does not match the interface request, then the maximal amount of communication work must be done to change the configuration and end up in the **enabled** state. For example, the interface requests data-taking triggering to be in effect, and the initial replies say that the state is **enabled**, but that the configuration is the one for calibration. The server app is first told to transition from **enabled** to **halted**, transitioning through intermediate states along the way. It is then told to change to the data-taking configuration (which can only be done when the state is **halted**). After the configuration is changed, it is told to transition from **halted** back to **enabled**, also through intermediate states.

For last two cases above, it may be clarified for generality that the trigger client program takes whatever state the server app says it is in and drives it to the target state if the server app is not already at the target state. For the second case (only needing to change the state to comply with what the interface requested), the trigger client can start from any state that is not the **enabled** state and drive it to the **enabled** state. It is not limited to starting from **halted** and going to **enabled**.

Likewise for the third case, the trigger client can start from any state that is not **halted** and drive it to the **halted** state in order to change the configuration. The point to emphasize for the second and third cases is that the trigger client can fully cope with all of the states of the server app's state machine.

The full flowchart for trigger server interaction is given in Figure 39. The finite state machine diagram for the XDAQ server app **TTCciControl** is shown in Figure 40.

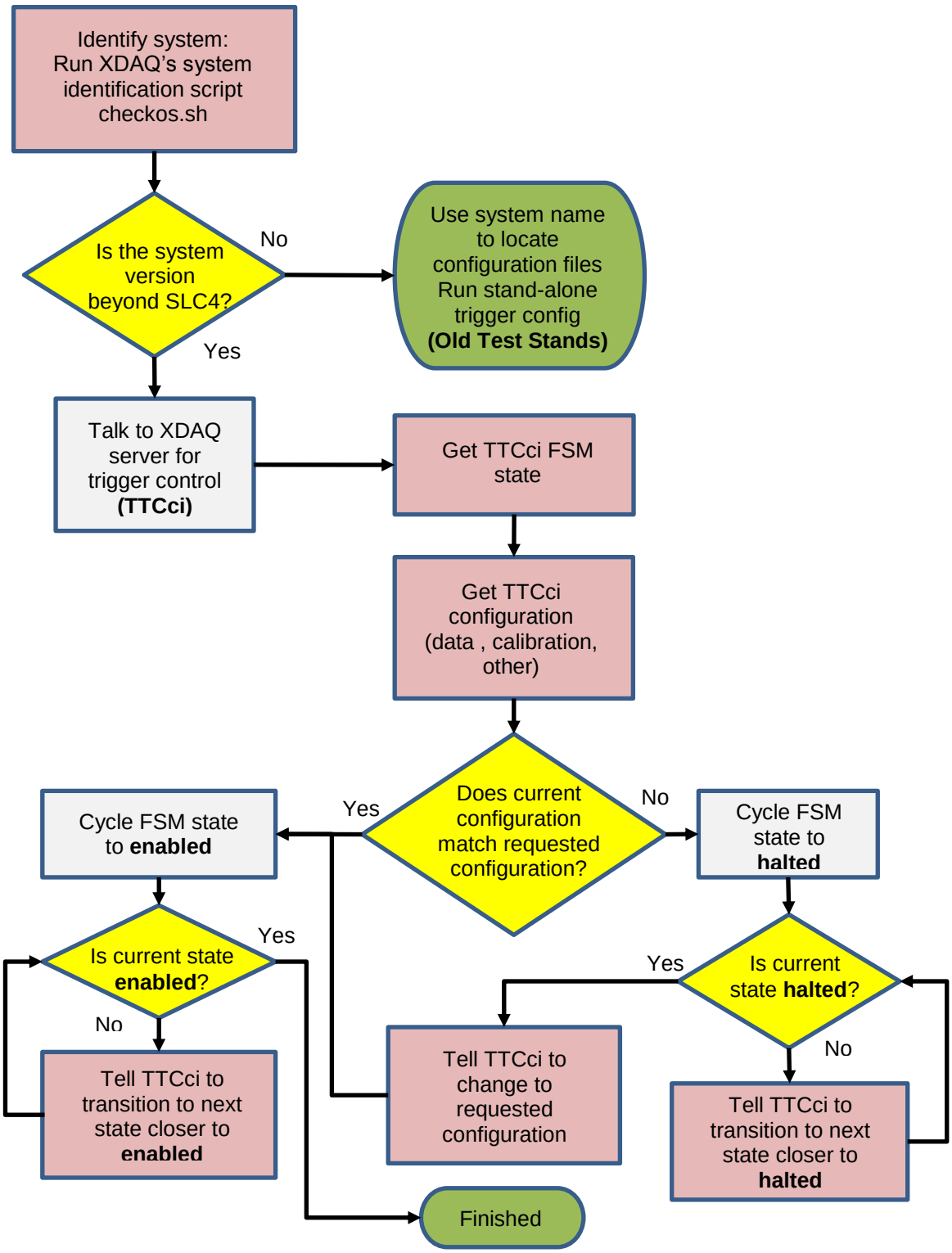


Figure 39: Trigger setup flowchart

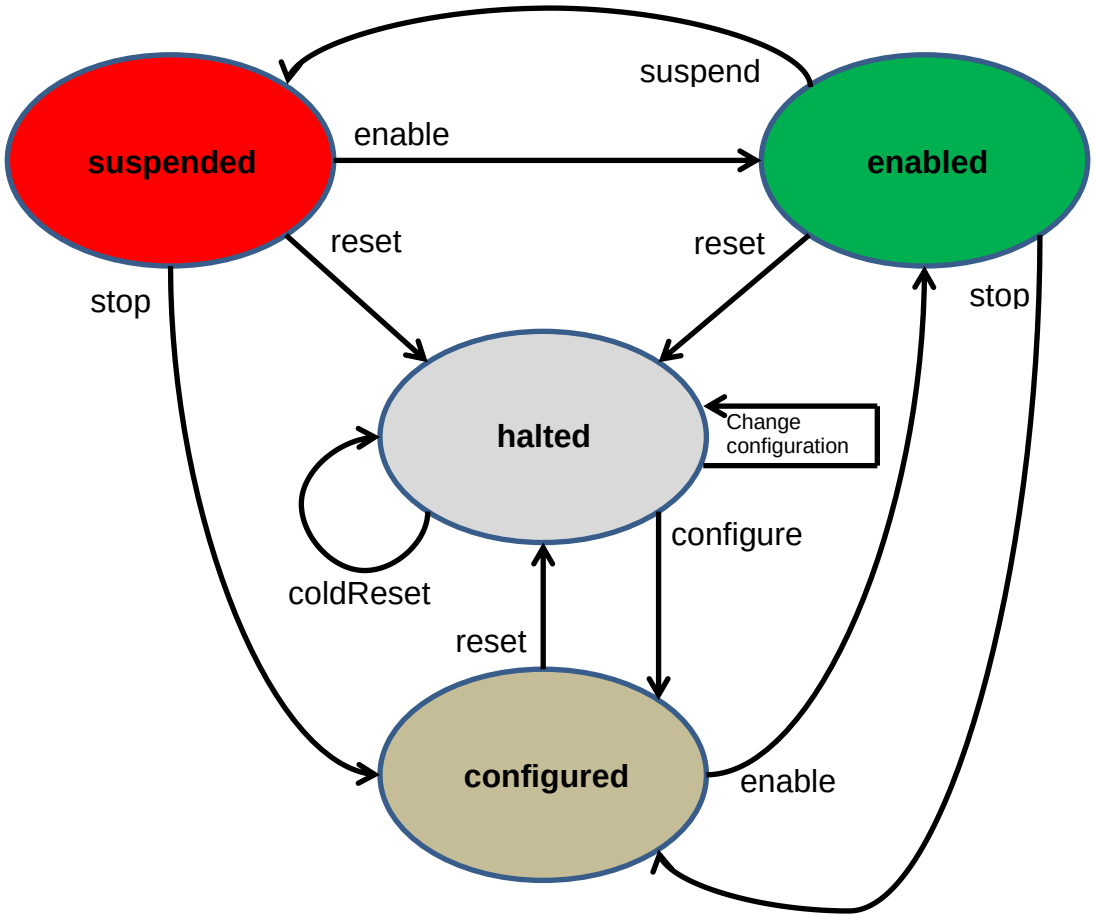


Figure 40: Finite State Machine diagram for the XDAQ TTCciControl application. The application is in the *halted* state when it first starts up.

Decoupled Data Run Monitoring

All the web pages sent to the browser from the browser interface server contain Javascript code that polls the interface server every five seconds for information about any running data jobs. The polling works just like any user-initiated action or the LumiDAQ's start/stop command to the interface server, as diagrammed in the server operation chart (Figure 30). It generates a transient view thread to do the required work, and returns a result. The result for the data monitoring thread is the reference to a visible indicator that displays the possible data-taking states: off, running, stalled, or some custom message. The full flowchart for the monitoring thread is provided in Figure 41.

What's notable about this monitoring technique is that it is decoupled from the software that is reading and recording data. It does not ask the running program to report on itself. Instead, the completely separate process of the interface server is making an independent check on the data taking process. The separate

server software can check the consistency of many things: the existence of output, whether output is growing or not, how large it is, and whether the data taking process continues to exist on the system. Multiple redundant checks are highly desirable for robustness of monitoring.

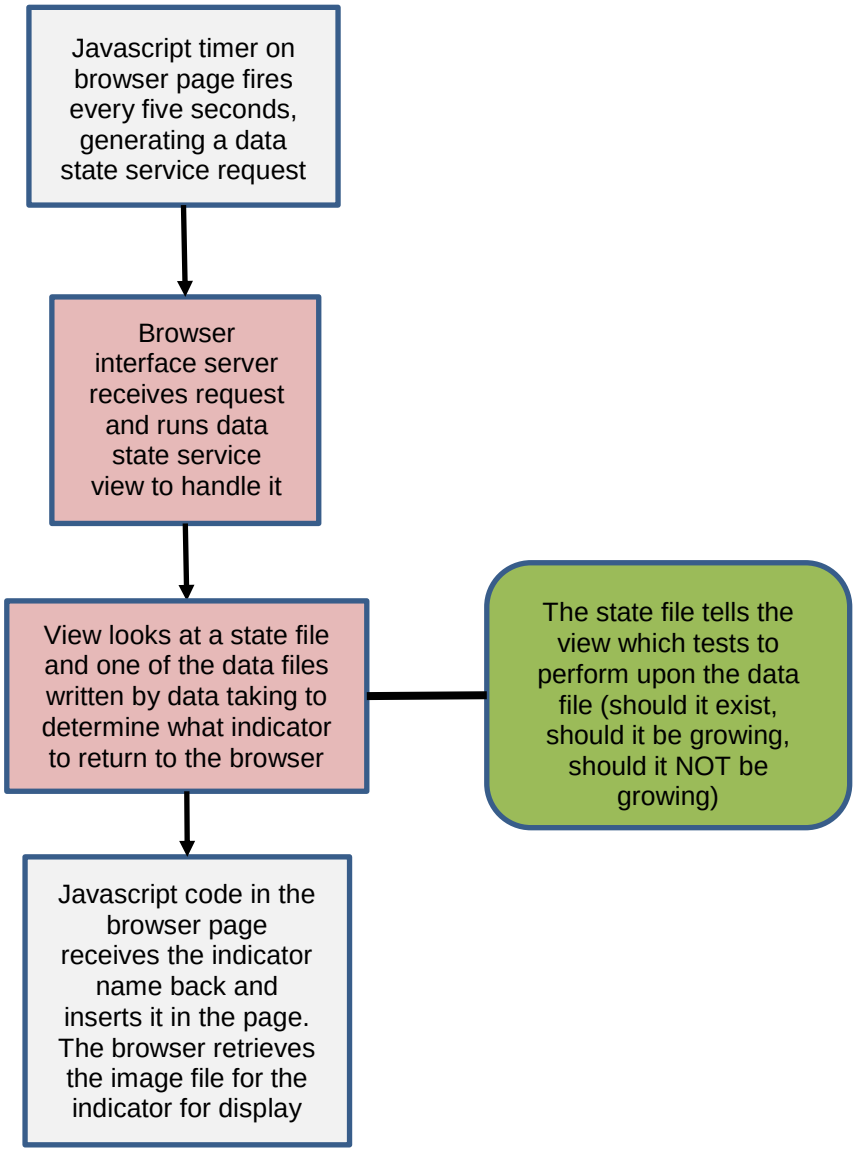


Figure 41: Flowchart for decoupled monitoring of data taking

Managing Configuration Data ("Global Runs" and sub-structure)

A side effect of running calibrations for the PLT system is the production of large amounts of modified configuration data. Configuration data is hardware settings for optimal communication from the PLT front-end electronics back to the PLT computer. It also includes hardware installation parameters of the logic boards on the PLT computer and adjustments to the PLT sensors. Only a few calibration types perform any adjustment of the communication parameters, but when they do, it is critical to pass their adjustments on to later calibrations and data runs as input to ensure that the results provide the intended benefit.

The configuration data is organized into a multi-dimensional pattern that corresponds to the inheritance of different sources of configuration revision. The major dimension is inheritance of manual revisions to configuration data. There is a set of static files (Figure 42) that comprise the data anchors of the interface system. Manual revisions of those files generally correspond to rearrangements of the PLT hardware and communication cabling. After any such revision, the changes must be loaded into the interface system by creating a new "global run". It is called this because inheritance of the new configuration data affects all subsequent calibration runs and data runs ("global" is being used as a generic term for both kinds of runs). The steps involved in creating a new global run are flowcharted in Figure 43.

The next lower dimension of configuration information management corresponds to *calibration sets*. A calibration set consists of one or more individual calibrations performed in a specific order. Each calibration set within a global run dimension refers to the same configuration data loaded by the creation of the global run: they all have a common parent that they use to begin inheritance for their calibrations. The common parent is the input for the first calibration of each set. The pattern of configurations for calibration sets and the calibrations within them is diagrammed in Figure 44.

The final, lowest dimension of configuration information is at the level of individual calibrations. An individual calibration inherits the configuration output by the preceding calibration within the same calibration set. As stated above, the inherited configuration is the one for the global run if the calibration is the first one in its set. The output configuration of the calibration becomes the input for the next calibration of the same set.

The steps involved in finding the configuration input for a calibration job and the handling of its configuration output are flowcharted in Figure 45.

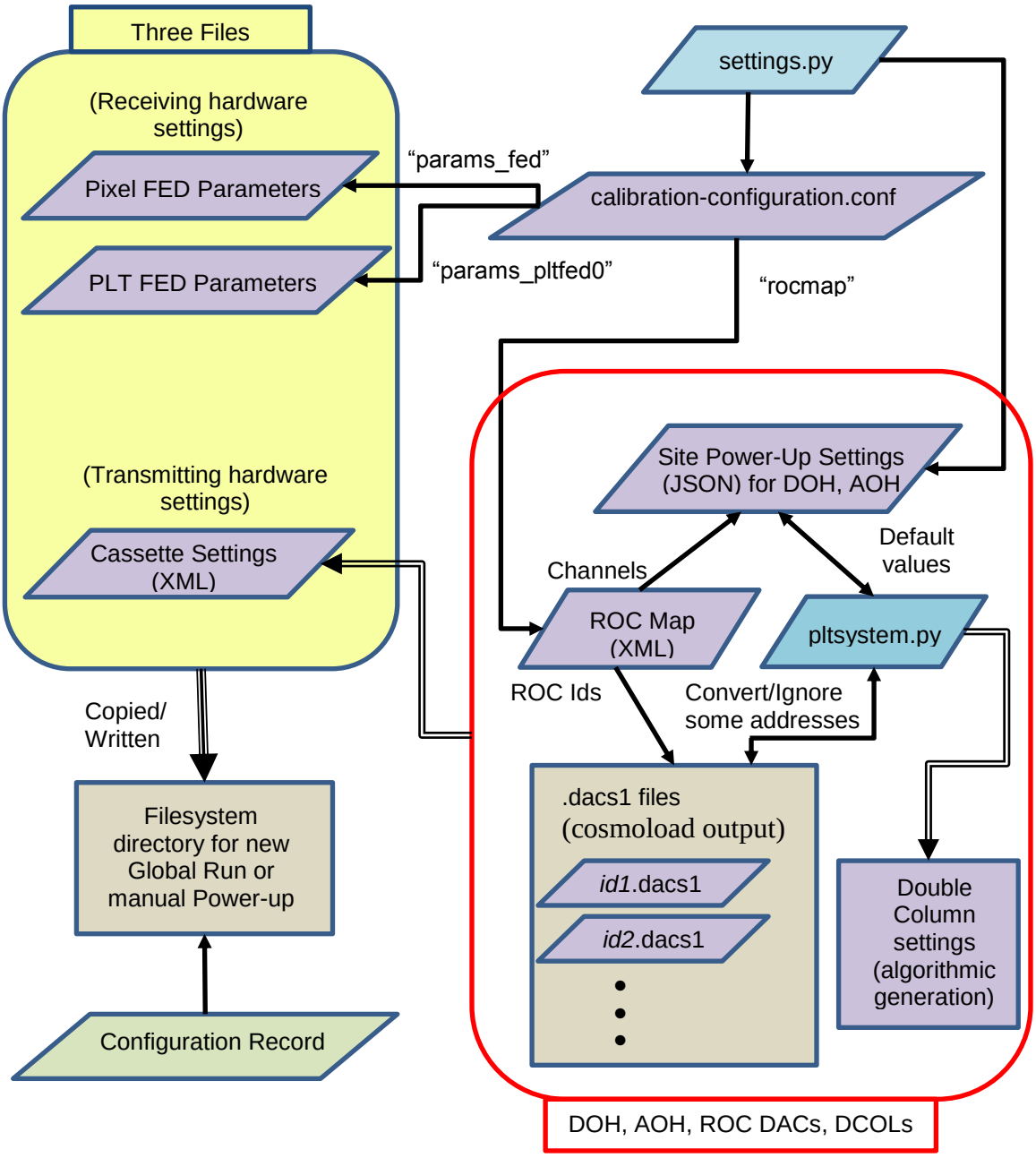


Figure 42: Configuration data anchors

Currently, there is only one configuration output for each individual calibration run. Re-running the same individual calibration replaces its previous configuration output. Future enhancement may add another second-to-last dimension to configuration data by creating separate configuration outputs for each run (re-run) of an individual calibration within a set. The utility of this is foreseen to allow multiple attempts to run a calibration, and to choose the best

one after being able to review all of the attempts. The chosen calibration's configuration output would then be arranged to become the input for the next calibration of the set.

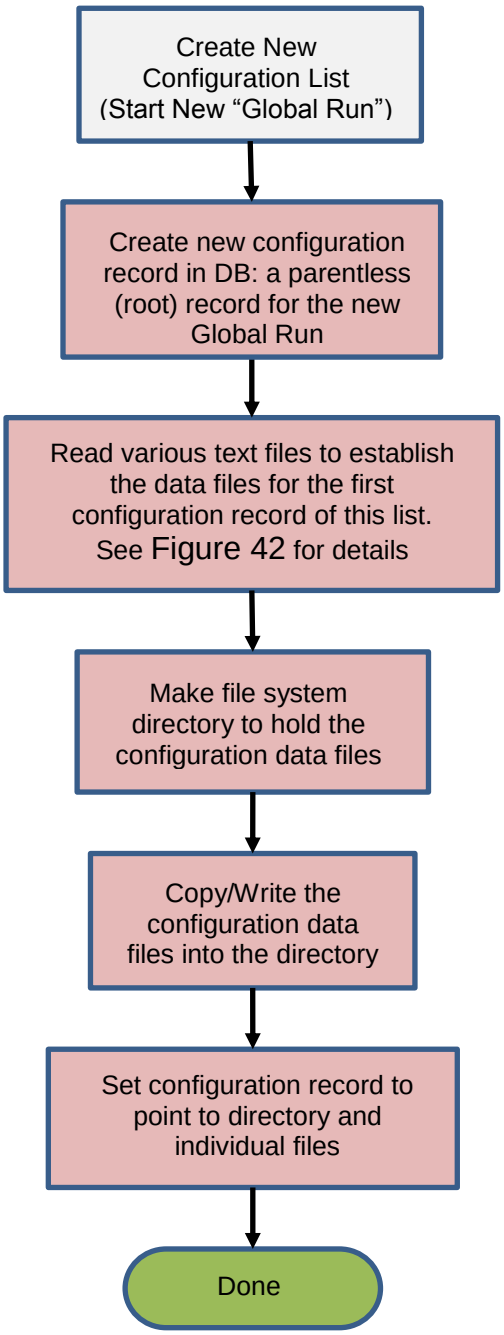


Figure 43: Configuration management flowchart: New configuration list

Data run usage of configuration information is for input only. The output of the last calibration of a calibration set is what is fed to the data taking job for configuration input. Figure 44 also shows an example of which configuration is picked up for a data run.

New global runs are empty (they have no calibration sets) when they are first created. All of the necessary calibration sets must be re-created for each new global run.

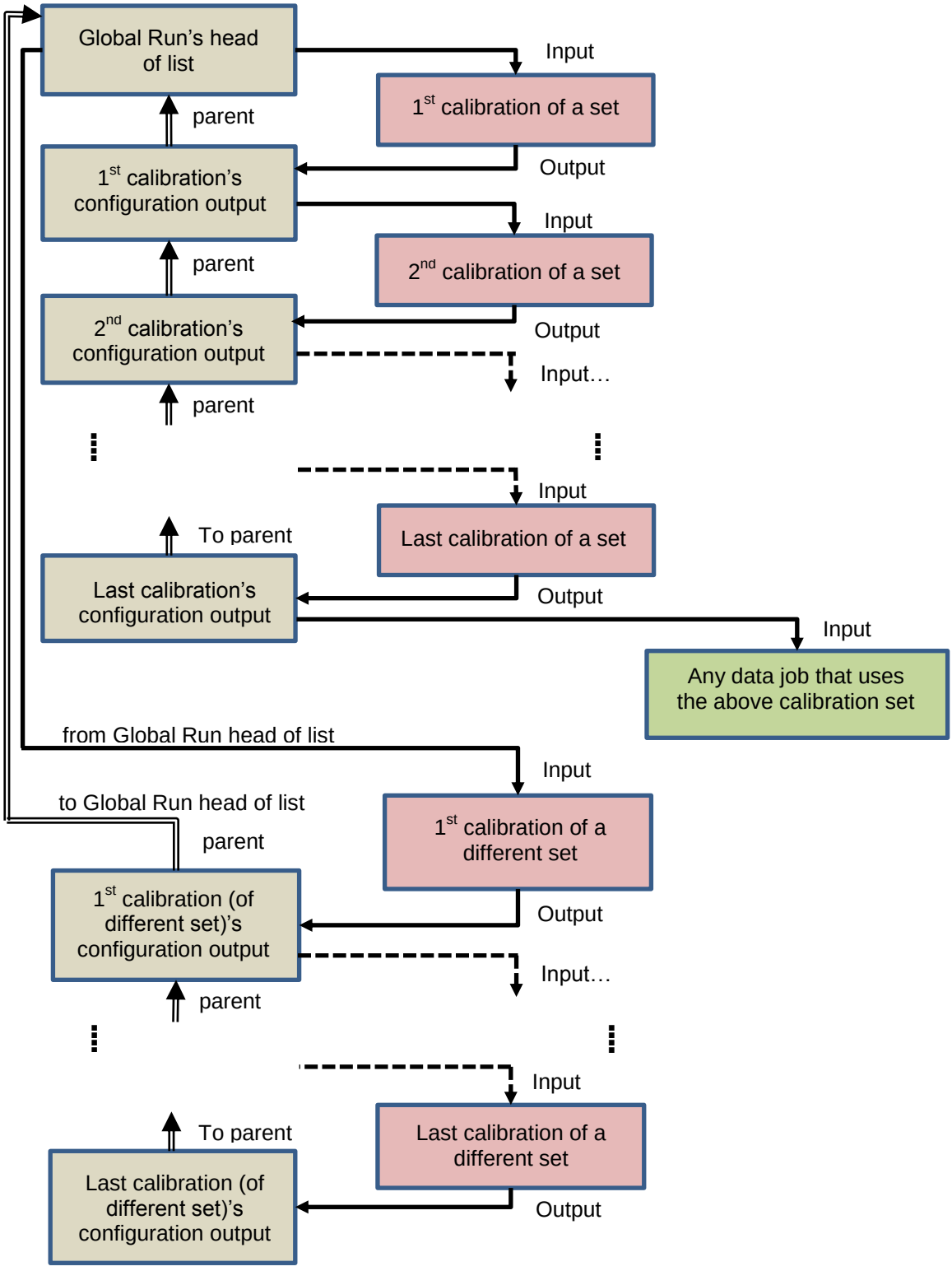


Figure 44: Configuration list and job usage diagram. See also Figure 49

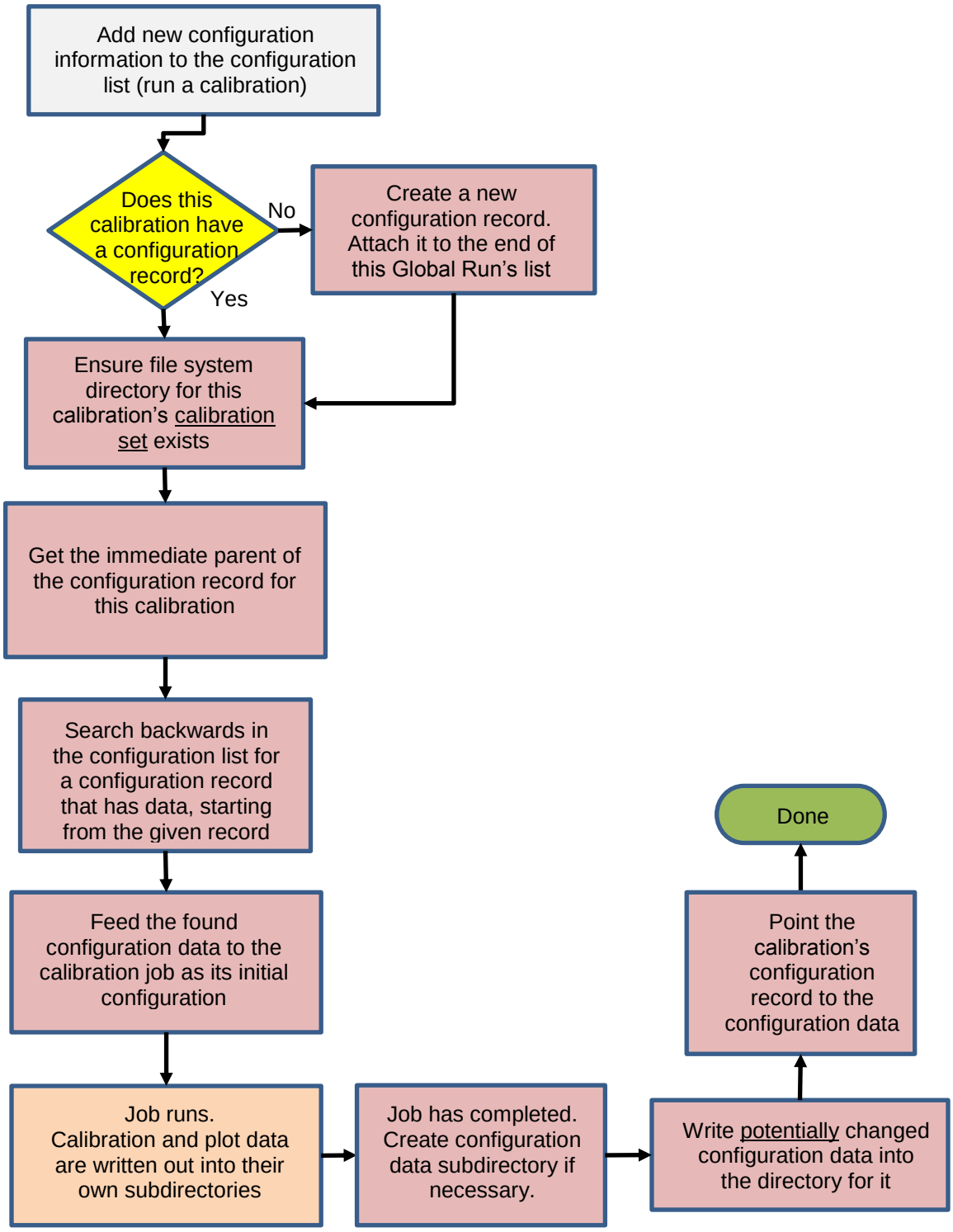


Figure 45: Configuration management flowchart: Add to configuration list

References

Hartl, C. (2013, August 6). *SOAP interface specification for TTC applications.* Retrieved from TWiki: https://twiki.cern.ch/twiki/pub/Sandbox/ChristianHartlSandboxForTTC/ttc-soap-interface.pdf

LHC Data Integration Working Group (LDIWG). (2004, January 8). *LHC Data Interchange Protocol (DIP) Definition.* Retrieved from EDMS: https://edms.cern.ch/file/457113/2/DIPDescription.doc

Mealy, G. H. (1955, September). A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal, 34*, 1045-1079. doi:10.1002/j.1538-7305.1955.tb03788.x

Moore, E. F. (1956). Gedanken-experiments on Sequential Machines. *Automata Studies, Annals of Mathematical Studies, 34*, 129-153.

Orsini, L. et al (Devel Man). (2014, September 30). *EDMS: CMS Online Software Developer's Manual.* Retrieved from EDMS: https://edms.cern.ch/document/998717

CHAPTER III
CMS PLT BROWSER INTERFACE DATABASE STRUCTURE

Abstract

The tables and relational connections between them are described, along with the data structure they are based on.

Database Establishment

The PLT browser interface uses a database to store both, permanent and transient information. Permanent information is a list of calibration results and links to the corresponding data files. It is also a list of completed data runs and links to the raw data file locations. The transient information is the run state of a calibration job and what type of calibration it is, or the run state of a data job and its identification.

Data Structure Modeled by the Database

A collection of calibration and data runs is managed as permanent information (Figure 46). These two categories of objects share common configuration information (Figure 47), as described in the last subsection of chapter II.



Figure 46: Conceptual Goal for the Interface's Database

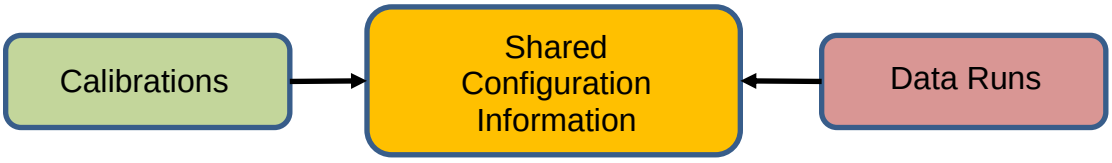


Figure 47: Common resource between calibrations and data runs

Two database tables are used to manage the multi-dimensional configuration data. One is a low-level table (“configuration”) whose records point to individual sets of files of configuration data. This table contains all the dimensions of the configuration data by virtue of the patterns of parent record references within the table. The other table is at the topmost level of PLT configuration management, called “global_run”. It is used to record which global runs in the configuration table are deactivated (superseded) ones, and which one is the currently active one. Only one global run is designated active at a time.

Figure 48 shows the two major dimensions of the configuration data in the configuration table, with their references to the global run table shown. Figure 49 illustrates the full detail of individual records within the configuration and global run tables.

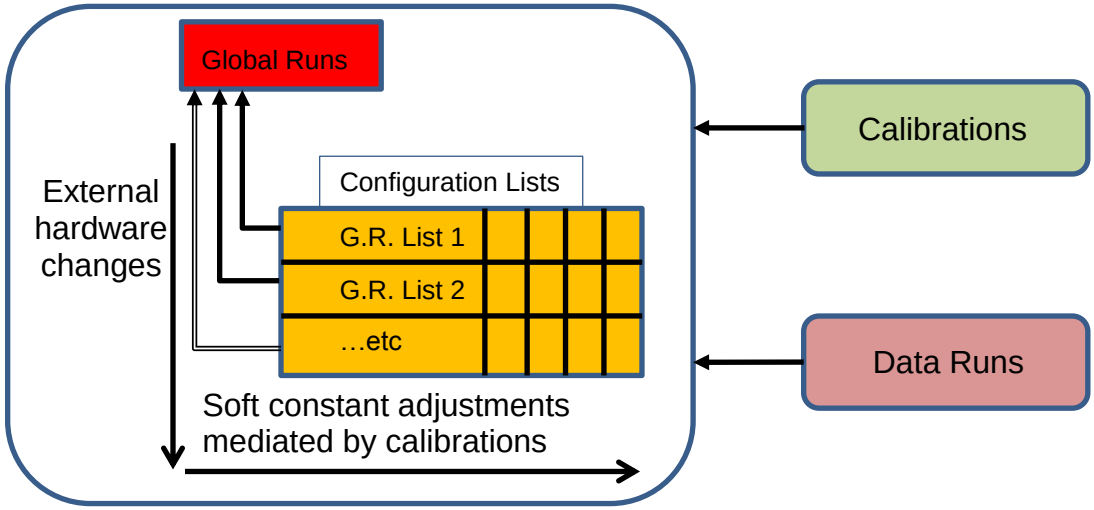


Figure 48: Necessary extra structure for configuration information

The calibrations are organized into separate sets of calibrations that are performed in a specific order with respect to each other. This must be modeled using at least two objects, individual calibrations and sets of calibrations. Good modeling practice requires a third object to represent the ordering of the individual calibrations within calibration sets, as ordering within a container is not an intrinsic property of the object being ordered. Ordering is an intermediary property between a container and any contained object. Figures Figure 50 and Figure 51 show the progressive development of this concept to its full detail.

Finally, there are many varieties of calibrations, and they all have a common core of pieces of information. In object oriented design, the individual types of calibrations are subclasses of a common calibration superclass.

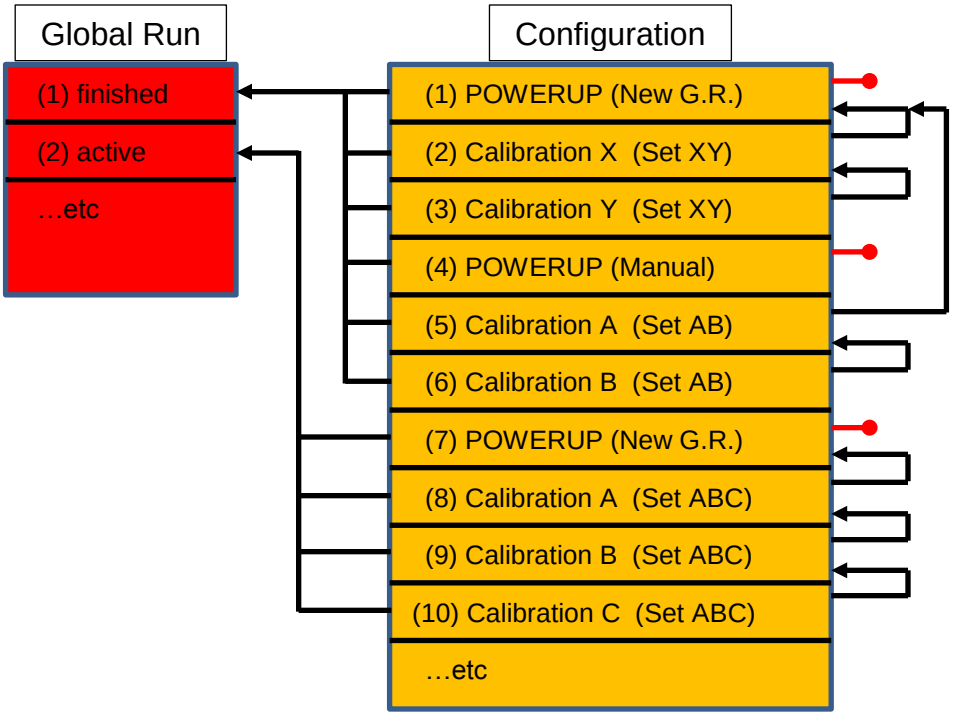


Figure 49: Global runs and configuration lists in full detail. See also Figure 44

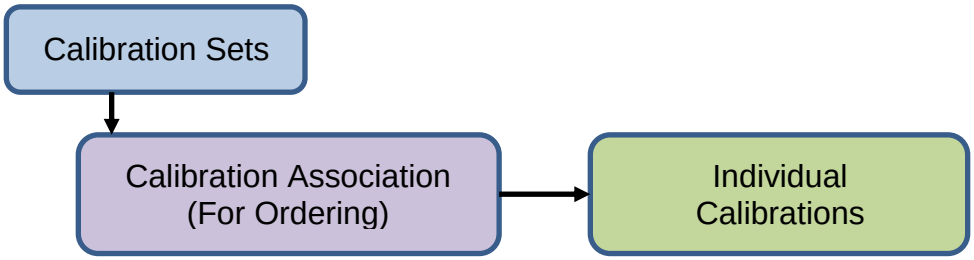


Figure 50: Extra structure for calibrations: organization into ordered sets

The total number of objects, and thus tables for our scheme of managing calibration runs and data runs is $N + 6$, where N represents the number of specific calibration types implemented as subclasses of the calibration superclass. N should be at least one to model any calibration at all. Figure 52 shows the entire collection of tables and their cross references.

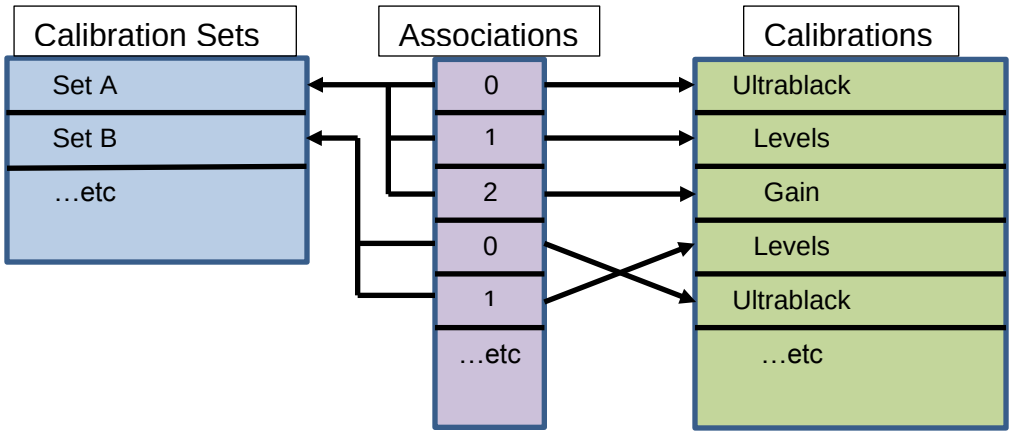


Figure 51: Full details of calibration data structure

An important feature in all database diagrams is that the directions of reference are backwards among the records from children to parents. This referencing scheme simplifies the structure of parent records: they don't have to refer to an unlimited number of child records. Instead, the child records refer to their parents. There can be an unlimited number of child records, each making a single reference to the same parent. This avoids the need for extra tables supporting multiple parent references to child records. Bottom-Up referencing is simpler than Top-Down referencing.

Storage of Bulk Data

Operation of the PLT regularly generates large amounts of data for its luminosity mission, and less regularly generates smaller amounts of data for calibration activity. Another type of data generated in tandem with calibration is the configuration data.

All forms of bulk data are stored in files on peripheral storage devices. The database stores information about the file locations (subdirectory and file names) rather than the data itself. The configuration table points to the configuration data files (up to ½ Megabyte per configuration), the calibration table points to the calibration files and any plots, and the data run table points to the luminosity data files.

The pattern of bulk storage of configuration data and calibration data is illustrated in Figure 53. Tables Table 1 through Table 7 display the field structures of the records for all the database tables.

The total data rate from the PLT is anticipated to be about 528 kilobytes per second (Bartz, et al., 2009). This accumulates to 48 Megabytes during each luminosity "section", which is 2^{20} orbits of the beam around the LHC, a period of

93.2 seconds. For an average lifetime of 10 hours for an LHC fill before an operator beam dump, a maximum data run size can achieve 18.1 Gigabytes.

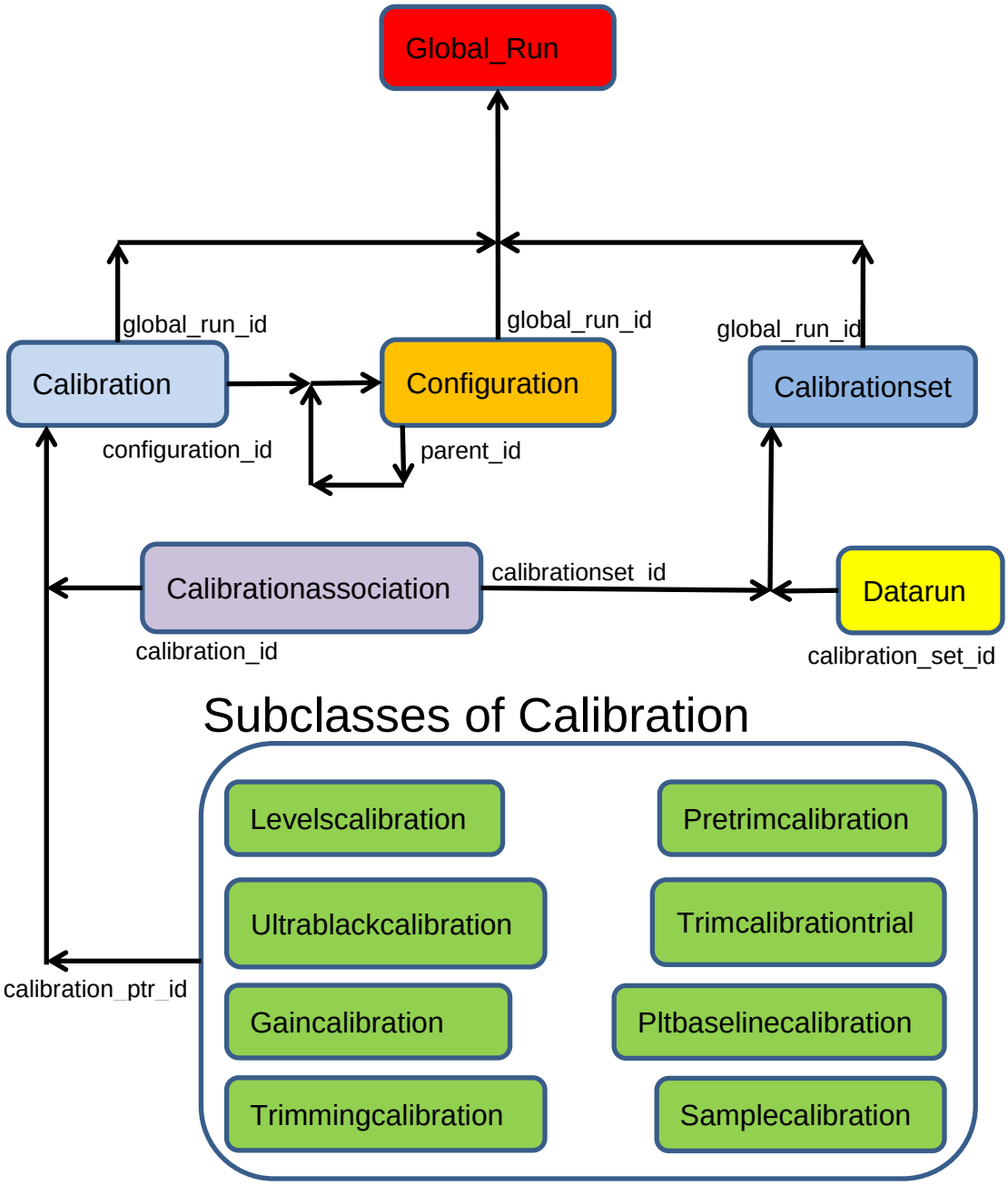


Figure 52: PLT Browser Interface Database Diagram

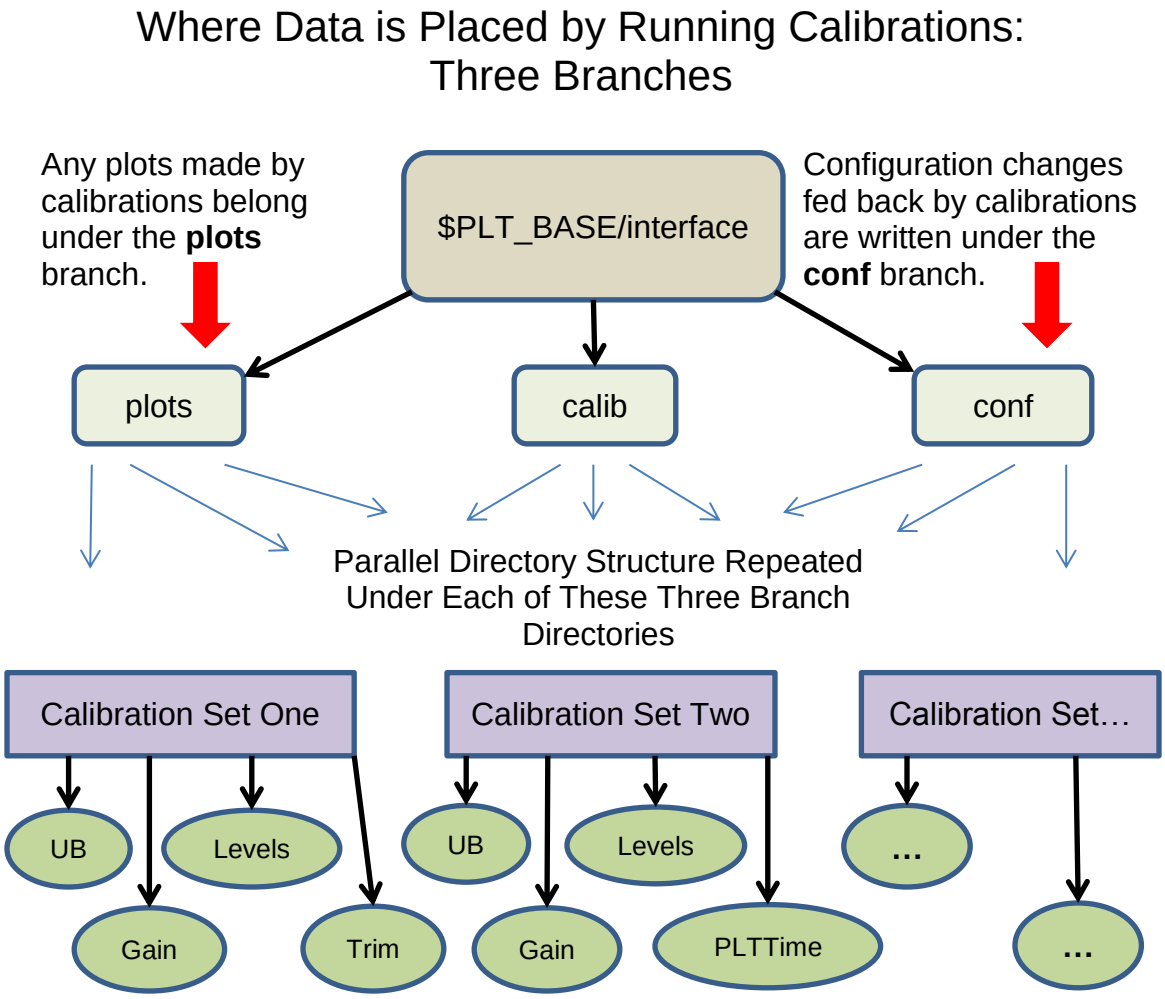


Figure 53: Storage Pattern for Calibration and Configuration Data in the Filesystem

Table 1: Record structure of Global_Run table

Field	Description
id	Primary key
added	Date stamp
updated	Date stamp
description	Text
state	“halt”, “active”, “finished”

Table 2: Record structure of Configuration table

Field	Description
id	Primary key
added	Date stamp
updated	Date stamp
parent_id	Key to another record of this table, or NULL for head-of-list record and manual powerup records
description	Text
output_dir	Calibration set file system dir name + "/" + individual calibration's file system dir name
output_files	JSON notation dictionary of configuration data file names
global_run_id	Key to a record in the Global_Run table, or NULL for manual powerup records
order	0 for head-of-list record and manual powerup records. Increases monotonically for all other data records within the same list, crossing calibration set boundaries
active	0, 1 (false, true)

Table 3: Record structure of Calibration table

Field	Description
Id	Primary key
state	"halt", "initialized", "running", "done", "finished", "terminated"
name	Name of the calibration subclass whose instance uses this record ("UltrablackCalibration", "GainCalFast", etc)
start_time	Calibration job date stamp
stop_time	Calibration job date stamp
configuration_id	Key to the configuration record (in the Configuration table) for this calibration
added	Date stamp for this record
updated	Date stamp for this record
global_run_id	Key to a record in the Global_Run table
output_files	JSON notation list of data files the calibration job says it created

Table 4: Record structure of Calibrationset table

Field	Description
id	Primary key
name	User edited name. Defaulted to creation timestamp. File system directory name is made from this
description	Unused text
global_run_id	Key to a record in the Global_Run table
added	Date stamp
updated	Date stamp

Table 5: Record structure of Calibrationassociation table

Field	Description
id	Primary key
calibration_id	Key to a record in the Calibration table
calibrationset_id	Key to a record in the Calibrationset table
order	0-based ordering of the referred-to calibration record within the referred-to calibration set

Table 6: Record structure of Datarun table

Field	Description
id	Primary key
type	Text, usually “Beam” (may be “VdM”, etc)
state	“halt”, “initialized”, “running”, “pausing”, “done”, “terminated”
start_time	Data job date stamp
stop_time	Data job date stamp
calibration_set_id	Key to a record in the Calibrationset table
added	Date stamp for this record
updated	Date stamp for this record

Table 7: Record structure for all calibration subclass tables

Field	Description
calibration_ptr_id	Key to the superclass record in the Calibration table for this calibration subclass record

References

Bartz, E., Doroshenko, J., Halyo, V., Harrop, B., Hits, D., Macpherson, A., . . . Stone, R. (2009). A CMS Luminosity Monitor Using Single-Crystal CVD Diamond Pixel Detectors. *Journal of Instrumentation (JINST)*, 4-6.

CHAPTER IV
INSTALLING AND BUILDING THE BROWSER INTERFACE

Abstract

A narrative of the supporting software needed to build and operate the browser interface software.

Introduction

SLC4 is past its end of life, as of Feb 29, 2012. Installation on SLC5 and SLC6 is fully supported.

Installing and setting up the PLT Browser Interface on an SLC5/SLC6 system

These instructions begin from the point of having an SLC5 or SLC6 system set up.

It is recommended to have a user account on the system instead of installing the software as the root user. This introduces a layer of safety for the system as compared to applying actions from the root account, where unintended changes can have far-ranging to unrecoverable consequences. Root privilege, when required to execute system commands, may be obtained on a command-by-command basis by adding one's user id to the **sudoers** file. Then the **sudo** command permits root privilege for individual commands. The password to enter when prompted by **sudo** is the user account password, not the root account one. All of the command examples in this document shall show **sudo** used for commands that require system privilege, with exceptions explicitly marked.

To add oneself to the **sudoers** file, one must be root in the first place. After accomplishing that, run `visudo`. Use the down arrow key to get to the place in the file where there are comments about the COMMANDS section and there is a line for the root account itself:

```
root ALL=(ALL) ALL
```

At this point, insert a new line into the file (press `i` to insert text) after the root line above. Type the same stuff as seen for root, but substitute your own user id for `root`:

```
myuserid ALL=(ALL) ALL
```

Press return at the end of the line above, then save and exit in vi style (after pressing the escape key to stop entering data, type `:wq` and press return). Now your user id should be all set up to use **sudo**.

The first set of required major software packages are:

- CMSOS software in /opt/cmssw
- XDAQ software in /opt/xdaq
- ROOT installed separately from CMSOS software in /opt/root, with version matching that of ROOT in CMSOS
- Subversion (comes with os package, Development Tools group)

CMSOS may be installed according to instructions found in the TWiki page (Wissing, 2013), which uses **apt-get** to do the installation work. Follow the instructions to do an LCG-style installation, and start by setting VO_CMS_SW_DIR to /opt/cmssw. Since /opt is owned by root, the **sudo** command shall have to be used so the **mkdir** command in the TWiki instructions can succeed. The **mkdir** command referred to is the one under both shell versions of, "**Recipe for the impatient**", where the instructions for the, "Only once to initialize:", list of commands occurs. After performing the **mkdir** command under **sudo**, one must change the ownership and group of the /opt/cmssw directory to one's own user id and group, also using the **sudo** command for permission. Thereafter, one may run all the remaining commands as one's self. Note also that the narrative describes setting SCRAM_ARCH to something particular for SLC5: this must be adjusted for what is available for SLC6 systems.

After the "impatient" instructions for the once-only initializations with the shell one is using, one must jump down into the full documentation under the, "**Initialize the software area**", subsection. Resume at, "**Once bootstrap.sh is successful**". After taking care of the "fake package" issue documented, and doing "`apt-get update`" as advised further down, one is ready to perform other installations into the /opt/cmssw area.

Note also that an actual installation of a version of the CMSSW software package is not absolutely necessary. The actual dependencies sought by the interface software are the external+boost+*version_number*, external+python+*version_number*, and the external+uuid+*version_number* packages. One may need to explicitly install some or all of these packages even with a CMSSW package. The choices of each to install may be listed by:

```
$ apt-cache pkgnames | grep boost
```

…and:

```
$ apt-cache pkgnames | grep python
```

…and:

```
$ apt-cache pkgnames | grep uuid
```

It is generally safe to choose the highest version of **boost** shown and the highest 2.x version of **python** shown. Use the, "`apt-get install` *packagename*" command with each choice made, just as with the "fake package(s)". Another dependency, external+gcc+*version_number* should already be present after the bootstrap procedure for the **apt** environment.

The XDAQ software may be installed using the instructions found in the CMS OS user's manual documents on the EDMS page referred by (Orsini, L. et al (User Man), 2014), which uses **yum** to do the installation work. Section 3.6.1 ("*Setup a YUM client*") describes how to set up yum to do the installation. Older versions of the manual (pre-1.9.1) have the information in section 3.5.1. The sample data for the `yum.repo` file entries refer to SLC6 64 bit versions, and the older manual versions refer to SLC4 for 32 bits. Another web page (Orsini, L. et al (The CMS Experiment), 2014) provides guidance for valid combinations of XDAQ version, OS version, and 32/64 bit variation (under the section titled **Releases**). For example, version 11 for SLC5 for 64 bits may be retrieved by putting 11 for the occurrences of <n> in the example .repo file entries, and slc5x in place of the slc6x (or older slc4X) occurrences. The older manual examples also require the substitution of x86_64 in place of the i386 occurrences to retrieve the 64 bit software for SLC5.

Copying the manually created `xdaq.repo` file to the recommended place in the system requires elevated privilege (using **sudo**):

```
$ sudo cp –p xdaq.repo /etc/yum.repos.d
```

In addition to the package groups to install with the example commands in the manual, one more package group must be added: **hardware_worksuite**. It is critical for building the interface's C++ software, as it supplies **hal** include files for the VME and CAEN hardware interface classes. Everything may be combined in one command (ignore the line break caused by this document's margins):

```
$ sudo yum groupinstall extern_coretools coretools
extern_powerpack powerpack hardware_worksuite
```

For XDAQ version 12 on SLC6, there is a system library that XDAQ uses that may not be present after OS installation. That library is **numa**, for Non-Uniform Memory Access. It may be installed via:

```
$ sudo yum install numactl-devel
```

Root may be installed by visiting the venerable Root software site (The ROOT Team, 2014) and retrieving the tarball for the appropriate version. If a full installation of CMSSW already exists on the target system, listing /opt/cmssw/*SCRAM_ARCH*/lcg/root shall reveal the version number to match (where *SCRAM_ARCH* represents the actual installation directory present on the system, such as slc5_amd64_gcc462). Otherwise, one is free to get the most recent production version of root. Unpack the tarball and follow the instructions in the README/INSTALL file to configure with the correct architecture and specifying the *--prefix* option to be /opt/root (for the **configure** command) before 'make' and 'sudo make install'.

Root must be installed separately so that various C++ modules of the interface can successfully build and link against it. Such has not been possible with ROOT embedded in CMSOS.

Subversion should already be on the system, as it is a part of the base O.S. installation. Its presence on the system may be checked by:

```
$ ls –l /usr/bin/svn
```

If the **ls** command above does not find a file, then subversion may be installed by:

```
$ sudo yum install subversion
```

The Additional Software

With the above pre-requisites in place, one needs to get the following additional software that might not have been installed with the creation of the system from the installation medium:

- MySQL (comes with os package, MySQL DataBase group)
- scons (under dag package)
- virtualenv (obtained externally)

The data base used by the PLT Browser Interface is MySQL. It comes in several pieces for both supporting the MySQL server and talking to it from Python. The following **yum** commands install all the pieces that depend upon each other.

```
$ sudo yum install mysql
$ sudo yum install perl-dbd-mysql
$ sudo yum install mysql-server
$ sudo yum install mysql-devel
$ sudo yum install mx
$ sudo yum install mysql-python
```

Scons (The SCons Foundation, 2014) is software for building software, a replacement for **make** that is coded in Python. It is what we use to build the C++ software for the PLT Interface. Getting it is simpler than getting MySQL was:

```
$ sudo yum install scons
```

Virtualenv (Bicking, Ian; The Open Planning Project; PyPA, 2014) is a Python tool for creating isolated Python-version environments. It is necessary to manage the Python-version environment for Python applications that are version-sensitive. Our browser interface and the framework it operates under (**Django**) are coded in Python. We use virtualenv to keep the version of Python that runs our interface consistent.

Virtualenv is not provided by the yum repository of linuxsoft.cern.ch, so we obtain it directly from its web site. Here we are following the instructions to use virtualenv locally from source. The version shown used here is a minimum required version of virtualenv due to the version of another program that virtualenv downloads and installs, called **pip**. A later step in our process that loads our virtual environment, called **bootstrap**, has dependencies upon the version of **pip**.

```
$ curl -O
https://pypi.python.org/packages/source/v/virtualenv/virtualenv-
1.11.6.tar.gz
$ tar xvfz virtualenv-1.11.6.tar.gz
```

If the tar command above has a problem with the downloaded file, and the file size is abnormally small (about 28k), then the URL has been mis-typed (the downloaded file is actually an HTML page that says that the page you're looking for could not be found). The option to **curl** is a capital 'O', not a zero, and the URL has 'packages' plural and 'source' singular, for example.

After successful download and **tar** extraction, the virtualenv tool and its supporting retinue are now in the subdirectory called **virtualenv-1.11.6**. We will return to it shortly.

Our next action is to get the actual source for the PLT Interface by checking it out from its subversion repository. This downloads it into a local working copy. The command is:

```
$ svn checkout
svn+ssh://youruserid@svn.cern.ch/reps/cmsplt/trunk cmsplt
```

The *youruserid* part should be one's own CERN single sign-on user id, such as for logging in to lxplus, Indico, and Vidyo. The result of the svn command should be a local copy of the trunk version of **cmsplt** in a subdirectory of the same name.

```
$ cd cmsplt
$ . setenv.sh
```

The sourcing of the setenv.sh shell script sets our environment so that we may build the C++ software and use the correct version of Python for the Python software, plus have access to ROOT, XDAQ, and the CMS OS software . On SLC5 systems, the version of Python that comes with the system is 2.4.3, which is a little too primitive for the Django (Django Software Foundation, 2014) part of the PLT Interface (the Browser part). The `setenv.sh` script sets us up to use the version of Python which comes with the CMS OS software, which is version 2.6.4. That is more than sufficient for Django, which requires at least 2.5. The CMS OS version of Python will also be important when we return to the usage of virtualenv.

Continuing the commands for building the PLT interface:

```
$ cd interface
$ scons –j 2 >scons.log 2>&1
```

Scons should find the file named **SConstruct** in the `interface` directory and build everything it is told to build. The output redirection shown is purely optional; simply **less** the `scons.log` file to look at everything it did or **tail** the file instead to get a quick answer to whether or not everything succeeded. The last line should say, "`scons: done building targets.`", if all is well.

We are now ready to continue our involvement with virtualenv where we left off. The virtual environment will be created under `/opt` in the filesystem. That will require root privilege. At this point it must be mentioned that there is a strategic

limitation in our software system when it comes to running it in privileged mode using sudo. That limitation comes from the usage of the environment symbol LD_LIBRARY_PATH. Among other things, we use it to enable our usage of the correct version of Python in the CMS OS area. Sudo (and other setuid/setgid executables) blocks the propagation of LD_LIBRARY_PATH to the command being run, despite being marked for export, due to the potential for it to be used to sneak malicious code past security protections via an alternate library. Unix-like systems (Linux included) provide that variable's usage for diagnostics, testing, and self-contained relocatable application environments. Use Google to search for LD_LIBRARY_PATH to see fuller information on this subject.

The alternatives that face us are to either:

- run the virtual environment steps directly as root (not needing sudo), or …
- to create the place in the filesystem for the virtual environment and opening up all permissions for it using sudo, then running the virtual environment steps as an ordinary user without using sudo.

We will follow the latter procedure. It is simpler and more convenient because it lets us continue to do everything as an ordinary user. We will not have to interrupt ourselves with going off and being root for a while in another session, and then having to come back to our ordinary user session again.

To anticipate a typical FAQ-like question: the reason we cannot use the Python version that comes with the system (LD_LIBRARY_PATH unnecessary for that) is that the resulting virtual environment would not be useful. We could use sudo to run the system Python (version 2.4.3) to create the virtual environment, but the environment would get version 2.4.3 of Python put into it instead of the version 2.6.4 we need from CMS OS (the environment contains a copy of the Python system that can be run completely independently). When we try to run anything Django with that environment (which is how we must run it for reliability), it will fail.

From anywhere:

```
$ sudo mkdir /opt/plt
$ sudo mkdir /opt/plt/virtualenv
$ sudo chmod o+rwx /opt/plt/virtualenv
```

Change directory back to where the virtualenv tar.gz file was extracted and run the following commands:

```
$ cd virtualenv-1.11.6
$ python virtualenv.py /opt/plt/virtualenv/BASELINE
```

```
$ cd ..
```

Now the virtual environment is created. The next step is to fill up our new virtual environment with a lot of other Python software, including Django itself. This step is called bootstrapping. Change directory back to where the cmsplt environment was checked out and follow through:

```
$ pltenv
$ cd cmsplt/frontend/plt_wbm
$ python bootstrap.py >bootstrap.log 2>&1
```

The first command is a shell function that was defined when we sourced the setenv.sh script several steps ago. It runs a script that lives in the virtual environment that we created, which “activates” the environment. We then run the bootstrap command within the environment, loading it up with the rest of what we need. The output redirection to `bootstrap.log` is optional and for convenience. The bootstrap process downloads various packages from the internet and runs builds upon them, notably the interface module to MySQL itself, and builds it against the environment’s Python and the installed MySQL software.

At this point we have one mission left: to deal with the database that the interface uses. This consists of three tasks: to initialize MySQL itself, create the database used by the PLT Browser Interface, and populate it.

Start the MySQL server for the first time:

```
$ sudo /etc/init.d/mysqld start
```

When the server is started for the first time after being installed, the **mysqld** command above also performs a data area creation and initialization for the server. The creation and initialization prints out a bunch of welcome information, including two **mysqladmin** commands to run to set the password for the mysql **root** user. The second suggested command shows the name of your current host to use with the second of the repeated **mysqladmin** commands below.

If the startup of the MySQL server does not print out the welcome information (and merely outputs `Starting myslqd:  [  OK  ]`), then it has already been started since its installation. In that case, the suggested host name to use with the second command below would be what is obtained from the command:

```
$ uname –n
```

Go ahead and administrate the mysql **root** user:

```
$ mysqladmin –u root password 'new mysql root user password'
```

```
$ mysqladmin -u root -h the.suggested.hostname password 'same new mysql
root user password'
```

If the mysql **root** user already exists, then the first **mysqladmin** command above will deny permission (it has already been run by a previous administrator). If the mysql ***root@the.suggested.hostname*** user also already exists, the second **mysqladmin** command above will also deny permission. One has to find out the mysql **root** user's password to create the **plt_wbm** database and user below (assuming that does not already exist). If one must "break in" to the mysql system because the password cannot be found, the MySQL server must be stopped (`sudo /etc/init.d/mysqld stop`) and the entire `/var/lib/mysql` data area will need to be moved aside to some other name (just rename the `mysql` subdirectory to something else that is descriptive). Starting the MySQL server again will create a new data area from scratch, putting the system into the state anticipated above for starting the MySQL server for the "first time". Re-follow the instructions above from that point.

The next commands specifically target what the PLT Browser Interface needs: its database and the privileges for it, along with the user and password that the interface will present to the MySQL server.

The password for the PLT interface user (plt_wbm) must be obtained from the Python source file `settings.py`. To conveniently reveal it from the command line (instead of using an editor to look at it), perform the following command:

```
$ grep PASSWORD settings.py
```

The password is the text between the second set of single quotes in the revealed lines. Lines that have nothing between the second set of single quotes may be ignored. Please note that the password must be typed in exactly the same case as revealed, or else the password supplied by the interface won't be recognized whenever it tries to talk to the MySQL database server.

Substitute the revealed password for *password from settings* within single quotes among the MySQL commands below:

```
$ mysql –u root –p
(enter password as prompted)
mysql> CREATE DATABASE plt_wbm;
mysql> GRANT ALL ON plt_wbm.* TO 'plt_wbm'@'localhost' IDENTIFIED BY
'password from settings';
mysql> FLUSH PRIVILEGES;
mysql> quit
```

Populating the interface's database is next:

```
$ python manage.py syncdb
(go ahead and create the superuser as prompted)
```

If the **syncdb** command worked, creating a lot of activity, then the database is now populated with all the tables that the interface uses. They will all be empty. The non-empty tables are the ones for Django's built-in admin application.

There's one thing left: starting the PLT Browser Interface's server. This is a test and development server that comes with Django.

```
$ python manage.py runserver
```

The command should do a few things and then end with a line about typing Ctrl-C to stop the server, and just sit there. If one sees that, then everything works, and one is finished.

Gratuitously, start up a browser on the same system and point it to http://localhost:8000. One should shortly see the PLT Browser Interface presenting the home page.

Supplementary: Chronic vs. One-Off Procedures

All the preceding content is a mixture of one-off procedures and procedures that one performs every time that one wishes to use the system after its installation and setup.

Adding oneself to the sudoers file was a one-off procedure. The installation of things with **apt-get**, **yum** and **make** were one-off procedures (such as for CMSOS, XDAQ, ROOT, scons, MySQL, and the MySQL server). The administration of the MySQL database with **mysqladmin** and **mysql** were one-off procedures. The creation of the virtual environment and filling it with bootstrap was also a one-off procedure.

All of the above were one-offs with system-wide availability. Downloading the subversion cmsplt environment and using scons to build the C++ code in it are one-offs with respect to each cmsplt environment on the same system. The scons building procedure is furthermore also a one-off with respect to each set of changes made to the C++ source code within the individual cmsplt environment. All cmsplt environments on the same system will use the same (system-wide) virtual environment. They will all likewise refer to the same database.

The chronic stuff that one must do every time one logs in to the system to use the environment is the following (for review):

```
$ cd cmsplt
$ . setenv.sh
```

…then more commands to go do one of two major classes of activities: run any of many command line utilities, which include programs built from C++, shell scripts, ROOT scripts, and TCL scripts, OR start up the browser interface server (also referred to as the Web Front-End) and browse that from elsewhere.

To do the first class of activities:

```
$ cd interface
```

…then actually run what is required, as directed by more specific documentation elsewhere. An example of each mentioned command line subcategory is given below:

For C++ coded programs:

```
$ bin/someprogram parameters…
```

For shell scripts, some also under the **startup** subdir:

```
$ source scripts/scriptname.sh parameters…
```

For ROOT scripts:

```
$ root –l scripts/scriptname.C
```

For TCL scripts (**basicgui** or **advancedgui**):

```
$ wish scripts/scriptname.tcl &
```

For the second class of activities (start the interface server and browse it), the following should come instead after the setenv.sh command:

```
$ pltenv
$ cd frontend/plt_wbm
$ python manage.py runserver
```

One may then browse at http://localhost:8000. Note that the MySQL server must be running on the system for the interface's `runserver` command above to succeed. If the system has not been modified to start it with each boot-up, and it hasn't been left running since the last usage, then do:

```
$ sudo /etc/init.d/mysqld start
```

…before the `runserver` command. To shut it down after finishing with everything (having exited the browser and typed Ctrl-C to the interface server to stop it), run:

```
$ sudo /etc/init.d/mysqld stop
```

There will be a pause, then a line indicating that Stopping MySQL was [OK].

Two other commands for mysqld exist: **restart** and **status**:

```
$ sudo /etc/init.d/mysqld restart
$ /etc/init.d/mysqld status
```

Note that privilege is not necessary to use the **status** command.

For MySQL to be started whenever the system is booted up, one must add symbolic links to the /etc/init.d/mysqld script into the directories /etc/rc3.d, /etc/rc4.d, and /etc/rc5.d, and the names of the links must be S64mysqld. The locations for these links are specific to Scientific Linux (and the variety of Linux it is based upon). The directories correspond to actions for run levels 3, 4, and 5. Run level 3 corresponds to full multi-user system with networking available, run level 4 is user defined, and run level 5 is full multi-user with network and X window system GUI. These run levels are all the proper context for MySQL server.

The names of the links must be S64mysqld because that fits the naming pattern for startup files: `S`***`nn`***`servicename`, where ***`nn`*** is a number from 00 to 99 for the order in which to start the service relative to starting other services. The number 64 for our MySQL case is obtained from the **chkconfig** comment line in the mysqld script, conveniently done in the following way:

```
$ grep chkconfig /etc/init.d/mysqld
```

The result of doing the above for SL systems seen so far is:

```
# chkconfig: - 64 36
```

…where the '–' tells the installation of MySQL server to not set up starting it in any run level by default (which is why we are doing this explicitly), the `64` is the startup priority order, and the `36` is the shutdown priority order. All the run level directories already have shutdown links for MySQL server that have the `36` in them: all have a K36mysqld link to the /etc/init.d/mysqld script (the naming pattern for shutdown files being `K`***`nn`***`servicename`). These were created by installation of MySQL server.

chkconfig is a utility for managing startup and shutdown of services on Berkeley-style Unix systems (BSD). SL is System V style instead of Berkeley, but

the man page for **chkconfig** is always available, and the service scripts have the chkconfig information lines in them for usage on BSD systems.

To go ahead and create the symbolic links in each of the listed runlevel directories, do:

```
$ sudo ln –s ../init.d/mysqld S64mysqld
```

…where we also may note that the link is to ../init.d/mysqld instead of /etc/init.d/mysqld in order to match what installation created for the K36mysqld links.

Supplement II: PLT Strip Chart Monitoring Software

During physics data production, the PLT subsystem will be publishing its live luminosity data to a CMS-wide and LHC-wide data service that any other software can subscribe to. A data subscriber for monitoring PLT performance exists in the cmsplt software, borrowing technology from similar monitors for beam radiation for CMS (cmsbrm). It is also accompanied by a simulated publisher of PLT data for testing and development purposes.

For the purpose of testing and development of the above publisher and subscriber on University of Tennessee systems, two other sets of scripts and software are present. One is for running a domain name server to allow publishers and subscribers to find the data caching service (DIP) to communicate with it, and the second is for the building and operation of the DIP service itself. When operating on CMS systems at CERN, in contrast, these two services are already running and available system-wide.

To summarize, there are four software entities to consider:

1. A DNS server for DIP communication.
2. A DIP cache server to store data from publishers and distribute it to subscribers.
3. A simulated PLT data publisher.
4. A PLT data subscriber that displays the data in a strip chart graphical form.

This section documents how to build, install, and run these software entities.

For usage on the **cmsvol1** system (a specific individual system at the University of Tennessee), all the necessary supporting system resources are already in place. For setup and usage on other systems, at least the XDAQ and CMSOS software must be installed as mentioned at the beginning of this chapter,

although this is purely to satisfy the `setenv.sh` script rather than used to run the four software entities of PLT strip chart monitoring. SVN will have to be available as well in order to have obtained the cmsplt software in the first place, also as mentioned at the beginning.

Additional resources to install on other systems that are needed to support PLT strip chart monitoring are the following:

- DIM DNS software from CERN, installed in /opt/dim.
- A DIP development system from CERN, installed in /opt/dip.
- Maven software installed in /opt/plt/maven.

The DIM software (Distributed Information Management) is obtained by visiting the CERN web page for DIM (C. Gaspar, EP Division (CERN), 2009) and following its instructions. Unzipping the downloaded .zip archive file under /opt, renaming the resulting directory to “dim”, and following the build instructions will accomplish the installation. If one is on a system that requires using **gmake** to build the software (also include the option to build JDIM), and the build stops when trying to build “did”, the dns server and its library will still be built and usable.

The DIP development system is obtained by visiting the CERN web page for it (Industrial Controls Engineering Department (CERN), 2014), and downloading the appropriate package for one’s system. Unzipping the downloaded .zip archive under /opt and renaming the resulting directory to “dip” will accomplish the installation. No building will be required. In the case of using a tarball provided for older systems and deprecated versions, the tar extraction will put everything into a subdirectory called **linux** instead of **dip-*versionnumber***. It is the **linux** directory that shall have to be renamed to “dip”.

Maven can be obtained from the Apache Maven web page (The Apache Software Foundation, 2014). Installation is similarly straightforward (no build actions): extract the tar file under /opt/plt and rename the resulting directory to “maven”. The README.txt file under the maven directory provides additional information to verify correct installation.

CMSVol Repository

There is a Java software repository on the University of Tennessee’s CMS Vol cluster that contains a snapshot version of CERN’s BRM (Beam Radiation Monitoring) software and all other CERN and third party software of the correct versions required by it. This repository is referenced by maven builds of the DIP

cache and the strip chart software in order to satisfy their dependencies. The repository server must be running on **cmsvol1** to fulfill the requests made by maven (otherwise, the builds fail).

The presence of the server on the **cmsvol1** system can be revealed by:

```
$ /local/nexus/bin/jsw/linux-x86-64/nexus status
```

If the message says that the server is not running, it may be started by:

```
$ sudo –u daemon /local/nexus/bin/jsw/linux-x86-64/nexus start
```

After it's started, the repository will be available to answer any requests from maven. It may also be browsed through the URL:

http://cmsvol1.phys.utk.edu:8060/nexus/index.html

The server may also be commanded to 'stop' and 'restart' like all other typical server processes:

```
$ sudo –u daemon /local/nexus/bin/jsw/linux-x86-64/nexus stop
```

The repository server is only absolutely necessary for the first build (and run) of the DIP cache software and the first build of the strip chart software performed by a user on the system. Maven stores downloaded objects into a repository in the user's login directory, under `.m2/repository`. Subsequent builds using maven will get objects from there instead of referencing the server, unless the user's repository is removed or otherwise cleared.

Building

With the software present on the system under the directory **cmsplt** as directed previously:

```
$ cd cmsplt
$ . setenv.sh
```

Jump into a deeper directory and build the first piece of the software:

```
$ cd frontend/publishers
```

The next command builds the test PLT data publisher, a C++ coded program called `plt-fake-realistic-publisher`. The source file it is built from is in the directory just entered, and its name is `plt-emulator-realistic.cpp`.

```
$ ./buildrealistic.sh
```

There should be no messages printed by building the publisher program. To build the next item on the itinerary, one must first move to a different directory:

```
$ cd ../brm-wbm
```

The following is a maven command. It performs a Java software build. The `pom.xml` file in this directory defines what the **mvn** command will do. The case variation in the spelling of the `-D` option to the **mvn** command is important. This command also occurs in the third line of the `README` file in this directory.

```
$ mvn -DskipTests=true clean assembly:assembly
```

This will output a lot of messages about all the actions being performed (warnings about various things, copying of resources, compiling source files (Java), building and verifying .war files and .jar files (web archive files and java archive files)). One should end up with INFO messages about build success at the end of it all:

```
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------
[INFO] Total time: 16.650s
[INFO] Finished at: Mon Sep 08 19:05:47 EDT 2014
[INFO] Final Memory: 14M/160M
[INFO] ------------------------------------------------------------------
```

The big result obtained from all of this is a tape archive file (.tar) that has a bunch of .jar files in it that the strip chart monitor program will load in when it runs. Change to another directory for the next command:

```
$ cd ../plt_wbm/core/static/displays
```

The following command will extract the needed files from the (**gzip** compressed) tape archive file that the '**mvn**' command constructed. Most of the important stuff will be put into a subdirectory called 'lib'. Another jar file will be put into the current directory along with four html files that won't actually be used for the strip chart application. The **tar** command below is one long command line (ignore the line ending caused by margin formatting in this document).

```
$ tar -xzvf $PLT_BASE/frontend/brm-wbm/target/brm-wbm-
1.0.3-SNAPSHOT-applet-package.tar.gz
```

There is one more thing to do before the unloaded jar files can be used. They must be signed. A Python program will be used that refers to a keystore file that came down from the repository. Using it will require a password. The password may be revealed by doing the following:

```
$ grep storepass $PLT_BASE/frontend/brm-wbm/pom.xml
```

The word between `<storepass>` and `</storepass>` is the password to use when prompted.

The following must be all one command line (ignore the line endings caused by margin formatting in this document).

```
$ python $PLT_BASE/frontend/brm-wbm/scripts/sign-jar.py
brm-wbm-1.0.3-SNAPSHOT.jar lib/jdataviewer-
compat.jar lib/dip.jar lib/spring.jar lib/xercesImpl.jar
lib/httpclient.jar lib/httpcore.jar lib/commons-logging.jar
lib/log4j.jar lib/flute.jar lib/sac.jar
```

Now all the preparatory steps for enabling stand-alone running of PLT data publishing and strip chart monitoring are finished.

An alternative: If one wishes to create a keystore from scratch to sign the jar files with (and use a truly private password), one must either remove the keystore file that came from the repository or move it to a different name, and then use a program called **keytool** to create a new one in its place with the same name. The same name must be used in order for the Python signing program to find it.

To remove:

```
$ rm -i $PLT_BASE/keystore/brm
```
(answer y to confirm removal)

To move out of the way instead:

```
$ mv $PLT_BASE/keystore/brm $PLT_BASE/keystore/brm_original
```

Create the new keystore:

```
$ keytool -genkeypair -keystore $PLT_BASE/keystore/brm -alias
brmkey –validity 7305
```

Answer all prompts for password, password confirmation, first and last name, organizational unit, organization, City or Locality, State or Province, and two letter country code.

Answer yes for confirmation of all of the information.

Just press RETURN for the key password for <brmkey>.

The jar files may then be signed using the new keystore file by running the Python signing program presented above. One may note that the key has been defined to be valid for 7305 days, which is 20 years' worth of days, including leap years.

Starting Up

Here's how to start up PLT strip chart monitoring on every log-in session.

```
$ cd cmsplt
$ . setenv.sh
$ cd frontend/brm-wbm
```

This starts up the local DNS service to allow publishers and subscribers to talk to the DIP cache service,

```
$ nohup scripts/launch-dip-dns.sh &>dip-dns.log &
```

This starts the DIP cache service itself.

```
$ nohup scripts/launch-dipcache.sh &>dipcache.log &
$ cd ../publishers
```

This is a publisher of "fake" PLT luminosity data. It sends its made-up data to the DIP cache for subscribers to read.

```
$ nohup ./plt-fake-realistic-publisher &>publisher.log &
$ cd ../plt_wbm/core/static/displays
```

Finally, there is a subscriber to the PLT data. It reads back the stuff made by `plt-fake-realistic-publisher` and displays it in a strip chart. The subscriber is coded in Java, configured using XML, and uses **javaws** (JAVA Web Start) to run.

There are two ways for it to run. One is server based, where the subscriber asks the PLT browser interface server for the jar files and the XML file that it needs, which then transmits them to the subscriber over an http connection. The other way to run is “stand-alone”, where the subscriber opens the jar files and the XML file directly from the file system itself. The difference between doing it one way versus the other is in the file that **javaws** uses to tell it what to do, a .jnlp file (Java Network Launch Protocol), which is also XML format data.

The server based way requires one to go through all of the system setup and building of the PLT browser interface software documented above, and then start the interface server to have it running like the DNS server, DIP cache, and publisher documented in this section. To start the server based strip chart subscriber, issue the command:

```
$ javaws launchPLTMonitor.jnlp
```

To start the stand-alone strip chart subscriber, issue the command:

```
$ ./launchPLTMonitor_standalone.sh
```

Both commands start a Java console process in the background and then finish quickly. After a short delay, the Java console will present a splash screen for the Java system version, and then another window stating that it is downloading the application. A further window will appear claiming that, "The application's digital signature could not be verified. Do you still want to run the application?" Go ahead and press the Run button. The strip chart window with two displays shall appear shortly. The strip chart may be dismissed by pressing the X button at the top right corner of the window.

When the entire complex of services is running, it is a pretense that the LHC is running, and that there is a running PLT inside the CMS experiment sending data to the DIP cache, and that one is in the CMS control room receiving the data on an operator's console.

Development and Testing Issues

The definition for the windows and displays of the strip chart is in the file `PLTSummary.xml`. This is the source file to modify in order to change the display, and both versions of the strip chart subscriber use it. It is referred to in the .jnlp file, as are the necessary jar files.

For modifying the .jnlp file, the situation is different for the server based versus the stand-alone method of running the strip chart subscriber. The server based

.jnlp file is straightforward, it is the parameter to the **javaws** command quoted above, `launchPLTMonitor.jnlp`. For the stand-alone strip chart subscriber, the file to modify is `launchPLTMonitor_standalone_jnlp_template.xml`. The `launchPLTMonitor_standalone.sh` script generates a .jnlp file from this template XML file every time it is executed. This is to make the stand-alone method portable in terms of where in the file system the jar files and the `PLTSummary.xml` is read from. The script substitutes the current definition of the PLT_BASE environment symbol into the template file to make the .jnlp file, and then runs a **javaws** command on the .jnlp file.

The **javaws** program speeds up the loading and execution of the Java software by caching pre-loaded copies of everything. This can get in the way of trying changes to the files, as the old cached version will be executed instead of one's new changes. The solution is to clear the cache by doing the following:

```
$ javaws -viewer
```

This invokes the Java Cache Viewer, part of the Java Control Panel. From there, one can interactively select the cached copy of the "**PLT Display**" application and use the "**Remove selected items**" button (with a big red X) to delete it. The next time the PLT Monitor is executed, it will be loaded from the files, and any source changes should then take effect.

Shutting Down

When finished with exercising the strip chart, there will still be three processes running in the background: the publisher, the DIP cache, and the DNS service. Starting them with '**nohup**' allows one to log off the system and leave them still running. Cleaning them up before logging off the system is a courtesy if one is not going to use them for an extended period of time, especially where the publisher process is concerned. The publisher process will be the most noticeable consumer of processor time.

Shutting the services down in the reverse order that they were started is also maximally graceful.

The process ids that were reported each time the '**nohup**' command was used are not the process ids to use to stop them (except for the publisher process). To get the correct process ids to stop, one shall have to do a little search for them in each case. This is done by using the '**ps**' command to list all of the processes associated with one's user id, combined with the '**grep**' command to filter out just the one of interest for each case. The first number in what is listed is the process id to use the '**kill**' command with to stop it. The '**grep**' process that also shows up

in the listing is simply the transient **grep** command that is filtering the list of processes, and will no longer exist after the listing is complete. The '**kill -2**' command is equivalent to typing Ctrl-C in the shell when running each respective program in the foreground (not having started it with '**nohup**' and '**&**' at the end of the command line).

```
$ ps -f -u yourusername | grep publisher
yourusername 30784 27646  0 21:56 pts/6    00:00:00 ./plt-fake-
realistic-publisher
yourusername 31114 27646  0 22:27 pts/6    00:00:00 grep publisher

$ kill -2 30784

$ ps -f -u yourusername | grep jetty:run
yourusername 31633 31632  0 23:05 pts/6    00:00:06
/opt/plt/jdk/bin/java -classpath /opt/plt/maven/boot/plexus-
classworlds-2.4.jar -Dclassworlds.conf=/opt/plt/maven/bin/m2.conf -
Dmaven.home=/opt/plt/maven
org.codehaus.plexus.classworlds.launcher.Launcher jetty:run
yourusername 31786 31550  0 23:20 pts/6    00:00:00 grep jetty:run

$ kill -2 31633

$ ps -f -u yourusername | grep dim
yourusername 31617 31616  0 23:03 pts/6    00:00:00 /opt/dim/linux/dns
yourusername 31808 31550  0 23:22 pts/6    00:00:00 grep dim

$ kill -2 31617
```

To check if the processes are running under some other unknown user, '`ps -ef`' may be used instead with each **grep** command.

Further Information

There are two CERN TWiki pages that provide additional information. (Hollingsworth, 2011) covers some of the above material in an earlier context of CERN systems. It also mentions web browser based diagnostics for `plt-fake-publisher`, and a Java coded DIP tester, both of which are also included and usable in the cmsplt software. (Hollingsworth, DipCache, 2010) provides additional background information on the DIP cache service.

References

Bicking, Ian; The Open Planning Project; PyPA. (2014, November 10). *virtualenv -- virtualenv 1.11.6 documentation*. Retrieved from virtualenv: http://www.virtualenv.org

C. Gaspar, EP Division (CERN). (2009, April). *Instructions for downloading, installing and running DIM on UNIX*. Retrieved from DIM: http://dim.web.cern.ch/dim/dim_unix.html

Django Software Foundation. (2014, March 2). *The Web framework for perfectionists with deadlines | Django*. Retrieved from django: https://www.djangoproject.com

Hollingsworth, M. (2010, May 6). *DipCache*. Retrieved from TWiki: https://twiki.cern.ch/twiki/bin/viewauth/CMS/DipCache

Hollingsworth, M. (2011, August 2). *PLTRealtimeDisplays*. Retrieved from TWiki: https://twiki.cern.ch/twiki/bin/viewauth/CMS/PLTRealtimeDisplays

Industrial Controls Engineering Department (CERN). (2014, May 5). *DIP API - EN-ICE - Confluence*. Retrieved from EN-ICE - Confluence: https://j2eeps.cern.ch/wikis/display/EN/DIP+API

Orsini, L. et al (The CMS Experiment). (2014, July 10). *The Official XDAQ CMS Online Software project page*. Retrieved from cmsos: https://svnweb.cern.ch/trac/cmsos (Redirected from http://xdaq.web.cern.ch)

Orsini, L. et al (User Man). (2014, March 3). *EDMS: CMS Online Software User's Manual.* Retrieved from EDMS: https://edms.cern.ch/document/1001791

The Apache Software Foundation. (2014, September 17). *Apache Maven Project*. Retrieved from Apache Maven Project: http://maven.apache.org

The ROOT Team. (2014, February 24). *ROOT | A Data Analysis Framework*. Retrieved from ROOT | A Data Analysis Framework: http://root.cern.ch

The SCons Foundation. (2014, March 2). *SCons: A software construction tool*. Retrieved from SCons: A software construction tool: http://www.scons.org

Wissing, C. (2013, April 26). *Installing CMSSW Using apt.* Retrieved from CMSPublic Web: https://twiki.cern.ch/twiki/bin/view/CMSPublic/SDTCMSSW_aptinstaller

CHAPTER V
NAVIGATING THE SOFTWARE FOR THE PLT BROWSER INTERFACE

Abstract

A guided tour of the source code is presented for software maintainers.

Introduction

The browser interface for the up and coming Pixel Luminosity Telescope (PLT) portion of the CMS (Compact Muon Solenoid) detector at the CERN's Large Hadron Collider is powered by a web development framework called Django (Django Software Foundation, 2014). Django in turn is coded in the interpreted, high-level, and object-oriented language called Python (Python Software Foundation, 2014). One's application coding using Django is in Python. Django organizes and deploys the diverse technologies of browser-server communication, HTML information markup, and database operations into a framework that one can just plug into with one's own HTML template files and Python code to achieve a web site of any size and complexity.

This document is intended to introduce new developers/maintainers to the ins and outs of the implementation of the PLT Browser Interface deployed using Django.

A few general items to start off with mentioning are the version of Django being used, and the other technologies used by this deployment. Django version 1.3 is what's currently in use. This is important to note for the sake of consulting the correct documentation and tutorials to read on the Django project web site.

One of the things one must decide when using Django is which database engine to use for the backend. In this case, MySQL is what is being used. That runs as a server process on the system, and client processes, such as the Django web server, talk to it via a socket file. The server works on the database files directly and prevents corruption problems that can be caused by multiple processes altering the files on their own. Multi-process safety is necessary for this deployment, as there are times when multiple threads are running that must talk to the database engine. Those times happen to be when one is running calibration jobs and when one is running data taking jobs.

Another technology, one that is not necessitated in any way by Django usage, is the ability to use C++ code from Python code. Boost (Dawes, Abrahams, & Rivera, 2013) is being used to connect Python and C++ code together, via its boost_python technology. A version of boost is included in the CMS software installed on the computer system. The aforementioned calibration and data taking software (and some other helpful bits of code) is coded in C++ and called

from the Python code of our application. The same C++ code is also used from command line programs also built from C++.

The final major web technology to mention, used by The PLT Browser Interface (and also not necessitated by Django), is Javascript. Javascript code, unlike the Python code, runs in the browser program itself (the Python code runs in the server on the remote system, by contrast). Our use of Javascript is enhanced and supported by including major third party Javascript packages of JQuery and ProtoVis. Altogether, Javascript is being used to provide the following things:

- Various fine formatting and rendition effects on all pages.
- An interactive graphical display of complex data on the startup page (luminosity data).
- List display widgets on subordinate pages to provide search and control over how many list items are shown.
- Support timed interval requests to the Django web server from the browser with insertion of received-back information into appropriate places on the visible web page.

For the last item mentioned above, there are two timed requests that run. One is for the displayed uptime of the server. This runs every two seconds. The other one is for the status of data taking on the system. This runs every five seconds.

The Files: What's where and how things are organized

To start off with introducing this thread of information, we must consider what is created when one downloads ("checks out") a copy of the development environment for **cmsplt** from the repository (via svn). From presuming that we have run the following command:

```
# svn checkout svn+ssh://some_user_name@svn.cern.ch/reps/cmsplt/trunk
cmsplt
```

…we can start with the subdirectory called **`cmsplt`** that is in the directory where we ran the 'svn' command example above.

```
drwxrwxr-x  6 mfoerste mfoerste 4096 Jul 25 23:25 analysis
drwxrwxr-x  3 mfoerste mfoerste 4096 Jul 25 23:25 cosmo4
drwxrwxr-x  5 mfoerste mfoerste 4096 Jul 25 23:25 database
drwxrwxr-x  3 mfoerste mfoerste 4096 Jul 25 23:25 EventDisplay
drwxrwxr-x  6 mfoerste mfoerste 4096 Jul 25 23:25 FEDStreamReader
drwxrwxr-x  7 mfoerste mfoerste 4096 Jul 25 23:25 frontend
drwxrwxr-x 16 mfoerste mfoerste 4096 Aug  6 02:27 interface
drwxrwxr-x  3 mfoerste mfoerste 4096 Jul 25 23:25 keystore
```

```
-rw-rw-r--  1 mfoerste mfoerste  795 Jul 25 23:25 README_svn
-rw-rw-r--  1 mfoerste mfoerste 1067 Jul 25 23:25 setenv-custom.sh
-rw-rw-r--  1 mfoerste mfoerste 4544 Aug  6 23:17 setenv.sh
-rw-rw-r--  1 mfoerste mfoerste 1174 Jul 25 23:25 setenv-slc5.sh
drwxrwxr-x  3 mfoerste mfoerste 4096 Jul 25 23:25 slink
drwxrwxr-x  7 mfoerste mfoerste 4096 Jul 25 23:25 testbeams
```

Under **cmsplt**, there are two subdirectories of immediate importance (highlighted in red): **interface** and **frontend**. All of the C++ code mentioned earlier lives in subdirectories under **interface**. **interface** is also the directory in which one runs commands to build some or all of that C++ code. It also contains other subdirectories that masses of data are written to (both calibration and data taking), where various text format configuration files are located, an older GUI for the PLT, trigger control, and historical testing scripts and data.

The other of the two mentioned subdirectories, **frontend**, is where all the Django-powered action resides.

```
drwxrwxr-x 6 mfoerste mfoerste 4096 Jul 25 23:25 brm-wbm
drwxrwxr-x 4 mfoerste mfoerste 4096 Jul 25 23:25 displays
drwxrwxr-x 6 mfoerste mfoerste 4096 Aug  6 02:38 plt_wbm
drwxrwxr-x 4 mfoerste mfoerste 4096 Jul 25 23:24 publishers
```

Actually, that's a slight oversimplification. It is the **plt_wbm** subdirectory under **frontend** that has the entire Django-powered system. The sibling directories of **plt_wbm** hold things extracted and further derived from another svn repository called **cmsbrm**, which is for CMS Beam Radiation Monitoring. Those are safety systems for the CMS detector, something that the originator of the browser interface also partly created and worked upon. From here on, we shall ignore those siblings and concentrate our attention solely upon the contents under `cmsplt/frontend/plt_wbm`.

The name of our system root directory, **plt_wbm**, stands for PLT Web Based Management. The Django documentation describes this top level point as the Django *project*. Django uses this project name as the name for the database, and also as the user name for accessing the database. Underneath any Django project is one or more *applications*. Django comes with a handful of its own applications "out-of-the-box" to do with managing user ids and permissions and other administrative stuff. These built-in applications may be voluntarily used or dis-used according to the developer's and administrator's whims. Aside from making those decisions, one then proceeds to define one's own application or applications. In our case here, our PLT Browser Interface application has been named 'core'. Admittedly, this is very generic and non-specific, but it works just fine.

The larger purpose for laboriously introducing all this information about *project* and *application* is to show where all the naming of the data base and its tables come from. Each table in the project's database (plt_wbm) has its name constructed from the application it belongs to followed by an underscore and then the name of the table as defined in the application's **models** module (more about that soon). An illustrative example is the DataRun table in our PLT Browser Interface application (called "core"). The name of the table for it in our MySQL database ends up as "core_datarun" (any camel-case in the name in the application's models is also reduced to being uniformly lower-cased). Specifying the full name in **db.table** notation results in "plt_wbm.core_datarun" in contexts where the database name is not implicit. Knowing this is especially useful when administrating the tables manually with SQL, or reading a text dump of the database.

What's under plt_wbm:

```
-rw-rw-r-- 1 mfoerste mfoerste  1674 Jul 25 23:25 bootstrap.py
-rw-rw-r-- 1 mfoerste mfoerste   196 Jul 25 23:25 context_processors.py
drwxrwxr-x 4 mfoerste mfoerste  4096 Aug  8 21:32 core
-rw-rw-r-- 1 mfoerste mfoerste   575 Jul 25 23:25 __init__.py
-rw-rw-r-- 1 mfoerste mfoerste   503 Jul 25 23:25 manage.py
-rw-rw-r-- 1 mfoerste mfoerste 22595 Jul 25 23:25 pltsystem.py
-rw-rw-r-- 1 mfoerste mfoerste    73 Jul 25 23:25 requirements.txt
drwxrwxr-x 3 mfoerste mfoerste  4096 Jul 25 23:25 scripts
-rw-rw-r-- 1 mfoerste mfoerste   647 Jul 25 23:25 server.py
-rw-rw-r-- 1 mfoerste mfoerste 10606 Jul 25 23:25 settings.py
drwxrwxr-x 4 mfoerste mfoerste  4096 Aug  9 21:40 templates
-rw-rw-r-- 1 mfoerste mfoerste  1012 Jul 25 23:25 urls.py
```

Enumerating the collection of files seen here:

`manage.py` – This comes from Django and supplies many subcommands documented on the Django web site. Once operational, the two most often used subcommands will be **runserver** and **dbshell**:

```
# python manage.py runserver
```

…and…

```
# python manage.py dbshell
```

The **runserver** subcommand starts up a test and development web server that one may connect to with a web browser to test and further develop any of the applications that the Django project knows about. That includes our "core" application for the PLT Browser Interface.

The **dbshell** subcommand starts up the interactive shell for the database engine being used, getting one into the system as the project user with the password already supplied. One is also put in the context of doing

operations upon the project database, so that SQL commands issued need only specify the table names without needing to include which database they belong to. This is a great convenience for doing direct maintenance, setups for testing, cleanups after testing, and so forth.

When putting the PLT Browser Interface onto a system for the first time, when the project database and none of the application tables exist, one will also make use of the **syncdb** subcommand. It narrates off of the models for each of the project's applications and creates the tables in the database that correspond to each model defined in each application's models file. This is used for both comprehensive creation of everything the first time and for updates to add new tables. This and all of the other useful `manage.py` subcommands are thoroughly documented on the Django web site.

`bootstrap.py` and **`requirements.txt`** – These are, like the **syncdb** subcommand, a one-off first time installation command. The bootstrap.py program ***does*** have to be run before Django can be used because it is used to install Django and a few other Python tools directly from the web. The requirements.txt file contains data that the bootstrap program reads, telling it what to download and install, and what version(s) to get. The bootstrap program is also responsible for building the code that allows Django to talk to the database engine that is set to be used. The place that bootstrap installs stuff for the working PLT Browser Interface to use is a place defined by another tool, not seen here, called **virtualenv**. Virtualenv is used to manage the Python version environment for Python applications that are version-sensitive.

`settings.py` – This plays a very central and important role in defining everything about the Django project and its applications. Anything having to do with settings for the PLT Browser Interface application, or for any of the other applications in the project, is located here. Among the project-wide highlights are which apps are active or disabled in the project, the name of the project and its database and the Django user for the database, and which database engine is being used. One must consult the Django documentation and read through the actual settings file to get a full appreciation of its power.

`urls.py` – This contains regular expressions (pattern matching) to define where URLs in the browser's address field get directed to in the rest of the Django system (what Python code gets executed to respond to the URL request). The urls.py file at this level defines the URL patterns for the entire **plt_wbm** *project*, encompassing all of its applications. Currently this

contains a reference to Django's admin application, and a reference to the urls.py file for our **core** application.

`pltsystem.py` – This is where we encounter some system management code for the **core** application of the PLT Browser Interface. The Python version of code for powering up the PLT and establishing trigger settings lives here.

`__init__.py` – Like any good Python-powered system, we also have one of these. Python treats directories that have an __init__.py in them as packages of Python code. This __init__.py has code to support the server startup.

`context_processors.py` – This makes knowledge of the PLT power state and the active global run available to HTML template substitution (a primary Django feature) so that it shows up on every page of the PLT Browser Interface.

`server.py` – This is currently unused litter from earlier development.

The files under the **`scripts`** subdirectory are Python scripts for testing things during development. The **`core`** and **`templates`** subdirectories are major topics explained further below.

That finishes the introduction to things at the Django project level. Now we must investigate the files of the **core** application itself. Under **`core`**:

```
-rw-rw-r-- 1 mfoerste mfoerste  1377 Jul 25 23:25 fakecal.py
-rw-rw-r-- 1 mfoerste mfoerste   231 Jul 25 23:25 forms.py
-rw-rw-r-- 1 mfoerste mfoerste     0 Jul 25 23:25 __init__.py
-rw-rw-r-- 1 mfoerste mfoerste  3081 Jul 25 23:25 managers.py
-rw-rw-r-- 1 mfoerste mfoerste 37475 Aug  8 20:39 models.py
drwxrwxr-x 8 mfoerste mfoerste  4096 Jul 25 23:25 static
-rw-rw-r-- 1 mfoerste mfoerste   991 Jul 25 23:25 tests.py
-rw-rw-r-- 1 mfoerste mfoerste  2188 Jul 25 23:25 urls.py
-rw-rw-r-- 1 mfoerste mfoerste 29273 Jul 25 23:25 util.py
-rw-rw-r-- 1 mfoerste mfoerste 43495 Aug  7 23:51 views.py
```

The files critical to the Django operation of the **core** app are __init__.py, models.py, and views.py. This simply means that all Django applications have them. The urls.py file here is an optional exercise of Django where instead of having all URLs for all the project's applications defined in one urls.py at the project level, the URLs for this application have been segregated out into its own urls.py. All the other modules are the rest of the implementation of our **core** app that the developer is free to create, name, and organize.

`__init__.py` – This at the **core** app level is present for the Python package purposes. It has no additional code in it.

`models.py` – This finally gives us a look at an app's **models** module, mentioned earlier. It defines the database usage of the application by defining the tables that the application uses. Each table's record structure comes from extending the Django class for table modeling (called "Model"), and using the Django types for each field in the table. Each of any record of the table read in will correspond to an instantiation of an object of that class. Instantiating a new object from scratch will correspond to a newly inserted record in the table when it is saved. As is normally the case with Python, one may use more than just the types from Django for definitional purposes. One may create additional non-table data members and define methods for the derived class for additional features and functionality callable from other code. A notable example of this is for stateful control over which procedures may be executed in what order for calibration model objects and data run model objects.

`views.py` – The Django term for classes and procedures that deal with requests from the browser and construct results to send back is called a *View*. The collection of Views in views.py is where all the Python procedural code for dealing with the interface is located. Requests are directed in to procedures by the URL definitions, the appropriate processing is done, and results are returned. This holds for both user interaction in the browser and the automated Javascript timed interval requests from the browser. The processing for the server uptime information is in here, along with the processing for the current state of data-taking, and the processing of all the forms and gadgets in all the HTML pages in our PLT Browser Interface. After the processing for a user response has finished, execution cycles around eventually to Django's template processing, which substitutes data into the appropriate HTML template page for transmission back to the browser. Other procedures in the view code may be called during this final stage of processing to supply the information that gets substituted.

`urls.py` – The version sitting here in **`core`** is the one referred to by the urls.py at the project level. This provides the connections between URLs in the browser and the code in views.py to deal with them.

`util.py` – This contains utility code used by models procedures and views procedures. It contains "workloop" code that performs calibration jobs and data taking jobs in a separate thread, it contains queue code for feeding jobs to the workloop, another queue for control messages for jobs, access control classes for data used by data run monitoring, and more.

`forms.py` – This has code that compactly specifies the fields for Django to use to create a form for the creation of a new DataRun record. DataRun is a model in models.py. The new data run record corresponds to (what will be) a new data taking job. For full and general information, there is more about Django's forms facility in the Django documentation.

`fakecal.py` – An unused module that was intended to contain code for pretending that there is PLT hardware on a system that has none, for testing interface changes.

`managers.py` – Unused litter from earlier development of calibration management code.

`tests.py` – A self-descriptive module containing test code that is run by the **test** subcommand of the **`manage.py`** script, ie: `python manage.py test`

That's all for the Python code. There's one more lingering issue for the **`core`** subdirectory. That is the **`static`** subdirectory under it. The stuff stored under there is all kinds of other data referenced by HTML coding. That consists of image files, third-party Javascript code, and CSS (Cascading Style Sheet) files that set stylistic properties.

```
drwxrwxr-x 3 mfoerste mfoerste  4096 Aug  8 23:46 core
drwxrwxr-x 4 mfoerste mfoerste  4096 Jul 25 23:25 css
drwxrwxr-x 3 mfoerste mfoerste  4096 Jul 25 23:25 displays
drwxrwxr-x 3 mfoerste mfoerste  4096 Jul 25 23:25 images
-rw-rw-r-- 1 mfoerste mfoerste 85925 Jul 25 23:25 jquery-1.5.2.min.js
drwxrwxr-x 4 mfoerste mfoerste  4096 Jul 25 23:25 js
```

`displays` – The content of this directory is like the siblings of the **`plt_wbm`** directory mentioned earlier. It is **cmsbrm** related material, not used by our PLT Browser Interface.

`images` – This has the mentioned image files. That is comprised of the data run monitoring indicators, the data run control button images, and some images for the navigation bar.

`js` – This is where the Javascript sources for JQuery and ProtoVis are stored. Our HTML files include them from here.

`css` – Contains JQuery's own CSS files and images that it refers to in its own code.

`core` – Contains the one CSS file of our own that we use (base2.css), plus an earlier one and an image that it references.

Now we can move on to the final category and the end of our tour: the HTML template files.

Under templates:

```
-rw-rw-r-- 1 mfoerste mfoerste 3834 Jul 25 23:25 base.html
-rw-rw-r-- 1 mfoerste mfoerste 1195 Aug  9 21:40 base_with_nav.html
drwxrwxr-x 3 mfoerste mfoerste 4096 Jul 25 23:25 calibration-detail
-rw-rw-r-- 1 mfoerste mfoerste  226 Jul 25 23:25 calibration.html
-rw-rw-r-- 1 mfoerste mfoerste 2562 Jul 25 23:25 calibration_set.html
-rw-rw-r-- 1 mfoerste mfoerste 1375 Jul 25 23:25 calibration_sets.html
-rw-rw-r-- 1 mfoerste mfoerste  712 Jul 25 23:25 calibrations.html
-rw-rw-r-- 1 mfoerste mfoerste 2003 Jul 25 23:25 calibrations_new.html
-rw-rw-r-- 1 mfoerste mfoerste    0 Jul 25 23:25 config_explorer.html
-rw-rw-r-- 1 mfoerste mfoerste  646 Jul 25 23:25 data.html
-rw-rw-r-- 1 mfoerste mfoerste  246 Jul 25 23:25 data_new_run.html
-rw-rw-r-- 1 mfoerste mfoerste 2000 Aug  9 20:52 data_run.html
-rw-rw-r-- 1 mfoerste mfoerste  722 Aug  8 02:16 data_runs.html
-rw-rw-r-- 1 mfoerste mfoerste  687 Jul 25 23:25 global_run.html
-rw-rw-r-- 1 mfoerste mfoerste  921 Jul 25 23:25 global_runs.html
-rw-rw-r-- 1 mfoerste mfoerste 4142 Jul 25 23:25 index.html
-rw-rw-r-- 1 mfoerste mfoerste 2490 Jul 25 23:25 power.html
-rw-rw-r-- 1 mfoerste mfoerste  201 Jul 25 23:25 queue.html
-rw-rw-r-- 1 mfoerste mfoerste  134 Jul 25 23:25 settings.html
```

…and what's under calibration-detail, too:

```
-rw-rw-r-- 1 mfoerste mfoerste 797 Jul 25 23:25 GainCalFast.html
-rw-rw-r-- 1 mfoerste mfoerste 797 Jul 25 23:25 LevelsCalibration.html
-rw-rw-r-- 1 mfoerste mfoerste 797 Jul 25 23:25 SampleCalibration.html
-rw-rw-r-- 1 mfoerste mfoerste 797 Jul 25 23:25 TrimCalFast.html
-rw-rw-r-- 1 mfoerste mfoerste 797 Jul 25 23:25 UltrablackCalibration.html
```

This is where we see what shows up in the browser. These HTML files contain Django template coding for substituting data from the Python code. Anything between and including a pair of "`{%`" and "`%}`" is the Django template markup. The template processor markup offers conditional logic, iterative substitution from a list, inclusion of other template files, and much more. All the files seen in this directory and the **`calibration-detail`** subdirectory appear by themselves in the browser after processing, except for two of them. The template HTML files `base.html` and `base_with_nav.html` are themselves included (by the template processor) into the beginning of all of the other HTML files. They supply all of the surroundings that are always visible on each page: the top bar of information, the links to PLT power, the "Console" region on the right side, and the navigation bar in the upper center. The `base.html` included file also contains the Javascript code for periodically getting server uptime and data run

monitoring. The earlier mentioned `context_processors.py` at the project level supplies the substitution data called for by template coding in `base.html` itself (power state and global run number). Contrast that with the source of substitution data in all the other individual HTML files, which is `views.py` code.

The Connections: Chasing how things work from beginning to end

A vital part of being able to develop this system is simply being able to find the code that responds to a particular feature in the browser interface (links, buttons, et cetera). The start of this process is to look at the URL in the browser bar when one is on the page and feature of interest, and to follow a chain of connections to find both the Python code and the HTML template file that is involved. The step-by-step recipe for following the chain of connections is below:

- Look at the URL in the browser bar.
- Find the pattern that matches the URL among the url() definitions in `core/urls.py`.
- The second parameter of the url() definition tells where to look for the code. All of the entries so far are of the form: `plt_wbm.core.views.`<some name>. That ends up directing execution to look for the symbol <some name> inside `plt_wbm/core/views.py` (this is why those __init__.py files are in the **plt_wbm** directory and in the core directory: they allow views.py to be found using the dotted notation).
- The bunch of <some name> symbol definitions are near the end of views.py. They are assigned to <some class name>`.as_view()`, where the <some class name> part occurs earlier above in views.py also. The procedures defined under those classes will be what is executed for various portions of the interaction with buttons or other form items. The definition of what template file (or files) is (are) used by the view is located there also.

Now that the code for dealing with URLs is findable, we can now have a brief discussion that highlights how we are using Django's view classes as simple examples of how ***to*** use Django's view classes. We can then hand things over to Django's own documentation to explain how the rest of our view code works and introduce more advanced capabilities.

Once the view of interest has been found, one can then easily find the HTML template file(s) that it uses to send the response back to the browser. The simple

views have a member variable called **template_name** that is set to the name of one of the template files in `plt_wbm/templates`. More complex views override a method called **get_template_names()** and the return value is the name of the template file to use. The templates under the **`calibration-detail`** subdirectory of **`templates`** get used by this latter way.

In order to refer to our view data in the templates, we have to know what name(s) to use to refer to it to get it substituted. We can direct that in the view code by setting a member variable called **context_object_name** to the desired name we want to use in the template coding. We may refer to both a single data object this way for our single object views, or to a list of data objects for our list views.

We may also make any additional data visible by augmenting what is available to the substitution process. We do that by overriding a method called **get_context_data()**. Inside the override, we simply add the extra data and give it a name to refer to it by in the template for substitution.

Most of our views typically do what is described above. Another commonly seen method override is **post()**. This is what is called when a form is submitted from the browser, usually by clicking on a button (depending on the template).

Getting the data object(s) for the view is usually a case of either accessing **self.object** when in a single-object view, or accessing **queryset** for a list view. The availability of **self.object** does not apply to views that are for creating a new data object.

To get an idea of the order of execution of the methods typically seen in these views, consider the following narrative. Initially entering a view for an object or list of objects will get its **get_context_data()** method executed, after which the result ends up on the user's browser. Hitting a form button on the page (if it has one) will cause its **post()** method to be executed. If the **post()** redirects control back to the same view, the **get_context_data()** method will be executed again, and the user will see an updated view of the page on their browser. This pattern can go on several times with views for the control pages for calibration jobs or data taking jobs. Alternatively, the **post()** may be coded to redirect to a different view, or the user may hit a link on the page that takes them to a different view, which would cause the **get_context_data()** of the different view to be executed next (instead of the **post()** for the old view, in the case of following the link), followed by the user getting the different page on their browser.

The automated requests from the browser for "uptime" information or the data run monitoring invoke the override of the **get()** method in their respective views. They are coded only to return isolated data instead of redirecting to another whole view.

Consult “Class-based generic views” in the Django documentation, both the “topics” version and the “ref” version to see the full set of features that may be plugged-in to. The links “Overview” and “Generic built-in views”, on the “* Generic views:” line under “The view layer” on the Django version 1.3 documentation page should be used instead of following through the page-by-page links to generic views. The latter takes one through the deprecated function-based views information, where it is easy to overlook the links to class-based views along the way.

Connection to C++: Navigating Through the Python-to-C++ Interface

Now we head back over into the other great branch of the source code for the **cmsplt** project: the **`interface`** directory. Our connection to C++ begins with a C++ module that uses the boost technology for mating C++ to Python: `interface/src/common/PythonInterface.cpp`. Inside this source file is a definition for the cmsplt_interface shared library that contains boost_python interfaces for various C++ classes. That part begins with the boost_python macro, “`BOOST_PYTHON_MODULE(cmsplt_interface) {`“, which encapsulates the rest of the contents of the source file (with curly braces). The macro-looking notation used to map C++ code to identifiers used on the Python side is boost_python’s own DSEL (Domain-Specific Embedded Language), defined for this special purpose. DSELs in C++ in general are an advanced technique that uses the C++ expression template feature.

The notation for each interface definition is in the form of a class name mapping specification followed by one or more member definition mappings, which may be read-write or read-only:

```
class_<C++ class name>("class name for Python side" [, constructor parameters ]
)
    .def("member name for Python side", &C++class::C++member [, optional return
value policy ] )
    .def_readonly("same stuff", &C++class::C++member [, same stuff ] )
…
    .def("method name for Python side", &C++class::C++method [, same stuff ] );
```

Note that the `.def()` entries may specify either data members or method members.

There are other definitions for pure data, to allow C++ vectors to be iterable as Python lists, and so forth.

On the Python side of referring to the C++ entities, one instantiates interfaced C++ objects by coding:

```
thing = cmsplt_interface.SomeClass( parameters )
```

One calls interfaced C++ methods and refers to interfaced C++ data members by:

```
thingreturn = thing.some_method( parameters )
thingdata = thing.some_data_member
```

There are lots of examples of usage in action in `__init__.py`, `pltsystem.py`, `core/models.py`, `core/utils.py`, and `scripts/tests.py`.

Organization of the C++ code

In the initial rush to show where everything is for the Django side of the **cmsplt** system, saying anything about the organization of the C++ code was bypassed. Now we are now ready to find out where everything is structurally.

Mentioned in the beginning, the C++ source code lives (somewhere) under the **`interface`** subdirectory under **`cmsplt`** (where the sample svn command dumped everything).

```
drwxrwxr-x 2 mfoerste mfoerste    4096 Aug  6 02:27 bin
drwxrwxr-x 6 mfoerste mfoerste   12288 Aug  9 19:53 calib
-rwxrwxr-x 1 mfoerste mfoerste 4350375 Aug  6 02:26 cmsplt_interface.so
drwxrwxr-x 5 mfoerste mfoerste    4096 Jul 31 02:05 conf
drwxrwxr-x 3 mfoerste mfoerste    4096 Aug  9 19:55 data
drwxrwxr-x 4 mfoerste mfoerste    4096 Jul 25 23:25 include
-rwxrwxr-x 1 mfoerste mfoerste 2975725 Aug  6 02:25 libcmsplt_interface.so
-rw-rw-r-- 1 mfoerste mfoerste    1810 Jul 25 23:25 Makefile
-rw-rw-r-- 1 mfoerste mfoerste    4825 Jul 25 23:25 make_plot.C
-rw-rw-r-- 1 mfoerste mfoerste    4898 Jul 25 23:25 make_plot_vtrim.C
drwxrwxr-x 3 mfoerste mfoerste    4096 Jul 25 23:25 plots
drwxrwxr-x 3 mfoerste mfoerste    4096 Jul 25 23:25 python
-rw-rw-r-- 1 mfoerste mfoerste     194 Jul 25 23:25 README_startupgui
-rw-rw-r-- 1 mfoerste mfoerste     302 Jul 25 23:25 README_trimming
-rw-rw-r-- 1 mfoerste mfoerste     751 Jul 25 23:25 README.txt
drwxrwxr-x 3 mfoerste mfoerste    4096 Jul 25 23:25 RUScripts
-rw-rw-r-- 1 mfoerste mfoerste   10439 Jul 26 17:00 SConstruct
drwxrwxr-x 4 mfoerste mfoerste    4096 Jul 25 23:25 scripts
-rw-rw-r-- 1 mfoerste mfoerste    1174 Jul 25 23:25 setenv-slc5.sh
-rwxrwxr-x 1 mfoerste mfoerste     208 Jul 25 23:25 setup_TTCci.sh
drwxrwxr-x 4 mfoerste mfoerste    4096 Aug  6 02:27 src
drwxrwxr-x 7 mfoerste mfoerste    4096 Jul 25 23:25 startup
drwxrwxr-x 3 mfoerste mfoerste    4096 Jul 25 23:25 test
drwxrwxr-x 3 mfoerste mfoerste    4096 Jul 25 23:25 TIFScripts
drwxr-xr-x 3 mfoerste mfoerste    4096 Jan 24  2014 xdaq
```

The two subdirectories highlighted in red (**include** and **src**) are where the C++ code lives. The subdirectory highlighted in green (**bin**) is where the stand-alone programs built from the C++ code are put. The PLT Browser Interface does not use the stand-alone programs. Instead, it directly uses the C++ library code under **src/common** and **src/common/calib** via the boost_python interface. The software build process creates **cmsplt_interface.so** and **libcmsplt_interface.so** from that (seen in blue above), and that's how the Python interpreter gains access to the code. The following table shows what is under the **include** and **src** directories in parallel, describing the contents categorically and highlighting significant modules.

Table 8: C++ source directories and contents

include	**src**
ConfigFile.h, LinkDef.h **interface** Hardware support class headers (.h), RunManager.h **calib** BaseCalibration.h, Calibration (sub)class headers (.h), Data (sub)class headers (.h)	Main programs (.cpp) **common** Hardware support class impls. (.cc), RunManager.cc, PythonInterface.cpp **calib** BaseCalibration.cc, Calibration (sub)class impls. (.cc), Data (sub)class implementations (.cc)

The main program modules allow development and testing of individual calibrations by researchers (using the executables under **bin**). The RunManager module is responsible for XDAQ work loop functionality, running both calibration jobs and data taking jobs. The PythonInterface module, covered earlier, contains the Python-to-C++ class interfaces. The data taking modules currently subclass the BaseCalibration module instead of having their own base class.

All of the C++ code is compiled, and the stand-alone programs built, by using something called **scons** (The SCons Foundation, 2014), written in Python. It is a replacement for make. The instructions that scons looks for to build the project is in a Python source file called **SConstruct** (also highlighted in the **interface** directory listing above). Using **scons** to build everything is very straightforward: after sourcing the setenv.sh script in the **cmsplt** directory, cd to the **interface** directory and type **scons –j 2**.

```
# cd cmsplt
# . setenv.sh
# cd interface
# scons -j 2
```

After printing a lot to the terminal, **scons** will state at the end whether the build of all modules was successful or not. In case of success (or partial success), executable programs will exist under the **`bin`** subdirectory, and various **`.so`** files will exist in the **`interface`** directory.

Extra: Implementation of the List of Available Calibrations for New Calibration Sets

The five available calibrations to choose from on the **New Calibration Run** page, where one may create a new calibration *set*, are for now hardcoded in various source files coordinated together, instead of being concentrated into one list of information.

From the home page of the PLT Browser Interface, clicking on “Calibrations” in the navigation bar takes one to the page that shows all of the existing calibration sets. Clicking on the “New Calibration Set” button takes one to the **New Calibration Run** page. The URL for that page ends with “calibration_sets/?new=”. Looking that up in `core/urls.py` directs us to the `calibration_sets` view in `core/views.py`, which then directs us to the `CalibrationSetListView` class. That view class does not define a fixed template file for itself, but overrides the `get_template_names()` method to figure out what it is each time the view is called. The code in there asks if there is a request key called “new” in the request from the browser. In the case of the **New Calibration Run** page, there is a key called “new” in the request (it’s the “?new=” part in the URL quoted above). The name of the template file returned for having a key called “new” is `calibrations_new.html`, which is found under the `templates` directory. That template file shall be the starting point for seeing the implementation of available calibrations offered for new calibration sets. (For the curious: the other template file returned by `get_template_names()` is the one for viewing the list of existing calibration sets, which we passed through briefly on our way to the **New Calibration Run** page.)

In our starting point of `templates/calibrations_new.html`, there are five checkboxes defined, each one for a different calibration. The `value=` attribute of each checkbox tells what the Python code sees for each checked choice when the user submits the form by clicking on the “Create Calibration Set” button. As an aside, we may also note that it is embedded Javascript code in the page that allows the user to drag the checkboxes around to alter the order in which the calibrations are run. All of the dragging occurs strictly on the browser, and affects the ordering of the checkboxes seen by the software on the server end when the user finally clicks on the create button.

Table 9: Calibration interface correspondences 1 of 4: checkbox names to value attributes

Visible Checkbox Name	`value=` Attribute
Sample Calibration	“samplecal”
Ultrablack	“ultrablackcal”
Levels	“levelscal”
Gain	“gaincal”
Trimming	“trimming”

For the treatment of the `value=` attribute, code inside the `post()` method for the `CalibrationSetListView` view in `code/views.py` has a sequence of `if` tests for them, one for each of the values summarized above. Each case submits the name of a corresponding model calibration class (`core/models.py`) as the first parameter to another method of the same view, `make_calibration()`.

Table 10: Calibration interface correspondences 2 of 4: value attributes to model calibration classes

`value=` Attribute	Model Calibration Class
“samplecal”	SampleCalibration
“ultrablackcal”	UltrablackCalibration
“levelscal”	LevelsCalibration
“gaincal”	GainCalibration
“trimming”	TrimmingCalibration

Stepping inside the `make_calibration()` method next, we witness the class name parameter being used to instantiate the corresponding calibration object, which shortly gets saved as a new record in its table in the database. When the calibration job is run, the Python workloop code (`core/util.py`) retrieves the `class_name` member from the calibration job object and submits it as the first parameter to the run manager’s `queue_calibration()` method.

The `class_name` member is a **`property`** member of its calibration class (see `core/models.py`, again). That property of each respective calibration class is a method called `class_name()` that returns a string, allowing it to vary with each class.

Table 11: Calibration interface correspondences 3 of 4: model calibration classes to class_name properties

Model Calibration Class	`class_name` Property
SampleCalibration	“SampleCalibration”
UltrablackCalibration	“UltrablackCalibration”
LevelsCalibration	“LevelsCalibration”
GainCalibration	“GainCalFast”
TrimmingCalibration	“TrimCalFast”

The `queue_calibration()` method of the run manager is our entry point into the C++ code and the final link to the calibration code that actually gets used. Inside RunManager::queueCalibration() in `src/common/RunManager.cc`, we have one more transference of names. The incoming first parameter to queueCalibration() is a string (what the property function returned). Based on that, one must instantiate an object of a corresponding C++ class. This is done by a chain of `if()-else if()` statements.

Table 12: Calibration interface correspondences 4 of 4: class_name properties to C++ classes

`class_name` Property	C++ Class
“SampleCalibration”	SampleCalibration
“UltrablackCalibration”	UltrablackCalibration
“LevelsCalibration”	LevelsCalibration
“GainCalFast”	GainCalFast
“TrimCalFast”	TrimCalFast

This is the end of the journey for connecting the labeling of the web page check boxes to the C++ code that does the work. Table 13 summarizes this connection.

Table 13: Calibration interface net correspondences: checkbox names to C++ classes

Visible Checkbox Name	C++ Class
Sample Calibration	SampleCalibration
Ultrablack	UltrablackCalibration
Levels	LevelsCalibration
Gain	GainCalFast
Trimming	TrimCalFast

There is one more set of files to maintain in coordination with any changes to views, models, the **New Calibration Run** page, and the RunManager C++ code. That is the detail pages for each of the calibrations visible on the **New Calibration Run** page. They are located in `templates/calibration-detail`. It turns out that the calibration class object's `class_name` property is also used to determine which HTML template file to use when examining a calibration record in detail.

One may navigate to an existing calibration set (clicking on "Calibrations" in the navigation bar, then on any calibration set) and click on any of the calibrations listed there to look at a calibration's details. That gets us to a URL of "calibration/*<pk>*/", where the "*<pk>*" part is a number for the primary key of the `Calibration` superclass record for the calibration subclass record being examined. Each instance of a calibration record in the database consists of a `Calibration` superclass record (a record in the core_calibration table), and a record in the table for the individual type of calibration subclass. The latter record only consists of cross-referential information: the implementation of the calibration classes thus far do not specify additional data fields on top of what they inherit from the `Calibration` superclass.

Following the connection of the URL to its view code (how-to described earlier), we land in the CalibrationDetailView view. It overrides the default method `get_template_names()`, where the `class_name` property is used to construct the name of the template file to use for viewing the calibration record: "`calibration-detail/`*<class_name>*`.html`". So far, these only provide control buttons for running their respective calibrations individually.

Summarizing where everything for calibration sets is defined and located:

- Calibration labels and `value=` attribute correspondences for calibration checkboxes are defined in `templates/calibrations_new.html`.
- `value=` attributes and model class names correspondences are in `if` statements within `post()` in the CalibrationSetListView view in `core/views.py`.
- Model class names and their `class_name` property correspondences are at the tails of each calibration model's definition in `core/models.py`.
- The `class_name` properties and C++ classes correspondences are in `if()-else if()` statements in `queueCalibration()` in `src/common/RunManager.cc`.

- The calibration detail templates for the calibrations allowed in calibration sets (via `calibrations_new.html`) are in `templates/calibration-detail`. They are named using their `class_name` property: <*`class_name` property*>`.html`.

Decoupled Data Run Monitoring

Discussed earlier, the decoupled data run monitoring initially looks for and then continues to monitor a particular data file that should be growing while taking data, or not growing in the case of paused or stopped data taking. The name of the file to search for to monitor is specified as a pattern in **`settings.py`**. It needs to be a pattern due to a timestamp being part of the file name. **`DATA_RECORDING_FILENAME_PATTERN`** currently specifies the Data_Status file as the file to monitor, out of the five possible files to choose from as written by the data taking process. This is easy and convenient to change in order to switch to monitoring one of the other files, such as one with the most reliable frequency of size changes, to make change monitoring less prone to false stalling indications.

All other detail aspects of monitoring are as easily adjusted by the developer:

- The location and name of the state file for keeping track of the data monitoring state: **`DATA_RECORDING_STATE_FILE`**
- The file system path in which a new data taking output file is searched-for: **`DATA_RECORDING_ROOT`**
- The location and name of the file that stores the name of the file being monitored, once it has been found: **`DATA_RECORDING_FOUND_FILE`**
- The location and name of the file that stores the last-witnessed size of the file being monitored: **`DATA_RECORDING_SIZE_FILE`**

The entire suite of monitoring indicators has its own table of timeout thresholds in **`settings.py`** for adjusting when a new indicator changes to its “yellow alert” alter ego, and then further to its “red alert” status: **`DATA_RECORDING_AGED_INDICATORS`**. There are two other similarly purposed timeout constants for judging when “Pausing” should transition to “Paused”, and “Stopping” should transition to “Stopped”: **`DATA_RECORDING_PAUSING_TIMEOUT`**, and **`DATA_RECORDING_STOPPING_TIMEOUT`**, respectively.

The procedural code to process data run monitoring occupies the second half of **`views.py`**, starting with the **class DataRunStateService** and running to nearly

the end. It is supported by procedural code in **util.py**, taking up the last third of that module, starting with the **class DataRunState**. Finally, the Javascript code that calls the view every five seconds occurs at the definition of **updateDataRunState()** in **base.html**.

Trigger Setup Control

In the introduction to the contents of the **interface** directory, there was mention of some interesting things beside the C++ code. One of those mentioned was trigger control. This lives under three of the subdirectories under **interface**. They are **startup**, **python**, and **xdaq**. The **startup** directory contains the entry points to trigger control:

- **setup_TTCci-beam.sh** is the entry point for setting up triggers for physics data taking or van der Meer (VdM) scans.
- **setup_TTCci-cal.sh** is the entry point for setting up triggers for calibrating the PLT communications for pixel data and the read out chips' (ROCs') settings.
- **setup_TTCci-calfo.sh** is the entry point for setting up triggers for calibrating the PLT communications for its fast-or signals.
- **setup_TTCci_all.sh** is the supporting code behind all of the above entry points; it does all the work based on parameters given to it by the entry point scripts.

The other files and sub-directories under **startup** will be summarized later.

The entry point scripts are what the browser interface calls to perform trigger setup. An older pair of GUIs written in TCL also call them.

The **setup_TTCci_all.sh** script tests whether the system is advanced enough to use an XDAQ server process to control the triggers (SLC5 and later) versus having to use an older set of software that controls triggering in a stand-alone fashion (SLC4). The advanced software involves the next (and last) two of the three sub-directories for trigger setup: **python** and **xdaq**.

Two Python-coded programs exist under the **python** sub-directory (of course) to serve the triggering needs of the PLT:

- **start_xdaq.py**, as its name implies, starts up an XDAQ server process that contains an instance of the TTCciControl application. It is this application that actually controls the hardware to effect triggering.

- **xdaq_dialogue_to_enable.py** is the program that is used by the **setup_TTCci_all.sh** script to do the actual work of trigger mode changes. It talks to the TTCciControl application in the XDAQ server process via SOAP (Simple Open Access Protocol **(XML)**) messages over HTTP. The exchange between it and the application is shown at the bottom of the flowchart in Figure 39.

The other files in the **python** subdirectory are just three tests of running their respective C++ calibration classes from Python using the Boost Python interface.

The **xdaq** sub-directory is finally involved by holding files used by the Python trigger setup programs. The XDAQ startup program generates an XML configuration file for the XDAQ server process itself from internally hardcoded data and substitutions from the current attributes of the surrounding computer system. It puts the configuration file into the **xdaq** sub-directory under the name **XDAQConfigPLT.xml** and tells the XDAQ process to read that file for configuration when it starts up. The log file for the XDAQ server process is also written into the **xdaq** sub-directory, under the name **plt_xdaq.log**.

The three **.txt** files in **xdaq** are configuration files for the TTCciControl application, a combination of settings and macro definitions. They contain the coding for the way triggering works for data taking vs. general calibration vs. fast- or timing calibration for the PLT. They are referred to by the XDAQ dialogue program when it must tell the TTCciControl application to change its configuration. The control application reads the specified file when told to do so. One of them is specified as the default in the XDAQ configuration when XDAQ is started up by the other program.

Both of the trigger setup programs also make reference to the **settings.py** file in the Django part of the browser interface, even though the trigger setup programs do not use any of Django itself. This is to allow a computer system other than the local system and default port to be specified to run the XDAQ server and to communicate with it. This is defined by **XDAQ_SERVER_HOST** and **XDAQ_SERVER_PORT** in the settings file. They must be left undefined to allow the local computer and default port to be used, respectively.

The **start_xdaq.py** program takes no arguments. It gets all of its information from **settings.py** and the local system and environment. The dialogue program shows the options and arguments to use with it when it is run without arguments on the command line.

The Rest of the Interface Branch

Describing the rest of what's under the **`startup`** subdirectory of **`interface`** was deferred earlier in order to concentrate on describing trigger setup software. Resuming this thread, one finds that the other files and sub-directories under **`startup`** are an entire previous/parallel generation of PLT startup and setup technology, composed of shell scripts and data files of command line parameters (the latter fed to the **cmdLine** program built from C++). They do what the browser interface does from Python code when powering up and starting new global runs. Three of the four sub-directories contain site-specific variations for the various test stand setups used throughout the history of PLT research, development, and testing. They are:

- **`P5params`** for Point Five, the location of CMS at the LHC.
- **`RUparams`** for the test stand at Rutgers University.
- **`TIFparams`** for the test stand in the Test Integration Facility, a clean room in building 186 on CERN's Meyrin campus.

The **`CASSETTE_SETTINGS`** subdirectories under the site-specific sub-directories further contain files dealt with by some of the C++ coded calibration programs for pre-trim calibration and a few others. Maintainers may search for **`CASSETTE_SETTINGS`** among the calibration source files to discover the full details.

The **`scripts`** sub-directory contains an interesting item mentioned after the general introduction and after introducing the trigger setup entry point scripts: an earlier GUI for the PLT (actually two of them). They are written in TCL, and called **`basicgui.tcl`** and **`advancedgui.tcl`**. They present buttons for running the powerup/setup scripts under **`startup`** mentioned above. The **`calib`** sub-sub-directory contains scripts for running various C++ calibration programs, also under the command of the TCL GUI. All the other shell scripts, ROOT scripts (*.C) and Python scripts immediately under **`scripts`** are run directly on the command line.

The **`RUScripts`** (Rutgers) and **`TIFScripts`** (Test Integration Facility) sub-directories contain much earlier versions of the scripts in the **`startup`** sub-directory. They are relatively defunct. The **`RUScripts`** versions appear to have preceded the **`TIFScripts`** versions.

The **`test`** subdirectory contains Python tests of the Boost interface to the C++ code.

The **data** subdirectory is where data taken from the PLT is stored. The **calib**, **conf**, and **plots** sub-directories are used to store the three categories of output from calibrations when run under the browser interface. Figure 53 shows the pattern of data storage.

Of major note, the **conf** subdirectory also happens to be where all the configuration data anchor files of the system reside, as shown in Figure 42. Not shown in that figure is another category of XML data files, collectively referred to as DAC Map files, that are internally referred to by the figure-mentioned **rocmap.xml** data file. They carry the same kind of information as the **.dacs1** files indicated in the figure, but encoded in XML instead of as Cosmo format (unlabeled pairs of integers on each line). The information is all of the settings for each corresponding pixel read out chip (ROC), aside from pixel trim values.

Further Information

A very detailed resource for the developer, as it should be, is the source repository's change log. Each change to the browser interface code, and associated changes to other parts of the PLT interface that it runs or relies upon, has been accompanied by a mini-thesis of its own on the topic. To get the log file from the repository (all as one command line, ignoring the line breaks caused by this document's formatting):

```
$ svn log –v svn+ssh://youruserid@svn.cern.ch/reps/cmsplt/trunk
>svnlog.txt
```

…where *youruserid* is one's CERN single sign-on user id. The example file name **svnlog.txt** shall contain the full log.

References

Dawes, B., Abrahams, D., & Rivera, R. (2013, November 11). *Boost C++ Libraries*. Retrieved from Boost C++ Libraries: http://www.boost.org

Django Software Foundation. (2014, March 2). *The Web framework for perfectionists with deadlines | Django*. Retrieved from django: https://www.djangoproject.com

Python Software Foundation. (2014, March 3). *Python*. Retrieved from Python: http://python.org

The SCons Foundation. (2014, March 2). *SCons: A software construction tool*. Retrieved from SCons: A software construction tool: http://www.scons.org

CHAPTER VI
A RUN-THROUGH OF USING THE INTERFACE IN OFFLINE/TEST STAND MODE

Abstract

A slide show of screen shots depicting the usage of the PLT browser interface.

Introduction

When one first directs their browser to the URL for the PLT Browser Interface, this is what one sees on the Home page.

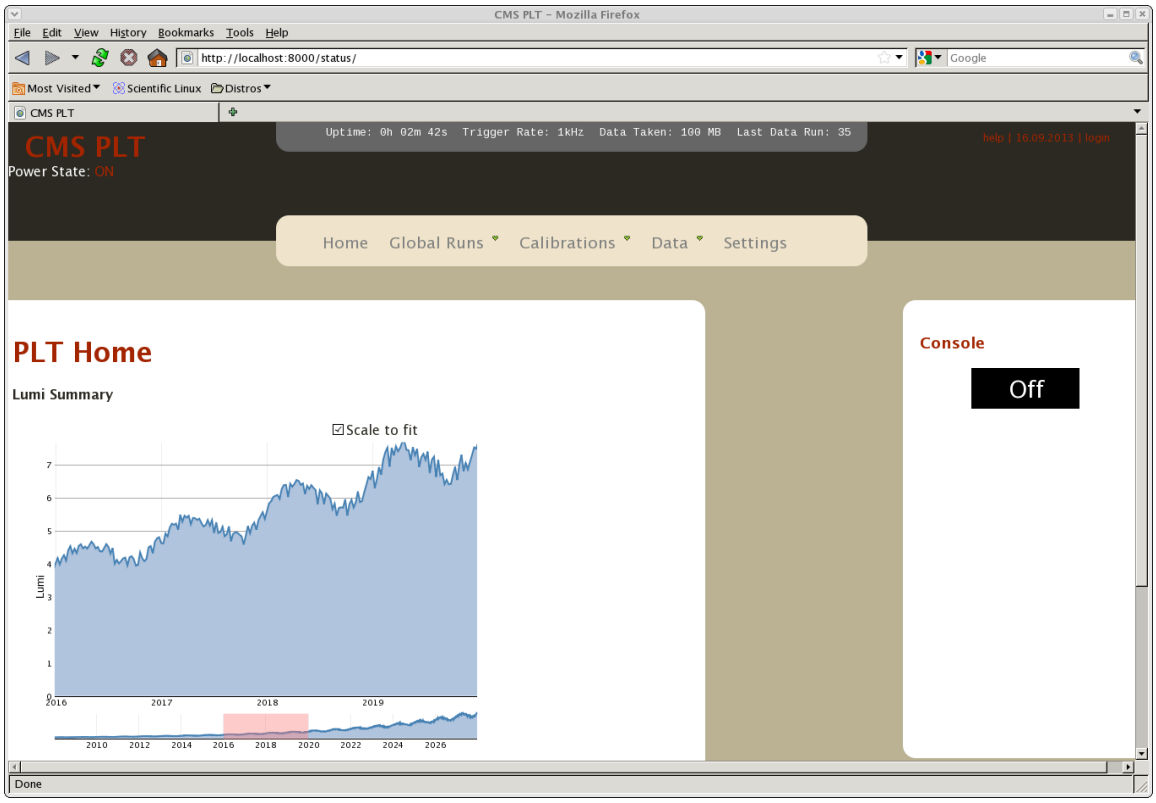


Figure 54: The home page of the PLT browser interface

In Figure 54, we see that the PLT is powered on and the console indicator for data runs shows that data taking is **Off**.

Creating and Running Data Taking Jobs

To start taking data, click upon the Data menu link in the navigation bar (shown circled and enlarged in Figure 55 below).

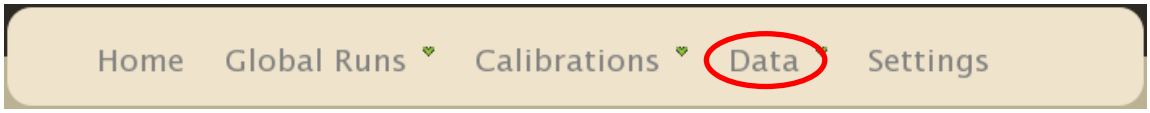


Figure 55: The navigation bar with the Data link highlighted

Since no data taking is currently active, the interface invites one to create a new data run. If data taking was not **Off**, then one would be taken to the control page for the current run. The latter shall be shown further along.

The first order of business here is to associate our new data run with a calibration set using the widget that is just above the Create button below (Figure 56).

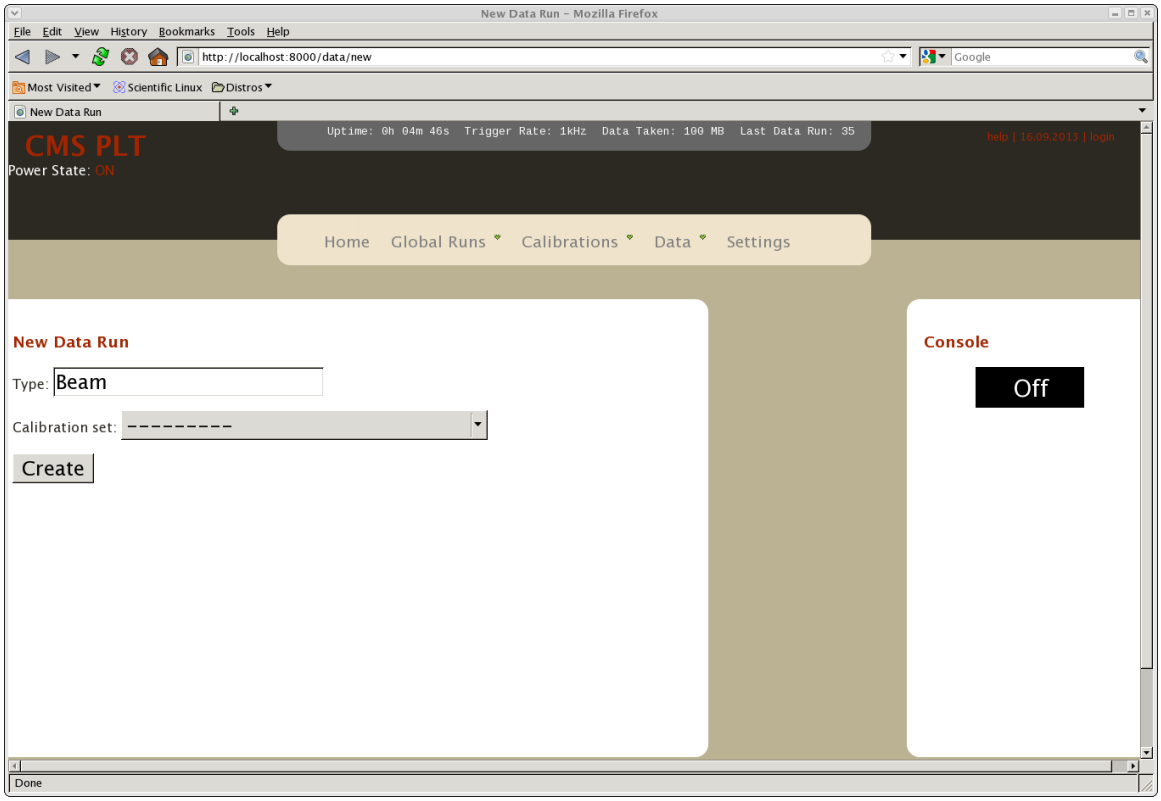


Figure 56: Starting a new data run

Here we see there is a long list of calibration sets to choose from, with one highlighted. We'll click on that one.

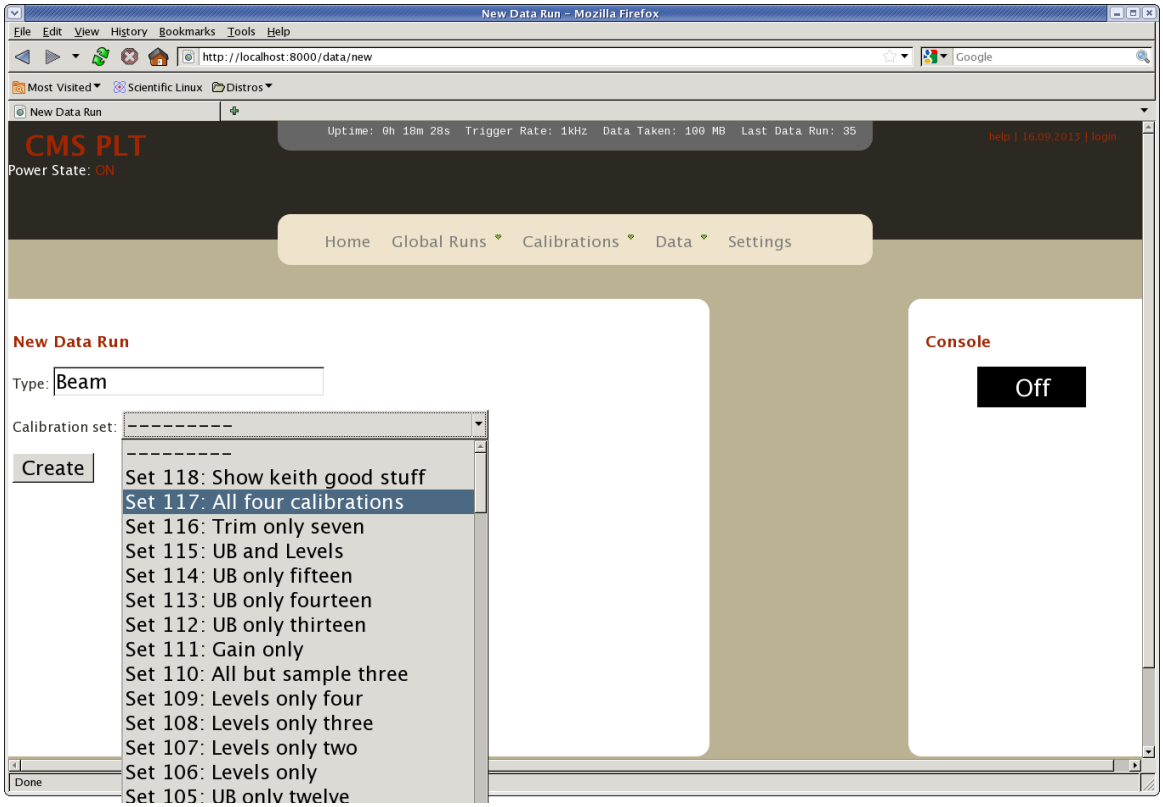


Figure 57: Choosing a calibration set for a new data run

The calibration set is now chosen. We may hit the Create button.

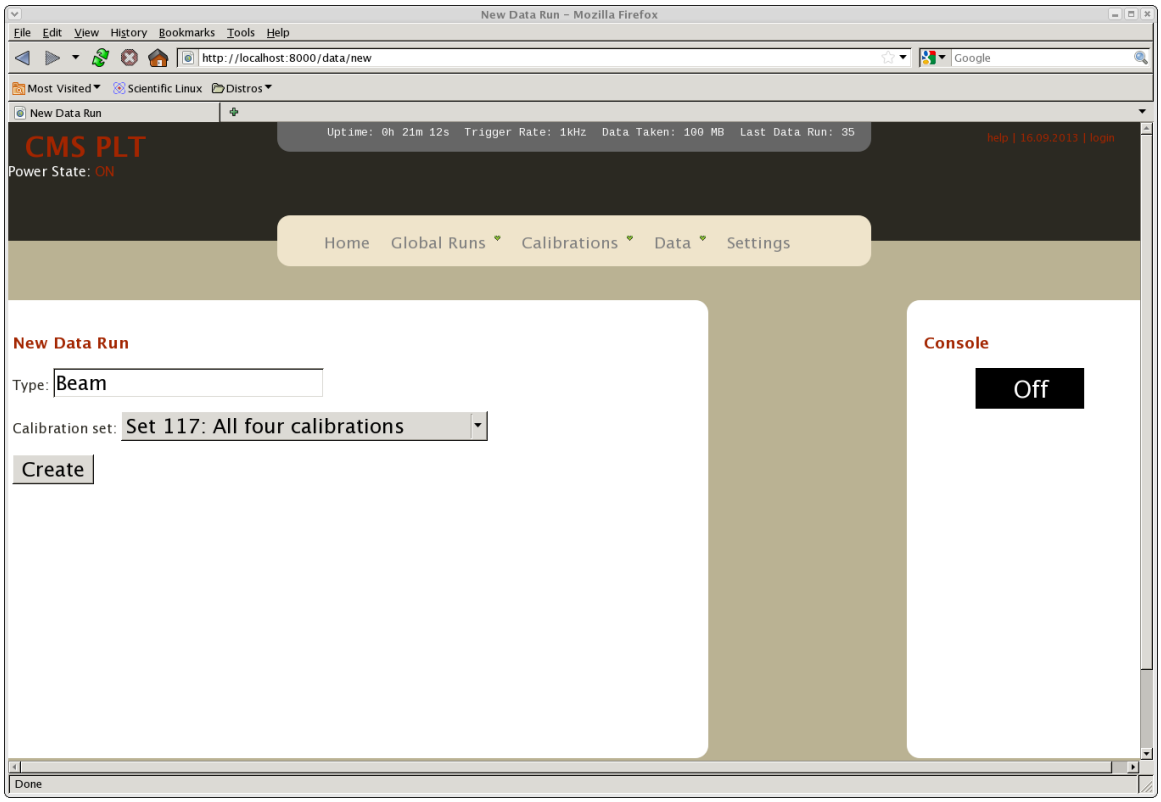


Figure 58: Create the data run

With a new set of records established behind the scenes, we may now initialize the data run. This step ensures that the triggers for data taking are set up and running. Click on the initialize button.

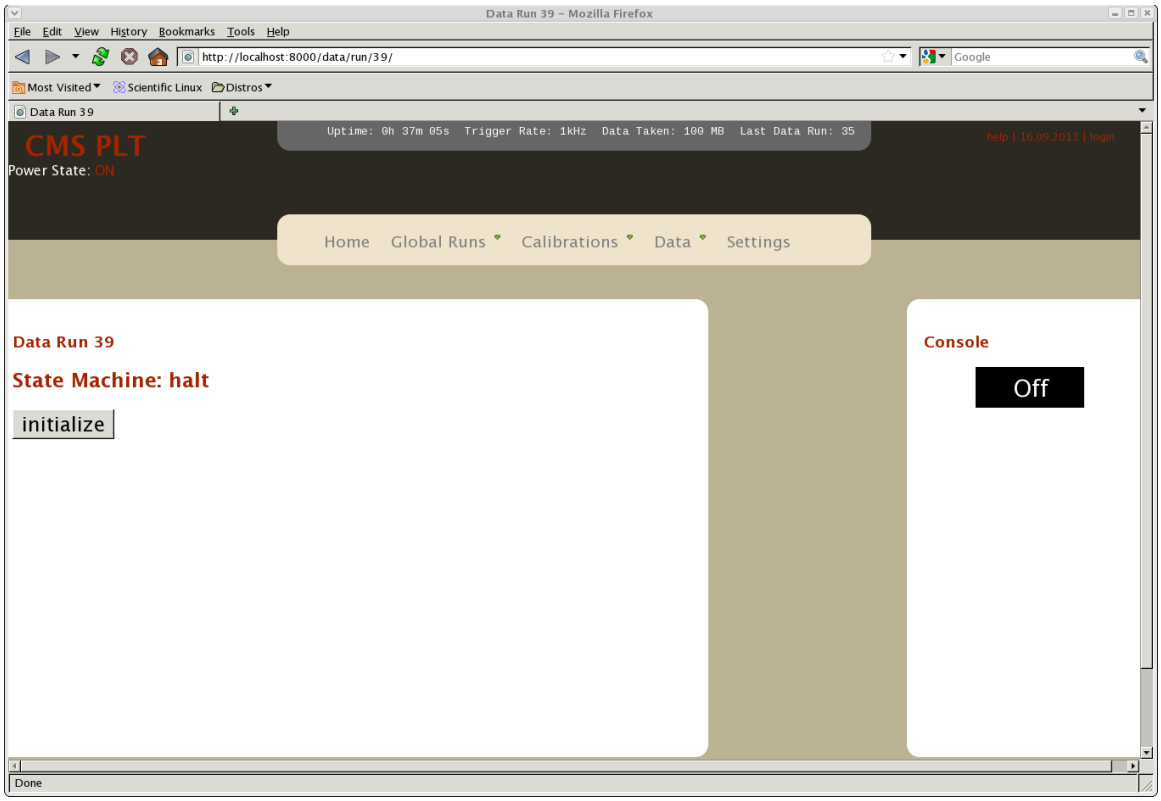


Figure 59: Initialize the data run

Now one may start the program that takes the data: press the Start button.

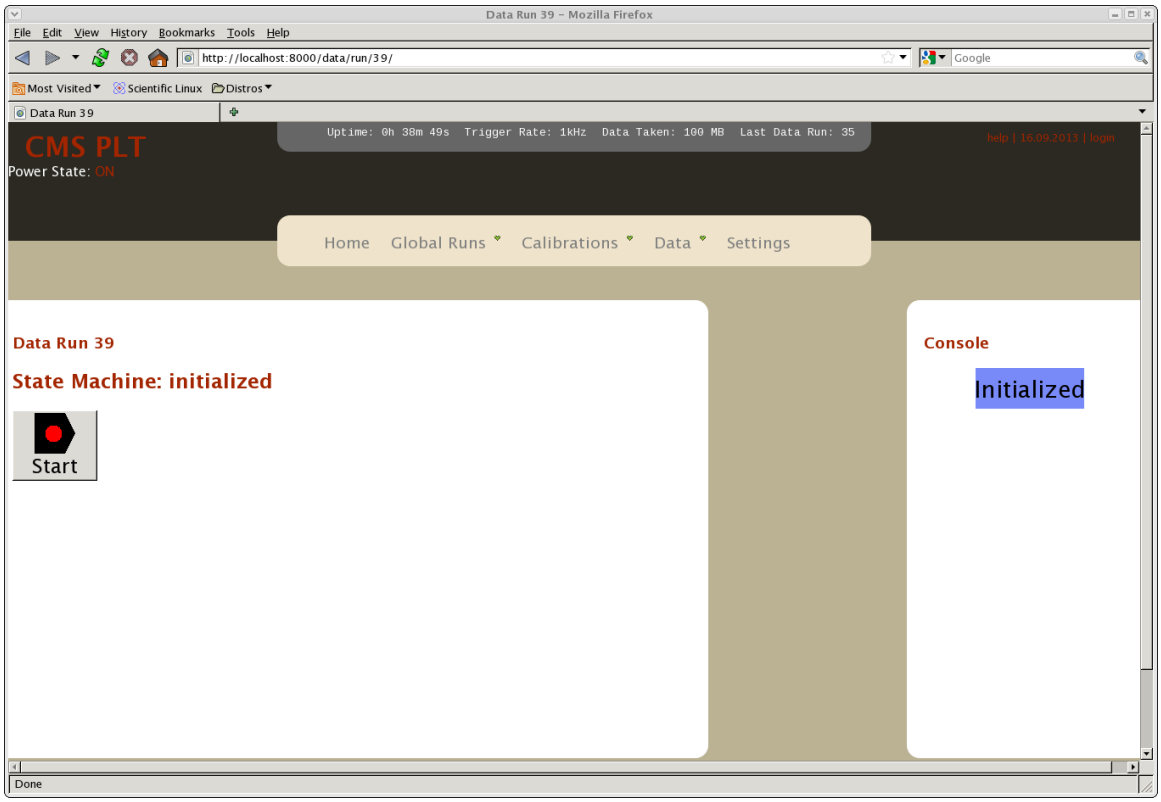


Figure 60: Start the data run

This is what the running state looks like: a choice of Pause or Stop buttons, and a pleasant green status indicator showing that the data taking process is indeed **Running** (the output files are showing changes each time they are examined).

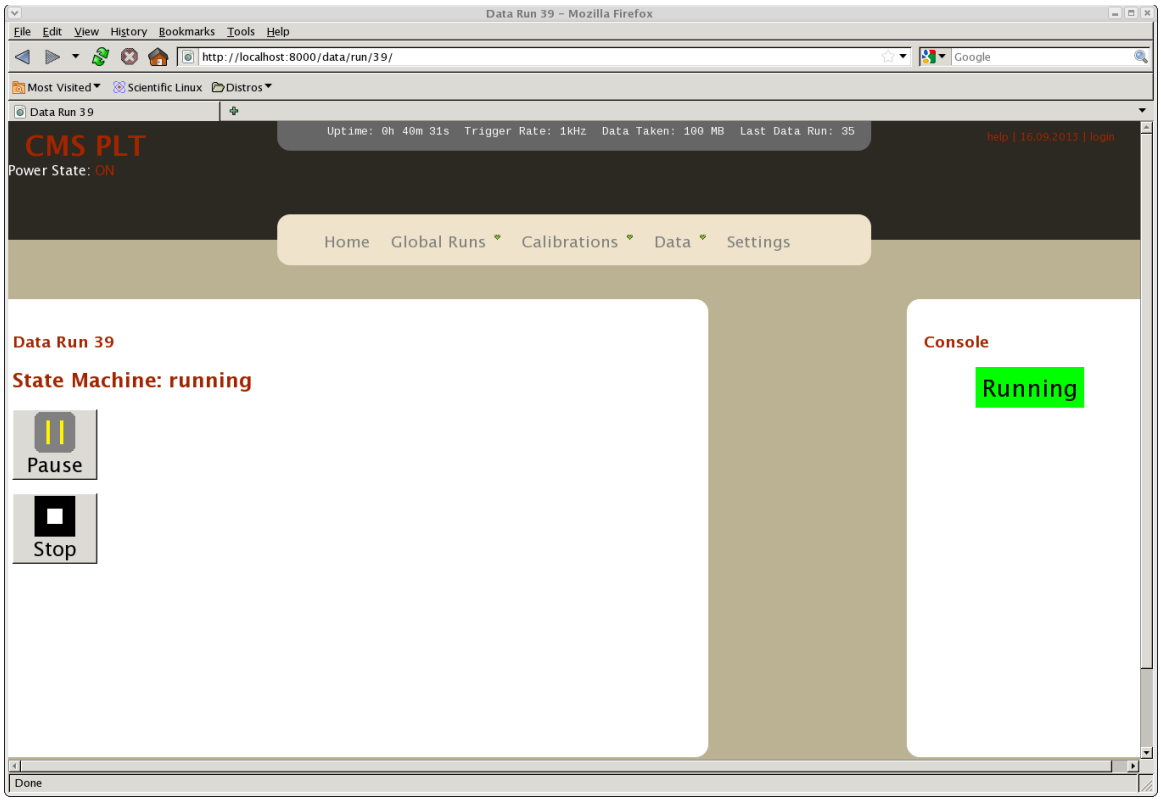


Figure 61: A running data run

If the data triggers are not firing, then one gets the unfortunate result below for the running state.

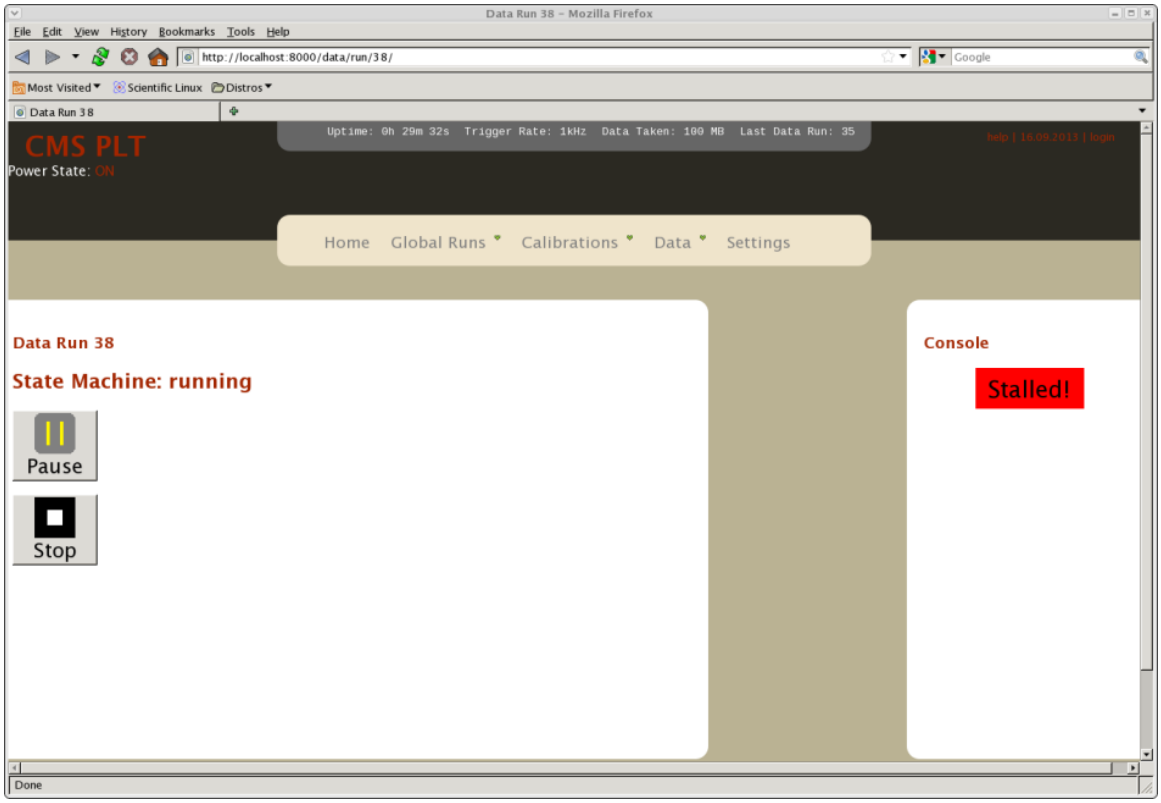


Figure 62: A stalled data run

If one switches away to some other PLT Browser page, such as the Home page (via the Home link on the navigation bar), one is still able to observe the data taking indicator at a glance.

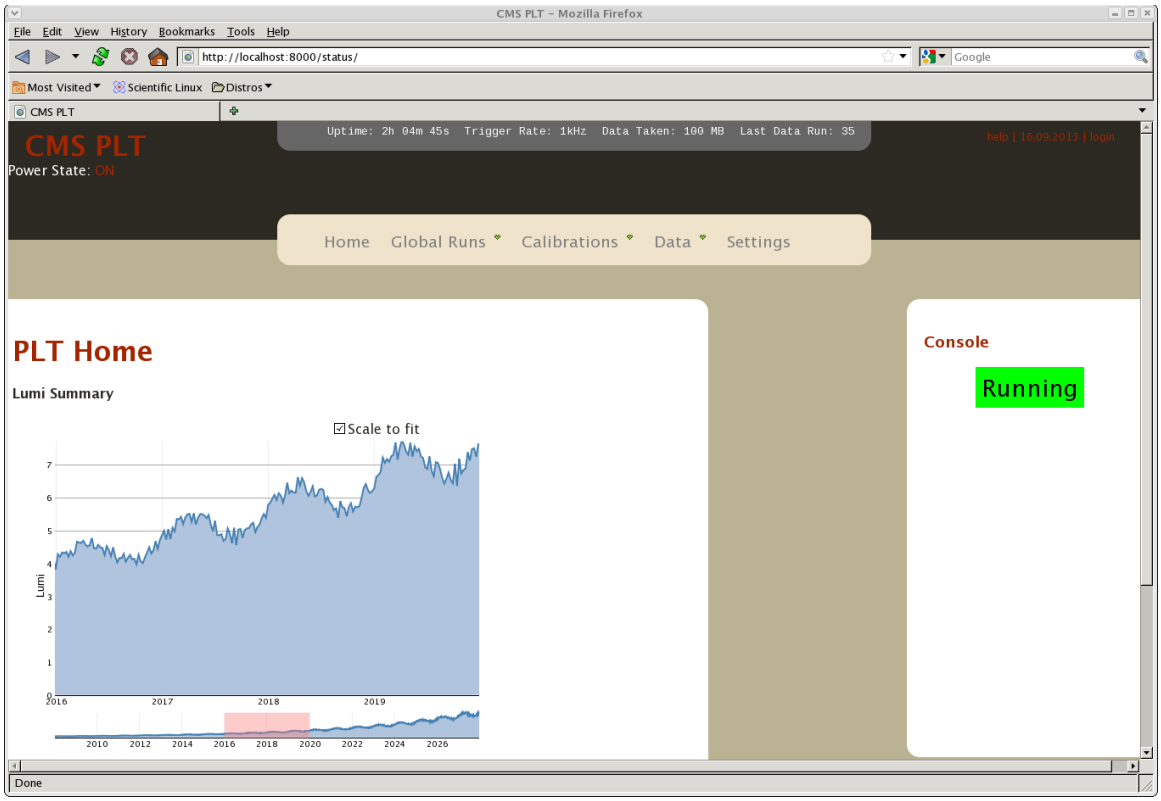


Figure 63: The data taking indicator is visible on the Home page

If one clicks upon the navigation bar's Data link as before, one is taken back to the control page for this data run.

From there, one may choose to Pause the data run…

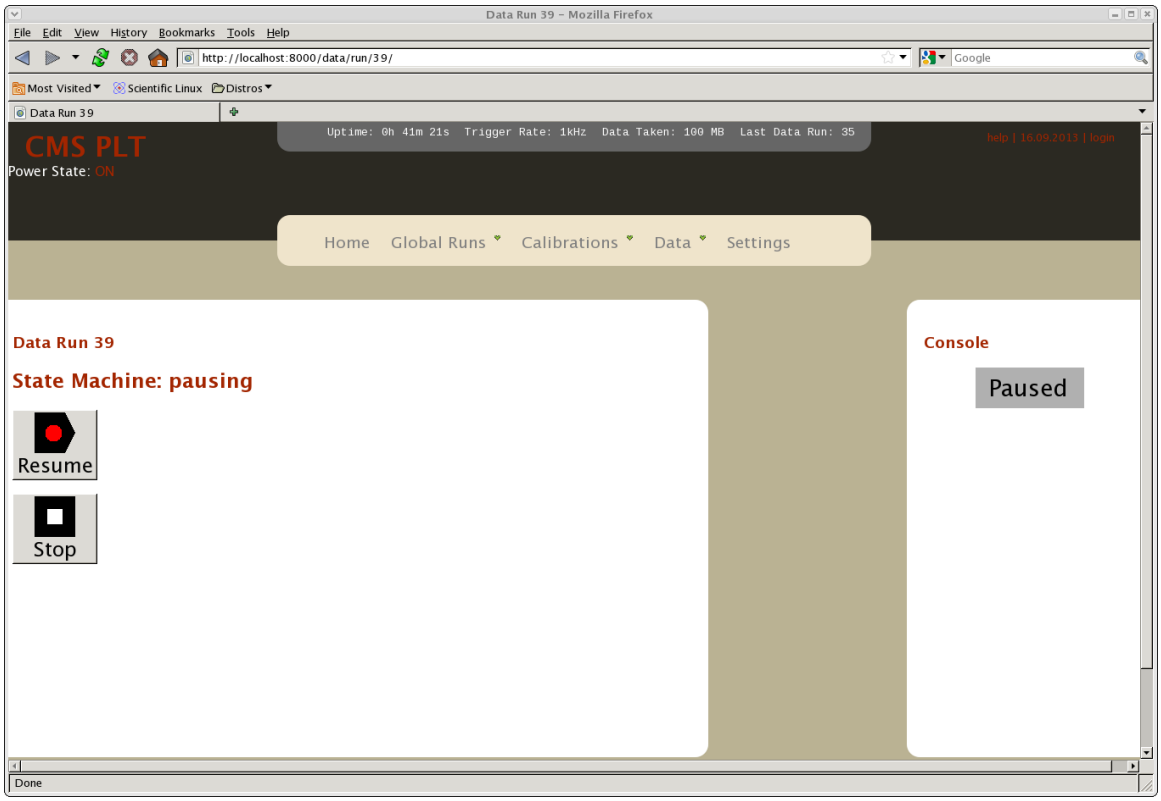


Figure 64: A paused data run

…and either Resume the data run or Stop it.

This is what the resumed state looks like. It is exactly identical to the regular running state.

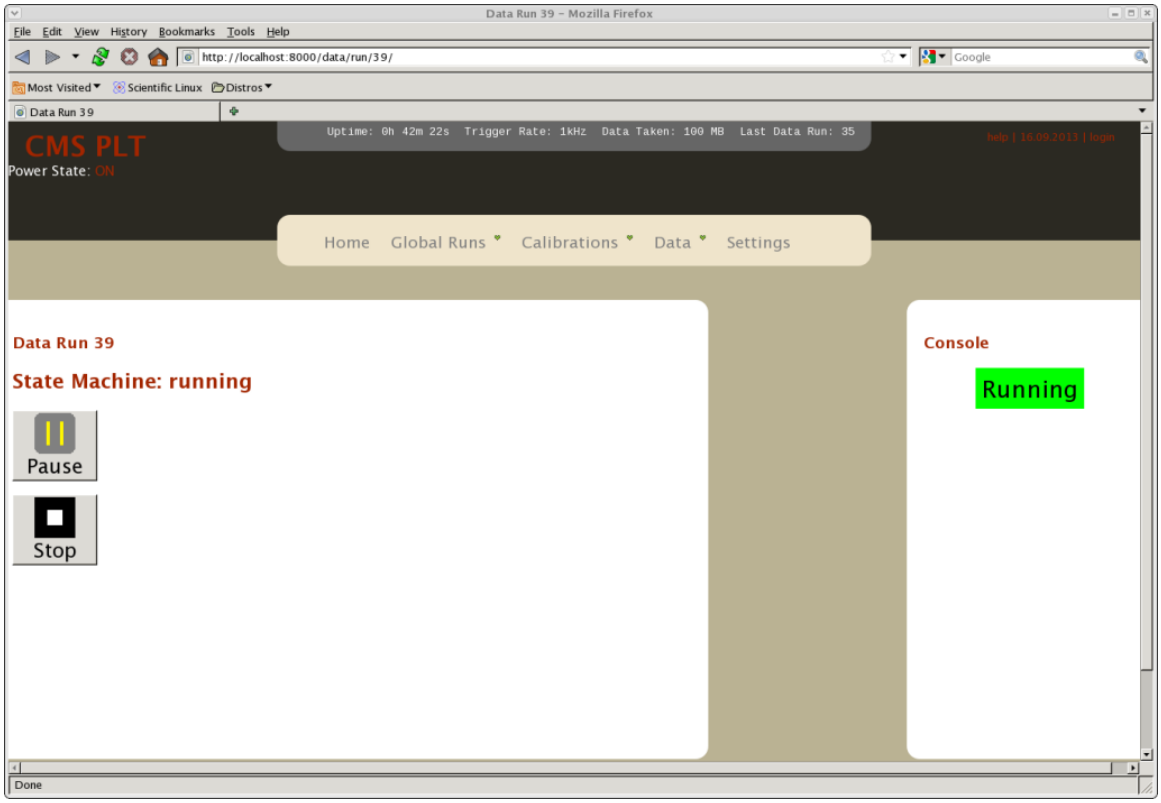


Figure 65: Resumed data run

When one chooses Stop, the following page is seen. At this point one may press the New/Current Data Run button to go through creating and starting a new data run, or click somewhere else to do something else. The "Current" part of the button's name merely reflects the fact that one may inspect an older data run that already exists while data taking for the latest data run is still happening. Pressing the button in that circumstance will take one back to the control page for the currently active data run. The button does *exactly* the same thing as the Data link in the navigation bar: it looks for an active data run and takes one to the controls for it if there is an active run, else it puts one into the creation of a new run.

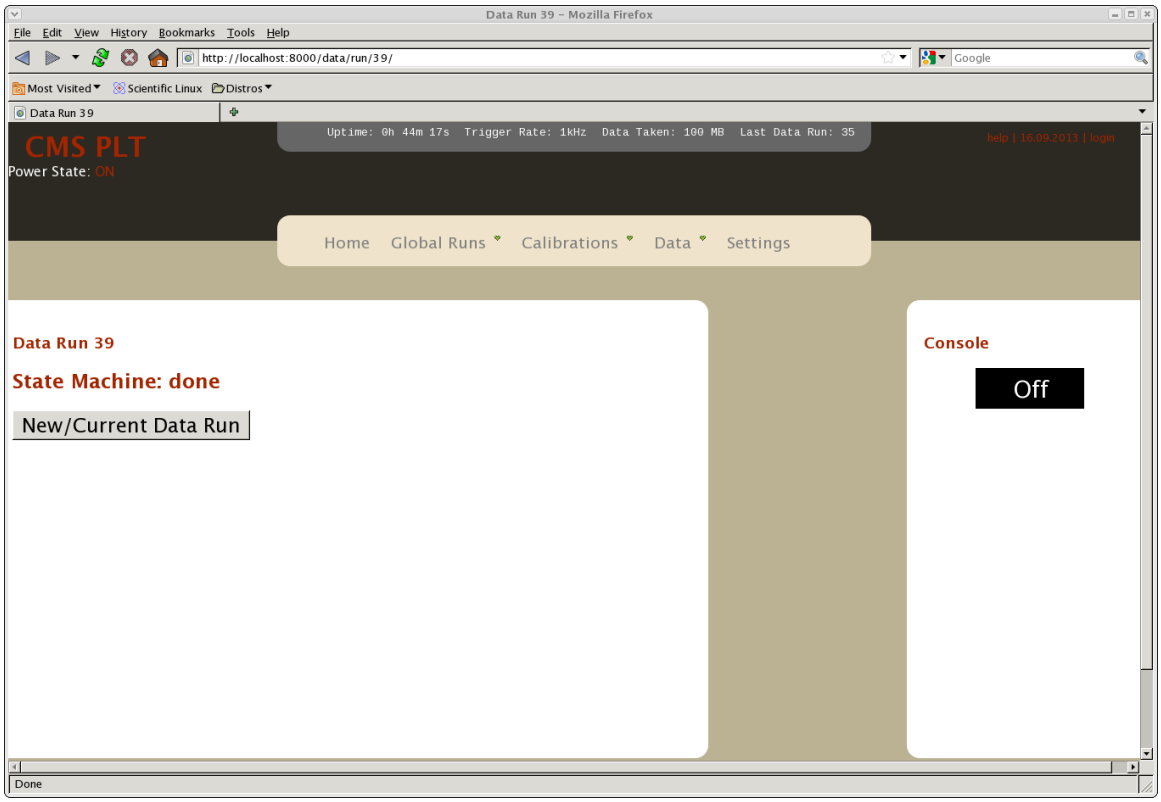


Figure 66: After stopping a data run

So far through this tour of data taking, we have seen only a handful of the console data taking indicators. Here is the full set, along with when they occur and for how long and what they mean.



Figure 67: Data taking Off

This shows when data taking is not being performed.



Figure 68: Gray Initializing...

This is a transient indicator at the very beginning of new data run initialization. It is usually too brief to be seen, and will be superseded by the first check on initialization progress.



Figure 69: Yellow Initializing...

This is a “yellow alert” warning indicator that appears if initialization takes an unexpectedly long time to complete.



Figure 70: Initialization Stalled!

This is a “red alert” indicator that initialization is taking an excessively long time and may never complete. It warrants investigation.



Figure 71: Cyan Starting...

This is a transient indicator that occurs immediately after pressing the Start button on the data run's control page. It will be superseded by the results of the first check on the data taking process.



Figure 72: Yellow Starting...

This is the indicator shown when the checks so far have not detected that the data taking process has produced any output at all; it is taking an unexpectedly long time ("yellow alert" for startup).



Figure 73: Start Stalled!

The "red alert" indicator for starting a data run. Nothing has shown up as output from the data taking process for way too long (and may never do so), and may be a system problem to be investigated.



Figure 74: Running

This is the indicator for normal running. Every check on the progress of data taking shows changes to its output.



Figure 75: Stalling...

This is the “yellow alert” indicator for data taking. There has been no change to the data taking output for a while.



Figure 76: Stalled!

This is the “red alert” indicator for data taking. There has been no change to the data taking output for an excessively long time, and should be investigated.



Figure 77: Dark gray Pausing...

This is the transient state indicator displayed immediately after pressing the Pause button on the data run’s control page. It will be superseded by the results of the next check upon the data taking process.



Figure 78: Yellow Pausing...

This is the “yellow alert” indicator for pausing the data taking process. It tells us that the data taking process is still changing its output for a while after having been told to pause.



Figure 79: Not Pausing!

This is the “red alert” indicator for pausing the data taking process. It tells us that the process does not seem to have gotten the message to pause and is still happily churning away taking data and changing its output. Investigation is required.



Figure 80: Paused

This is the indicator for having successfully entered the pause state for data taking. The data taking process appears to no longer be changing its output. Note that if a later check finds that the data taking process has “burped”, changing its output unexpectedly, the indicator will shift to either the yellow alert **Pausing…** indicator or the red alert **Not Pausing!** indicator above (depending on the time since pressing the Pause button). It will shift back to this one when the next check finds no changes to the data taking process’ output again.



Figure 81: Dark gray Resuming...

This is the transient state indicator displayed immediately after pressing the Resume button on the data run’s control page. It will be superseded by the results of the next check upon the data taking process.



Figure 82: Yellow Resuming...

This is the “yellow alert” indicator for resuming the data taking process from being paused. It tells us that no changes to the data taking process’ output have been observed for a while, so the data taking process has not yet resumed after having been told to resume.



Figure 83: Resumption Stalled!

This is the “red alert” indicator for resuming the data taking process from being paused. It tells us that no changes to the data taking process’ output have been observed for an excessively long time, so the data taking process does not seem to have gotten the message to resume. Just as with the **Stalled!** indicator for the running state, investigation is required.

Once any changes to the data taking process’ output have been observed, the state changes permanently back to running and the green **Running** indicator will be displayed. Any checks that find a lack of changes will result in the appropriate yellow alert **Stalling…** or red alert **Stalled!** indicator being displayed (none of the resumption set of indicators).

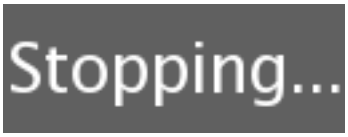


Figure 84: Dark gray Stopping...

This is the transient state indicator displayed immediately after pressing the Stop button on the data run’s control page. It will be superseded by the results of the next check upon the data taking process.



Figure 85: Yellow Stopping...

This is the “yellow alert” indicator for stopping the data taking process. It tells us that the process’ output has been continuing to change for a while after having been told to stop.



Figure 86: Not Stopping!

This is the “red alert” indicator for stopping the data taking process. It tells us that the process does not seem to have gotten the message to stop and has been changing its output for an excessively long time. Just as with the **Not Pausing!** indicator, investigation is required.



Figure 87: Stopped

This is a transient indicator for the data process having seemed to be stopped, no longer changing its output. It should be superseded shortly by database processing to the **done** state, which will show the **Off** indicator. Note that if a later check finds that the data process has “burped”, changing its output unexpectedly, the indicator will shift to either the yellow alert **Stopping…** indicator or the red alert **Not Stopping!** indicator above (depending on the time since pressing the Stop button). It will shift back to this one when the next check finds no changes to the data taking process’ output again.

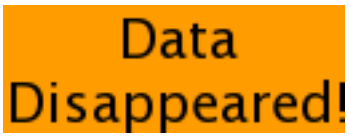


Figure 88: Data Disappeared!

This is an indicator for a peculiar condition. The condition could occur after data monitoring was already into one of the states past initialization (running, pausing, resuming, or stopping), so the output file for the data taking process had already been found and was known to the system. Somewhere along the way, that file disappeared, so the check on it could not find it. This is the indicator returned for that circumstance. This is very abnormal and must be investigated. Note that if the file spontaneously reappears, one of the normal suite of indicators shall return as well. Unexpected coding on the part of the data taking process to remove or rename or relocate the monitored file is a foreseeable cause of this malfunction, apart from external interference.



Figure 89: Lost Name!

This is the second indicator for another peculiar condition. This is complementary to the **Data Disappeared!** indicator; it happens if the *name* of the file to monitor is lost. Specifically, there's a file that contains the name of the file to monitor (the file to monitor is the data taking process' output). If that file-name-containing file disappears, that loses the name for us. This is also very abnormal and to be investigated. Spontaneous restoration of the name-containing file will also restore one of the more normal indicators. External interference by moving or deleting the name-containing file is the only foreseeable possible cause for this malfunction.

The collection of previous data runs can be accessed via a drop-down menu toggled by a widget to the right of the Data link in the navigation bar.

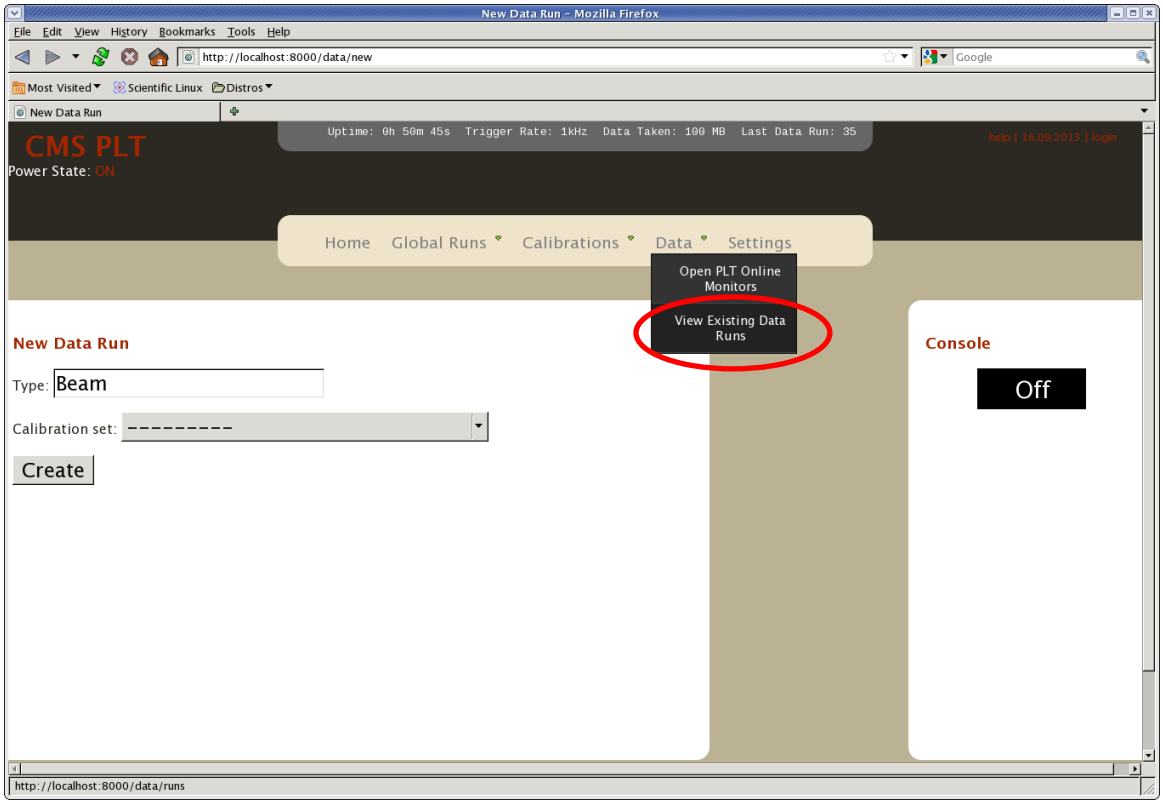


Figure 90: Menu link to view existing data runs

The View Existing Data Runs choice in the menu, circled in red, may be clicked to get to the data run list page.

The other drop-down menu selection for the Data link, Open PLT Online Monitors, is not functional.

The links for individual runs are the run numbers in the left most column.

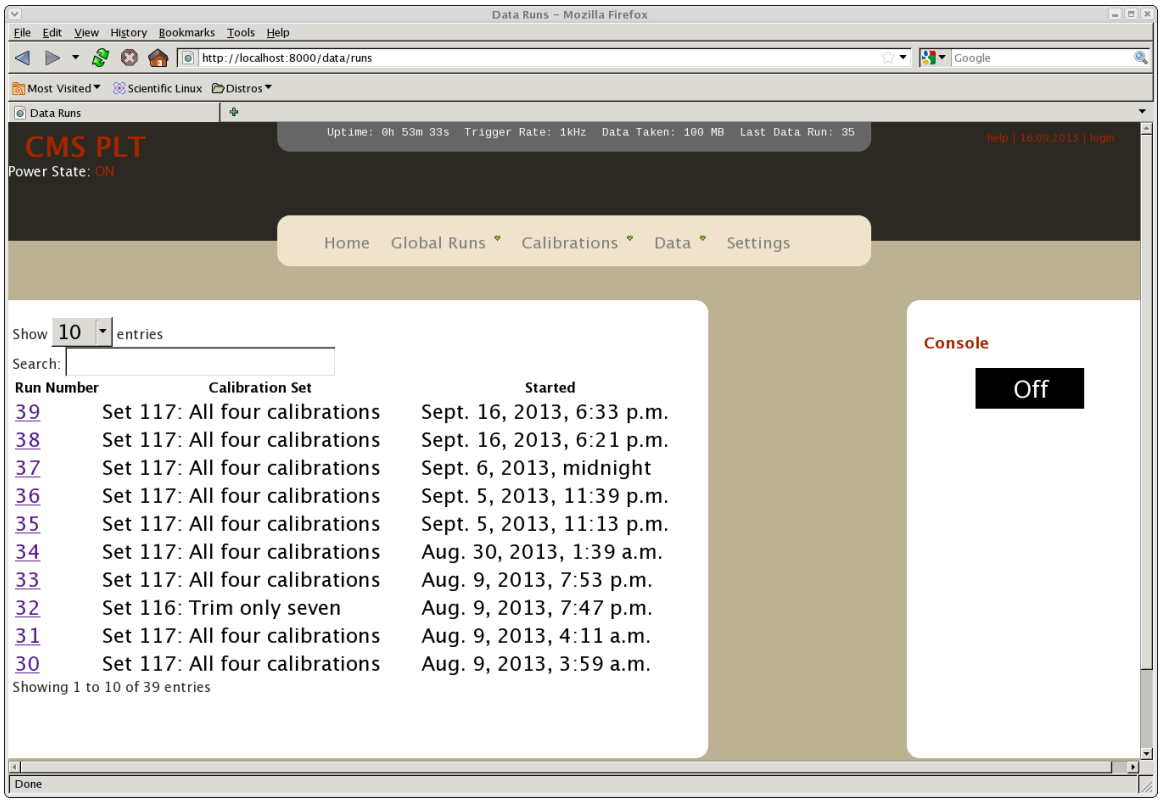


Figure 91: Listing of past data runs

Trying one of the links takes one to the individual run's page (trying 38)…

It is the same as the **done** page seen after one stops a run.

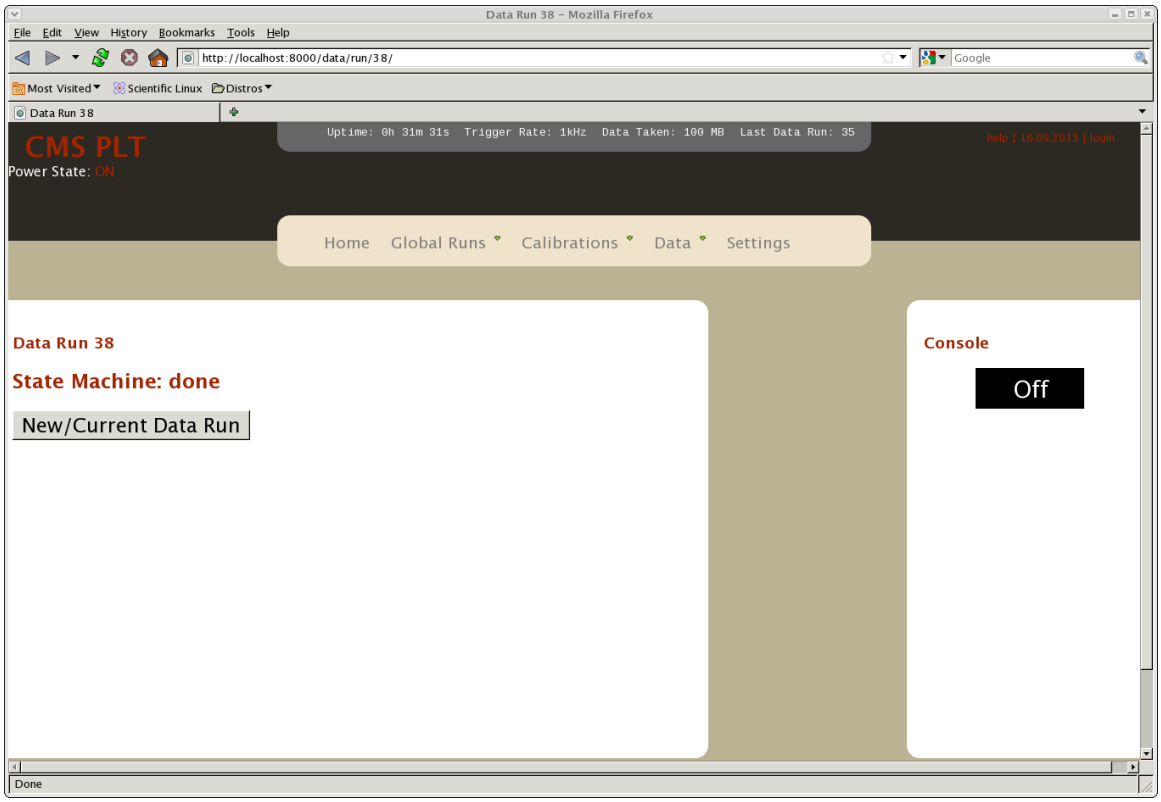


Figure 92: A past data run

Creating and Running Calibration Jobs

From anywhere, click upon the Calibration link on the navigation bar, enlarged and isolated below.

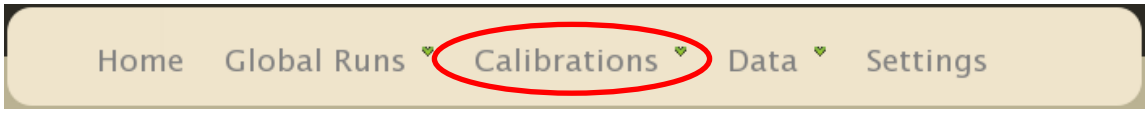


Figure 93: The navigation bar with the Calibration link highlighted

Arriving at the page below, press the New Calibration Set button

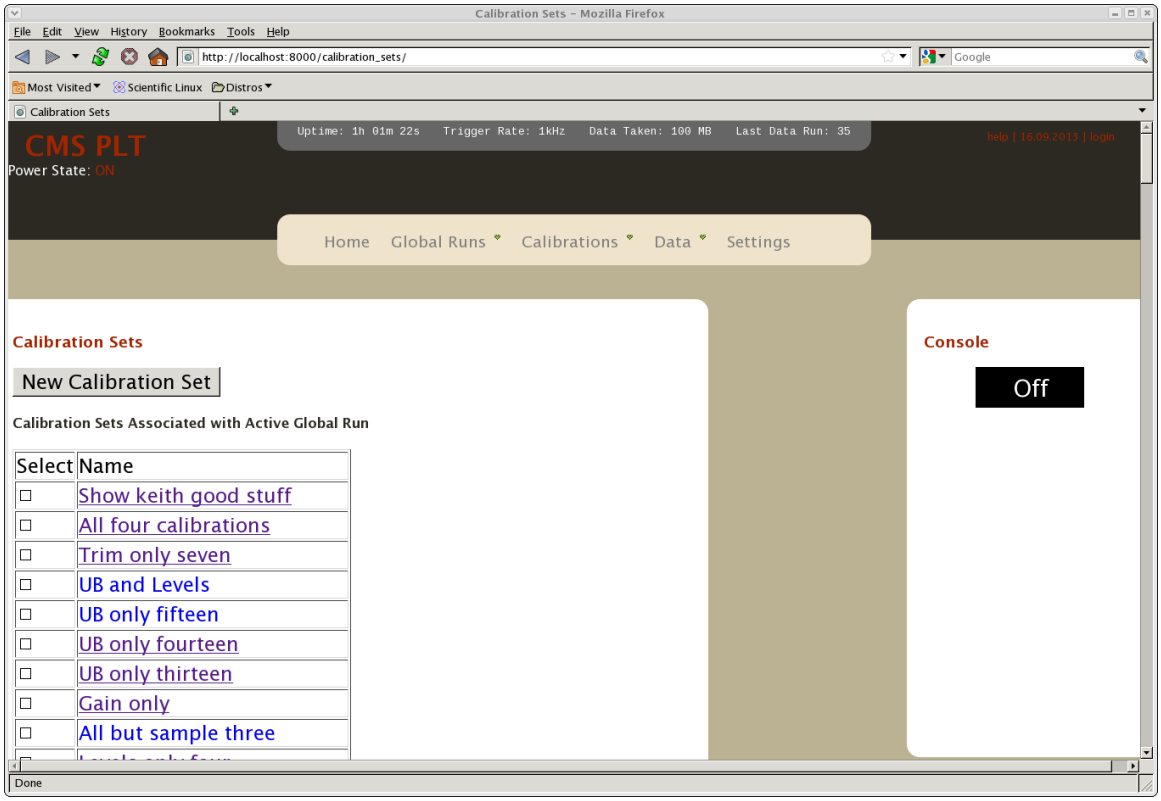


Figure 94: List of calibration sets

This is the new calibration set page. The name for the calibration set is defaulted to the creation date and time. Place the cursor in the Calibration Set Name field (circled in red) and change it to something more memorable and apropos.

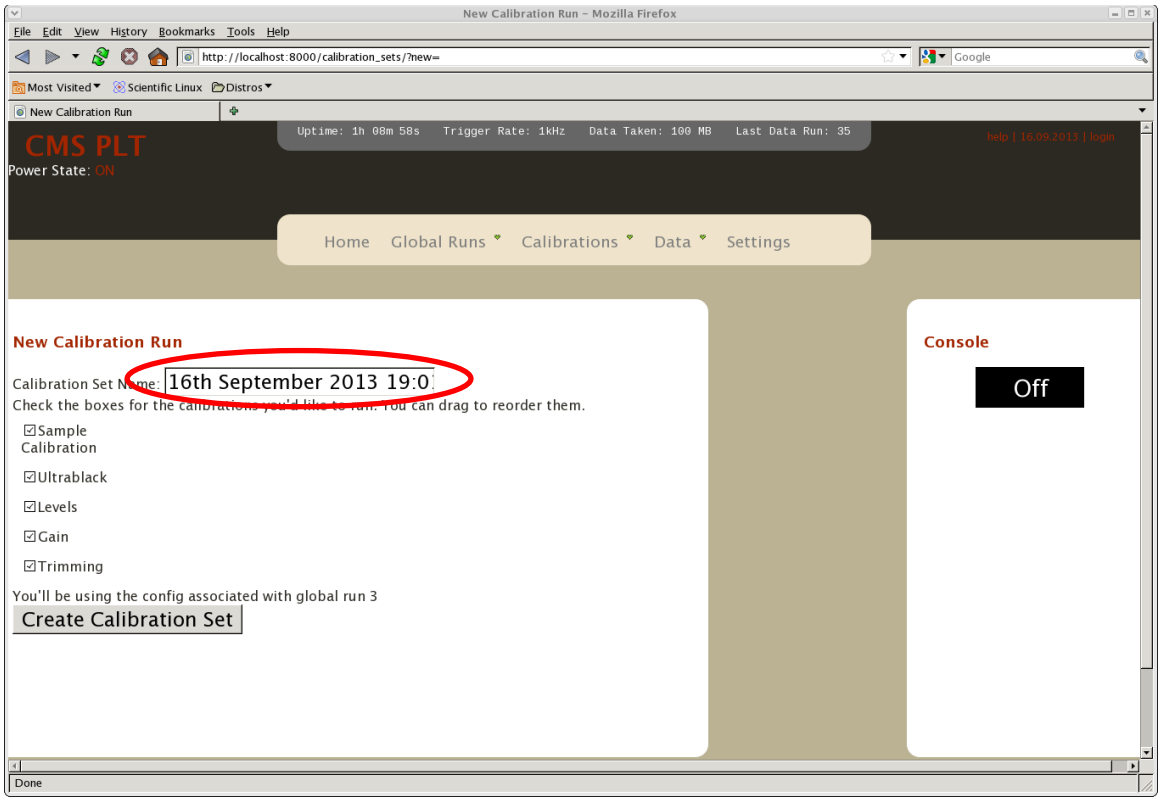


Figure 95: New calibration set: edit the name

In this example, the new calibration set has been given a name of “All Four Again”. Another default for new calibration set creation is to have all the supported calibrations selected. Our next step is to unselect the individual calibrations that we do not want to run for this calibration set.

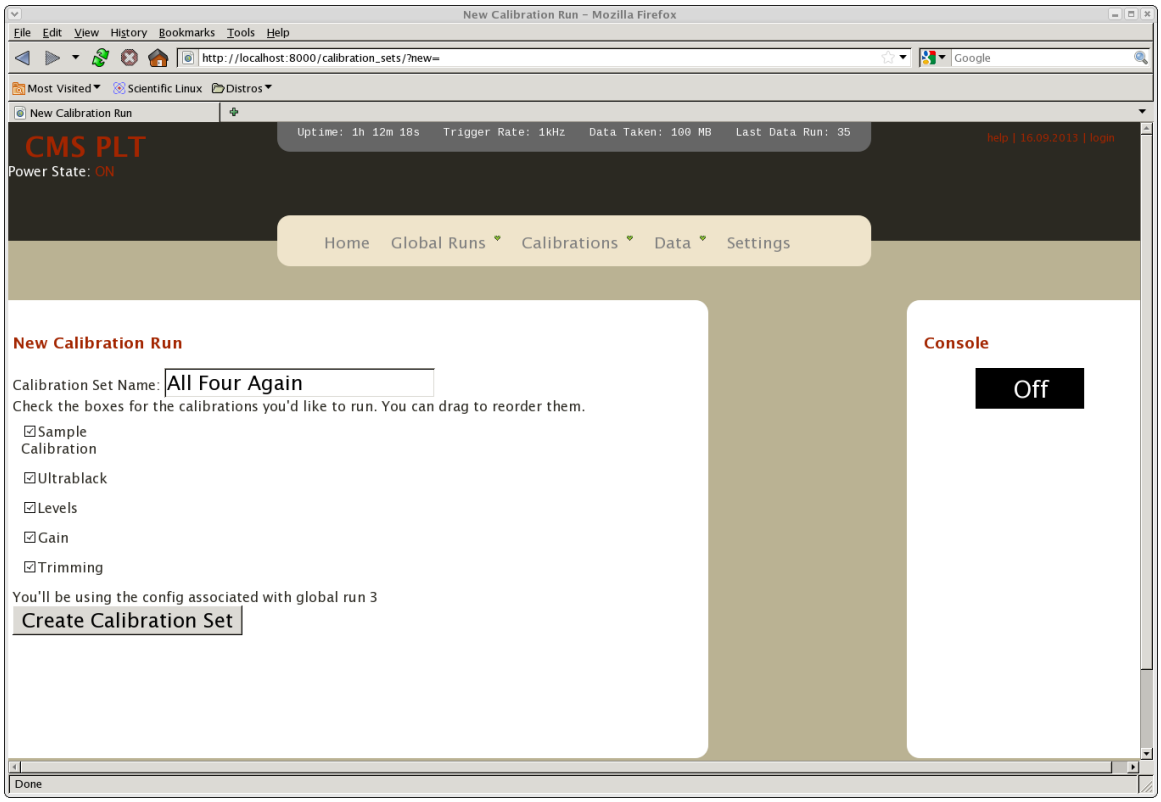


Figure 96: New calibration set: name edited

In this example, we have unselected the calibration called “Sample Calibration”. The order in which the individual calibrations are run may also be changed by dragging their names and checkboxes around. This page shows the “Gain” calibration in mid-drag to be positioned after the “Ultrablack” calibration and before the “Levels” calibration.

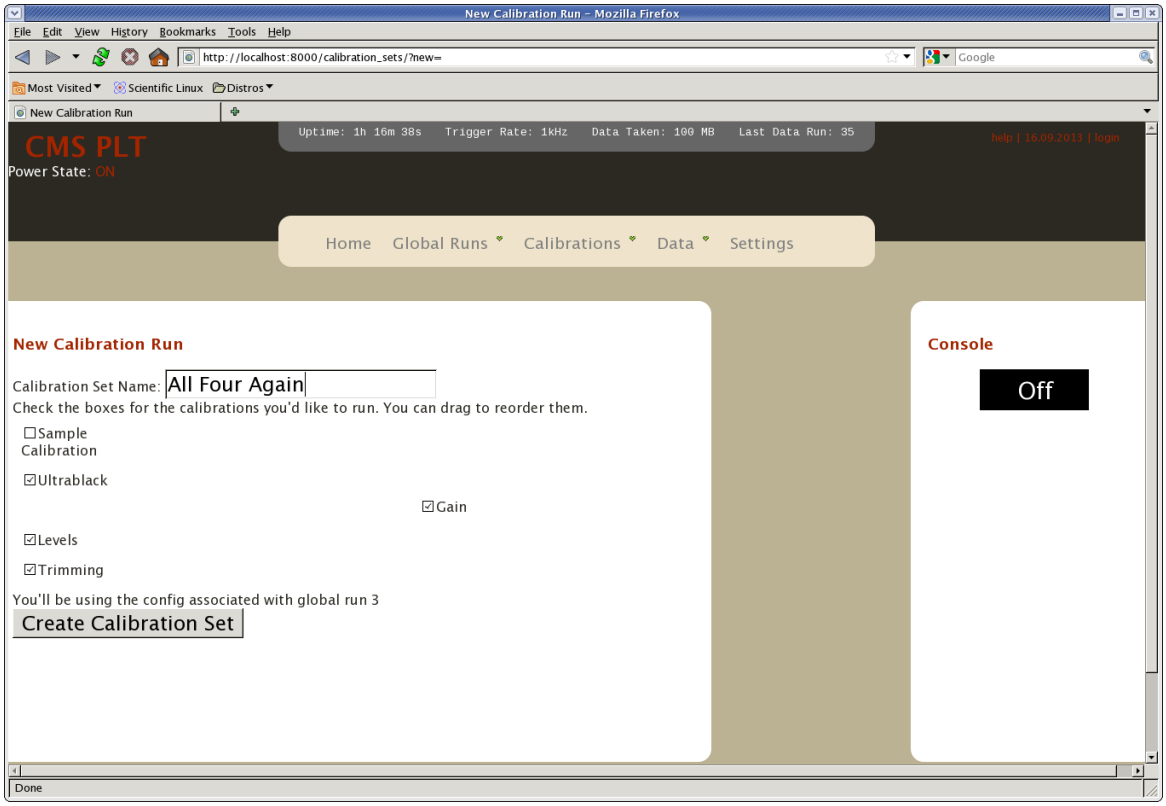


Figure 97: New calibration set: Sample calibration unselected, Gain calibration in mid-drag

After releasing the dragged “Gain” calibration, we’ll press the Create Calibration Set button.

Now the calibration set has been created. We may either select individual calibrations with their checkboxes (note the default is opposite that of the previous page; all are unselected by default) and press the Run Selected Calibrations button, OR we may simply press the Run All Calibrations button. The latter shall be demonstrated first.

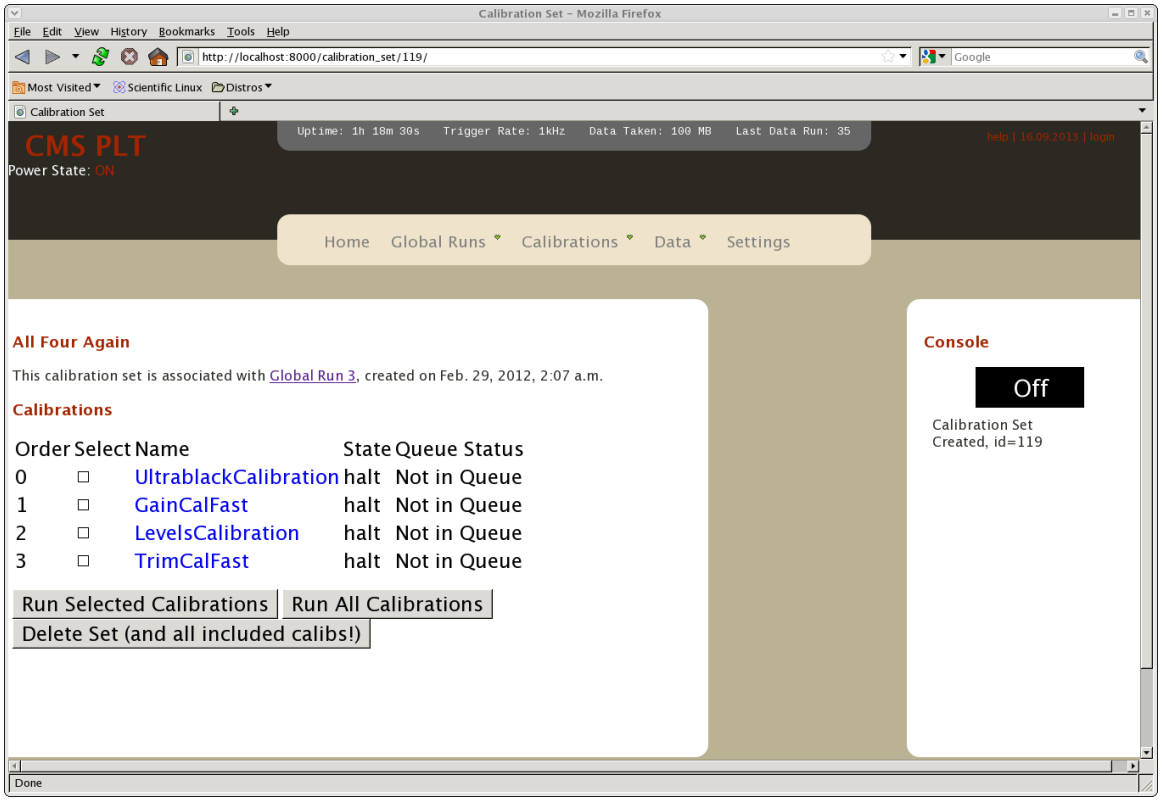


Figure 98: New calibration set: newly created

This is the result of the Run All Calibrations button. The run state and queue statuses are shown in progress.

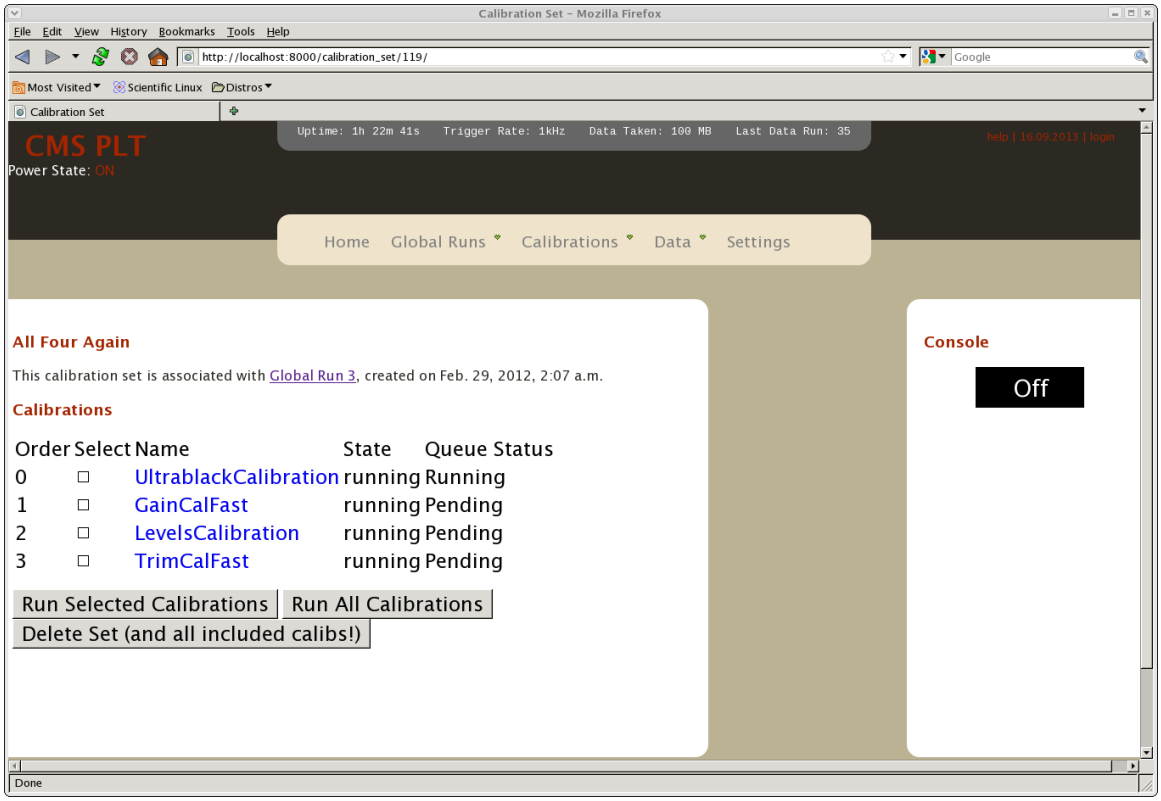


Figure 99: New calibration set: running all calibrations

When everything finishes, the states are all “done”.

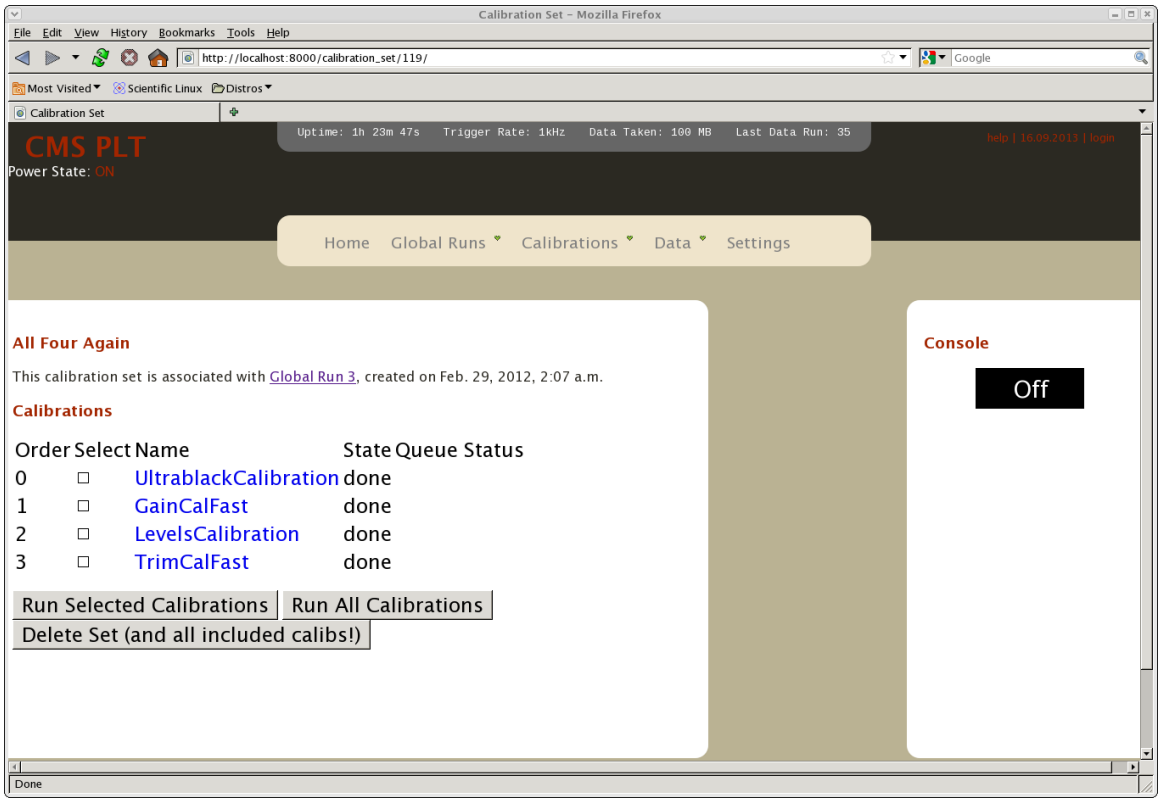


Figure 100: New calibration set: all calibrations done

Depending on the results of the calibrations, we may elect to re-run some of them by choosing them as shown below and then pressing the Run Selected Calibrations button. One may repeat this ad infinitum until satisfied.

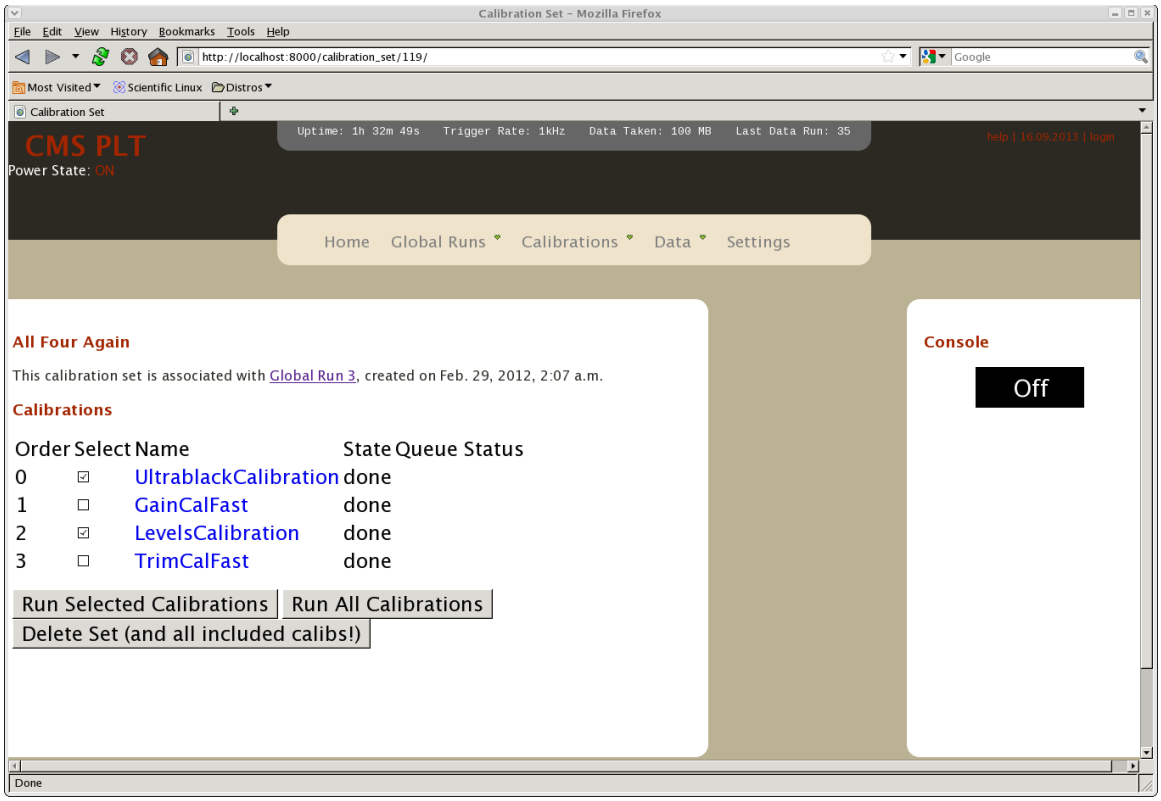


Figure 101: New calibration set: some calibrations selected for running

Examining the content of already-existing calibration sets is the other major functionality accessed with the navigation bar's Calibrations link. Going back to the list of existing calibration sets via the Calibrations link, we may choose to look at a different set, such as "All four calibrations", circled in red.

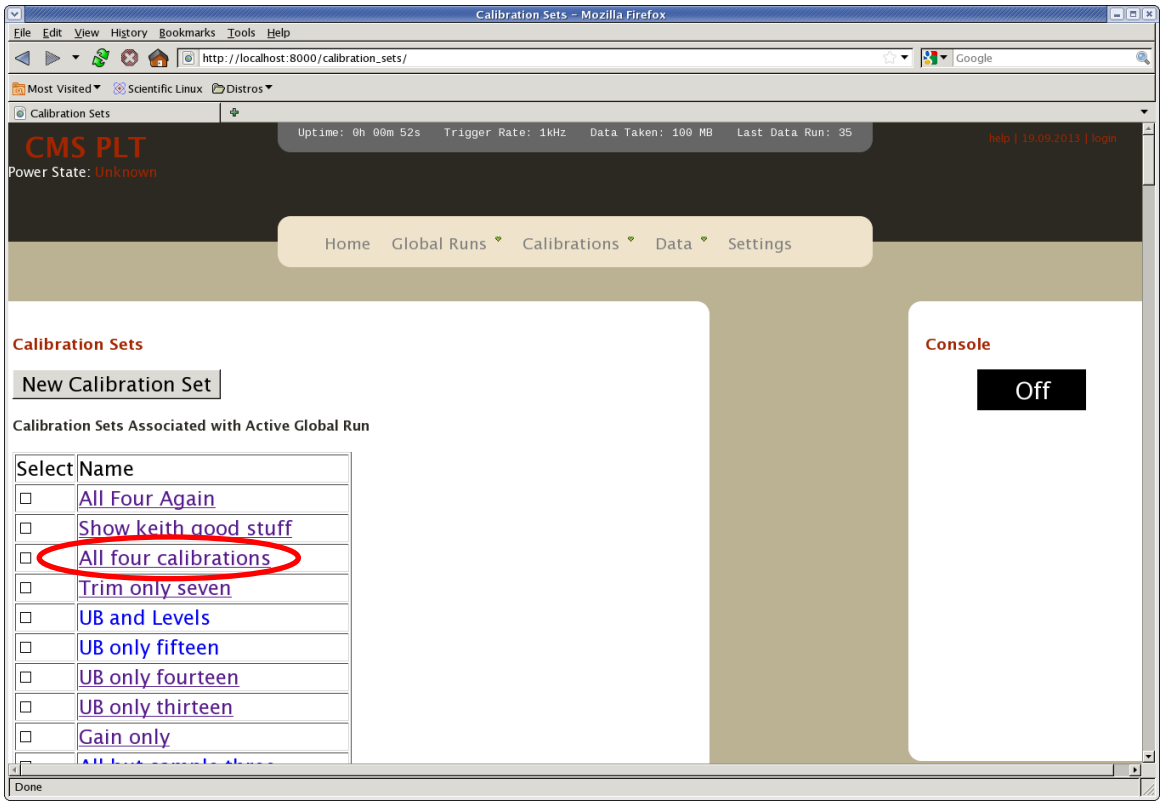


Figure 102: List of calibration sets: examine an existing set

Examining an individual calibration is just a matter of following the links again, this time choosing the “LevelsCalibration” link (circled in red).

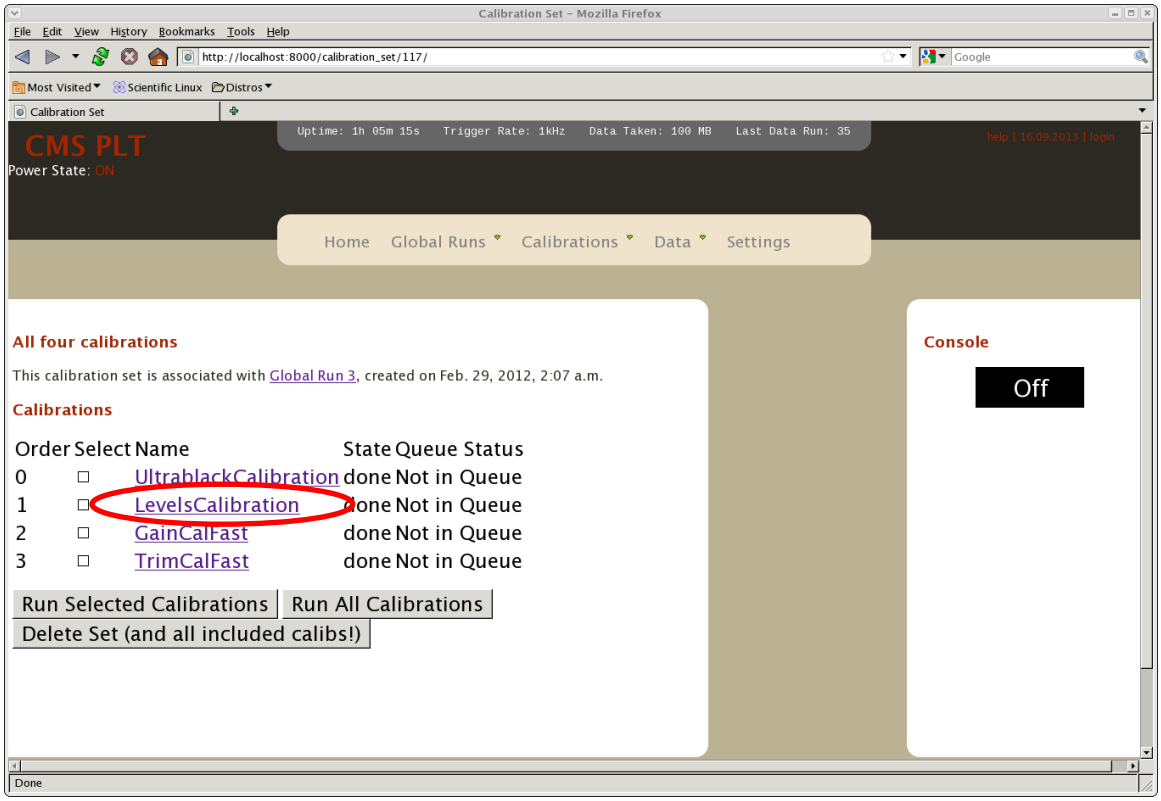


Figure 103: Existing calibration set: examine an individual calibration

Here one can see that the individual calibrations may be run either from the calibration set page level or from the individual calibration's page level.

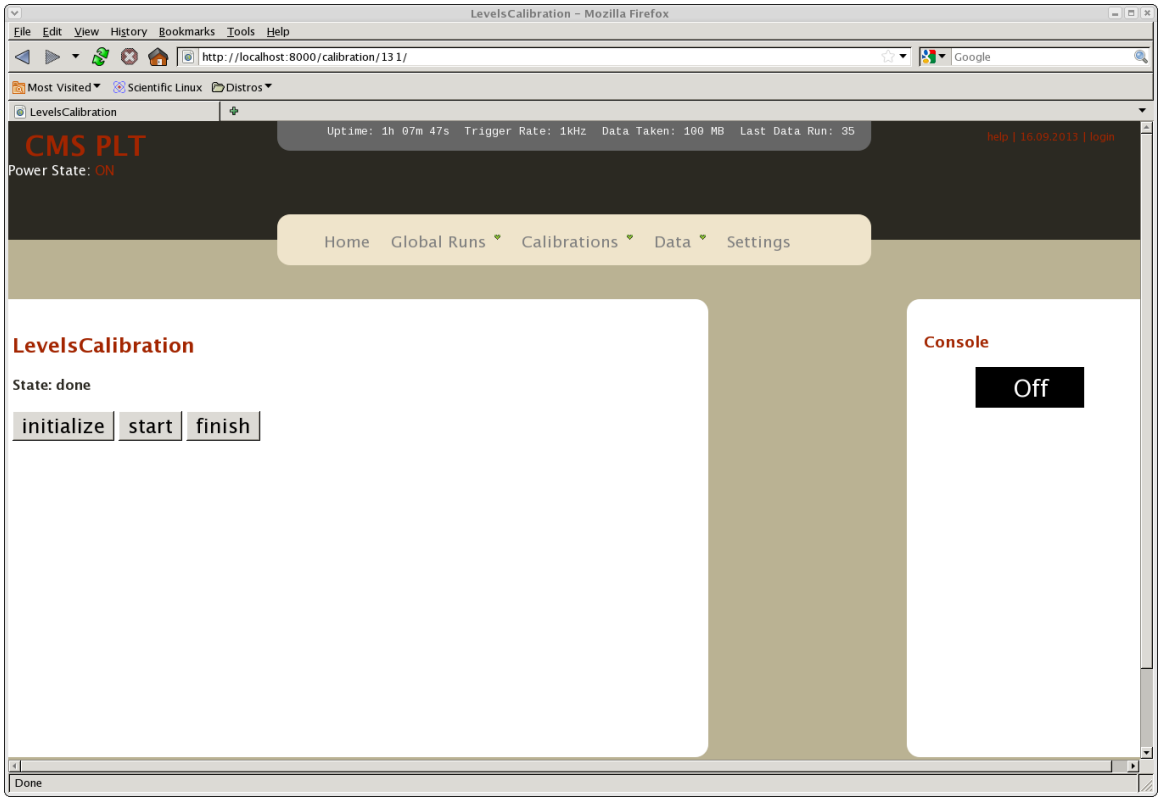


Figure 104: Individual calibration page

The Calibrations link in the navigation bar also has a widget for a drop-down menu. Both entries work and they merely take us to places that we have already visited in the preceding demonstrations.

The New Cal Run menu entry takes one to the same place as hitting the New Calibration Set button on the calibration set list page, and the View Calibration Sets menu entry takes one to the same place as the Calibrations link itself (the calibration set list page).

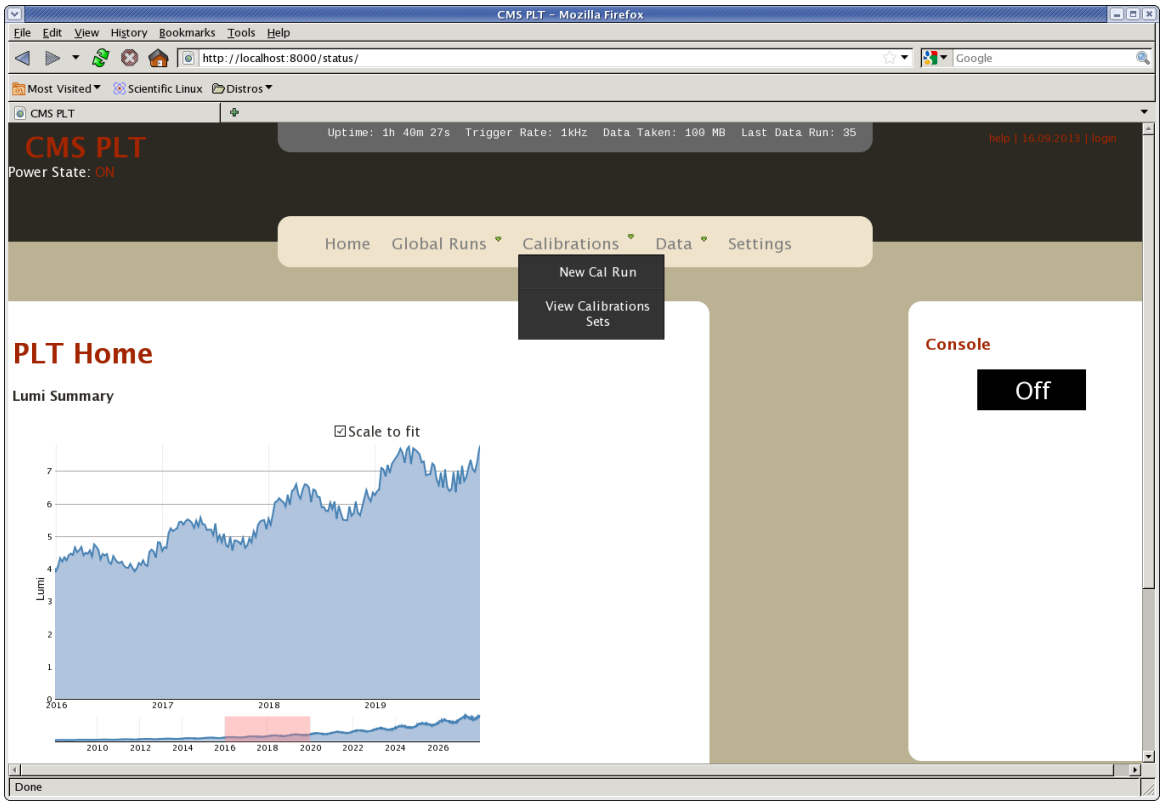


Figure 105: Submenu for the navigation bar's Calibrations link

Creating New Global Runs

One would want to create a new global run in order to start with a fresh set of configurations to be used with calibration runs and data runs. The fresh configurations would be initialized with power-up settings, which may have been updated, and modifications to the configurations are added on by calibration runs performed with the new global run.

From anywhere, click upon the Global Runs link in the navigation bar, enlarged and isolated below.

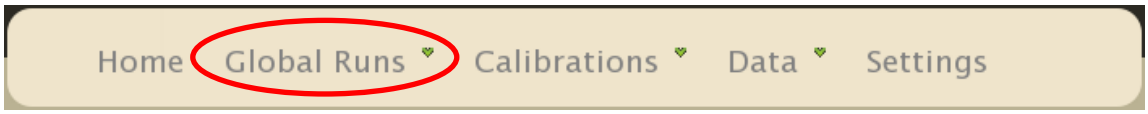


Figure 106: The navigation bar with the Global Runs link highlighted

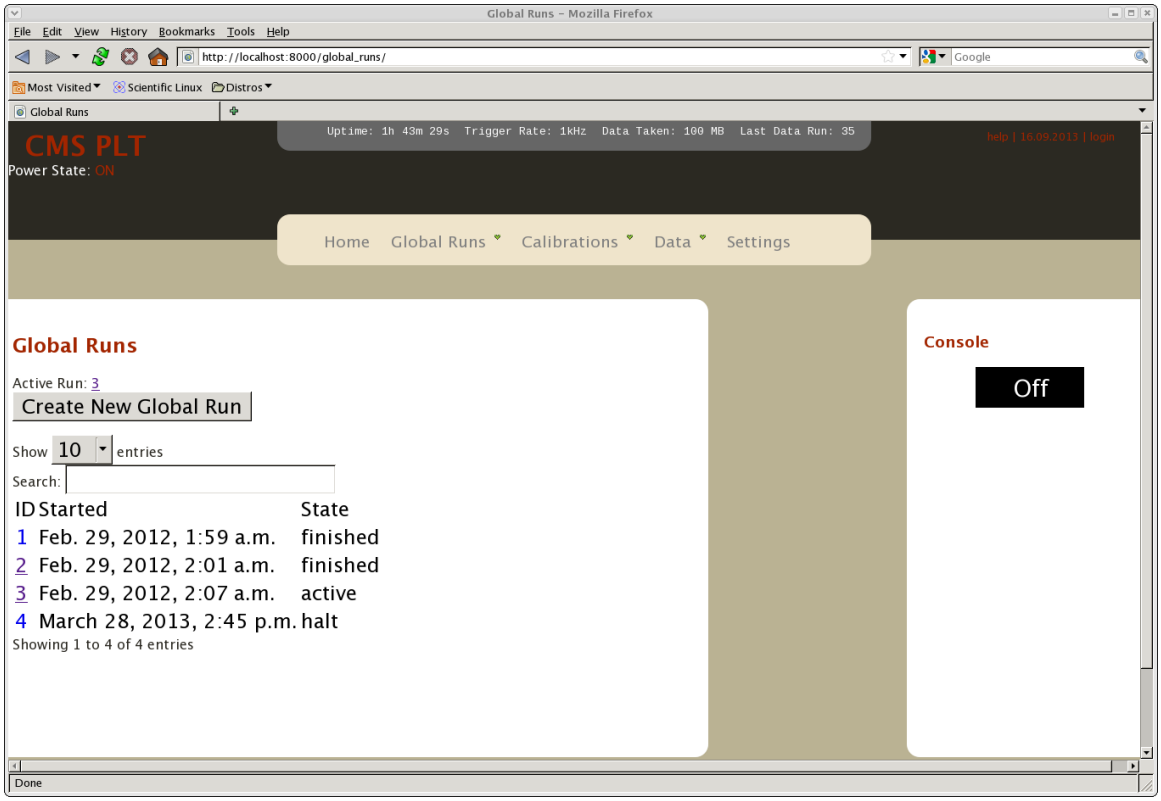


Figure 107: The list of global runs

After using the Create New Global Run button, we are put onto the control page for a new global run, numbered 5. Note that there is nothing listed under "**Configurations (in cascade Order)**". That merely shows that there are no configuration records for this global run yet.

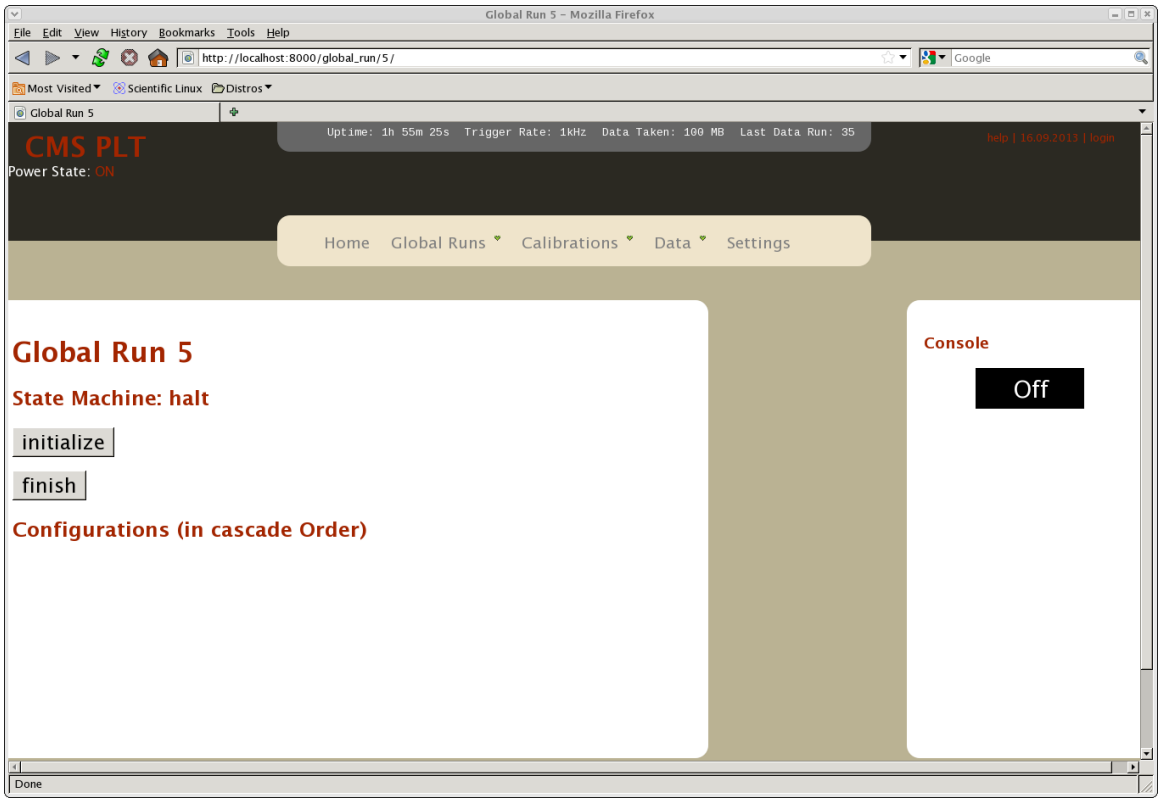


Figure 108: New global run page

When we press the initialize button, all other global runs are changed to the **finished** state, a POWERUP record is created to start the list of configuration records, and this global run is changed to the **active** state.

Here is the global run after initialization.

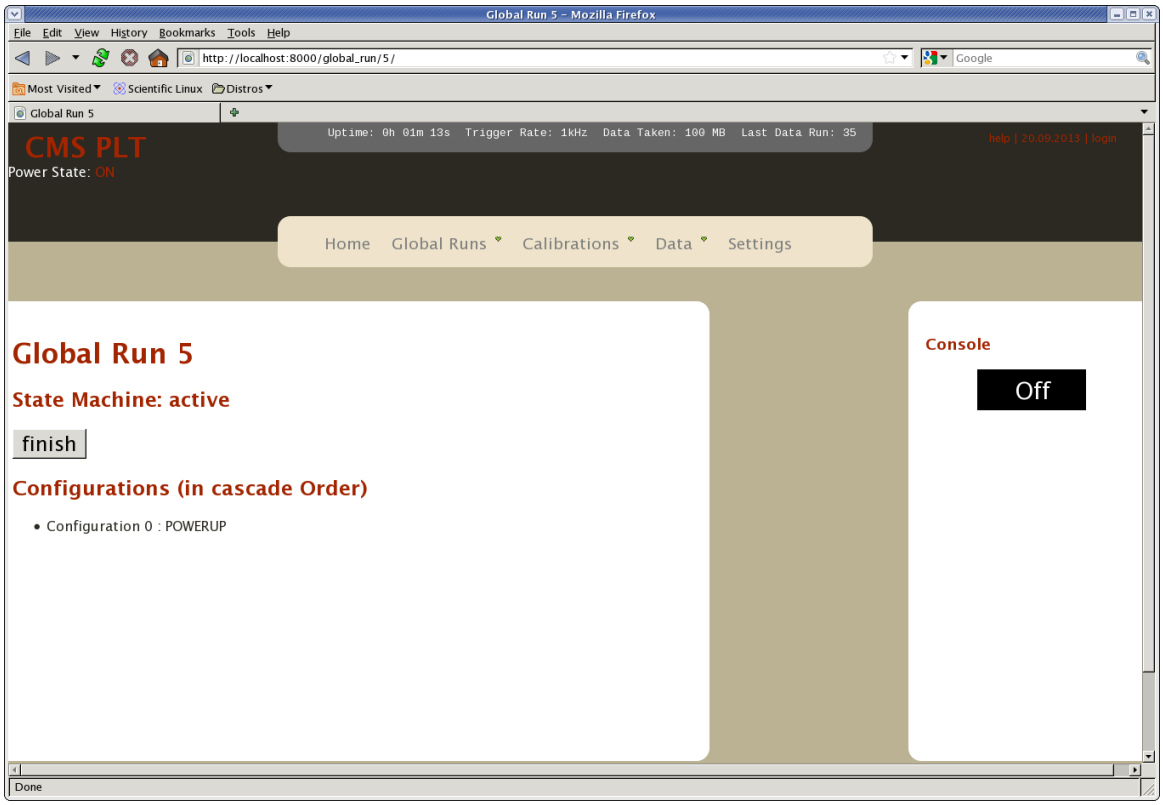


Figure 109: Initialized new global run

Performing calibrations will add more configuration records beyond the POWERUP record seen here. For an example of this, we can look at a previous global run record that has now been finished.

We can get there by using the navigation bar's Global Runs link to get back to the global run list page and then select one of the run numbers on the leftmost column.

Here we have chosen global run 3 (now finished) to show added configuration records due to performing calibrations, as well as demonstrating the examination of a global run in the list of global runs.

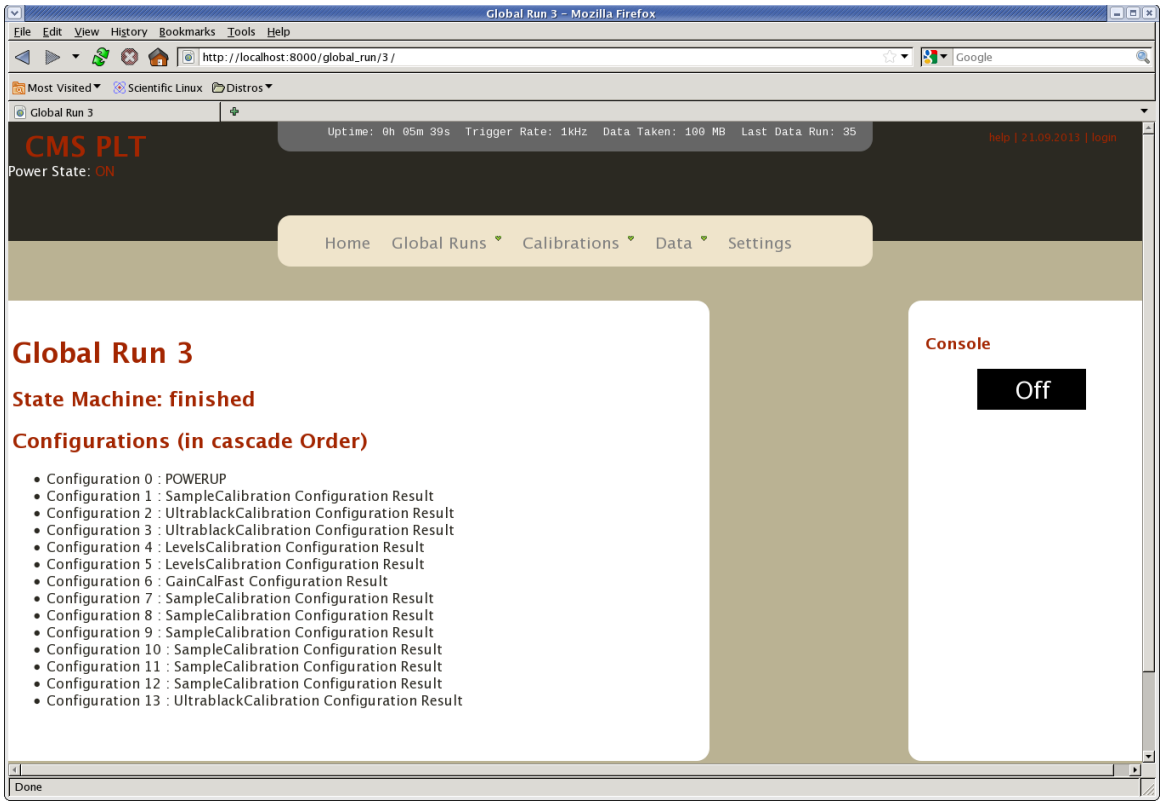


Figure 110: A previously finished global run, showing its configurations

Like the Calibrations link in the navigation bar, the Global Runs link in the navigation bar also has a drop-down menu widget at its upper right. It also provides two working entries that take us to places that we have already visited in the preceding demonstrations.

The Create New Global Run menu entry does the same thing as hitting the Create New Global Run button on the global run list page (creates a new global run record and puts one on its control page), and the View Existing Global Runs menu entry takes one to the same place as the Global Runs link itself (the global run list page).

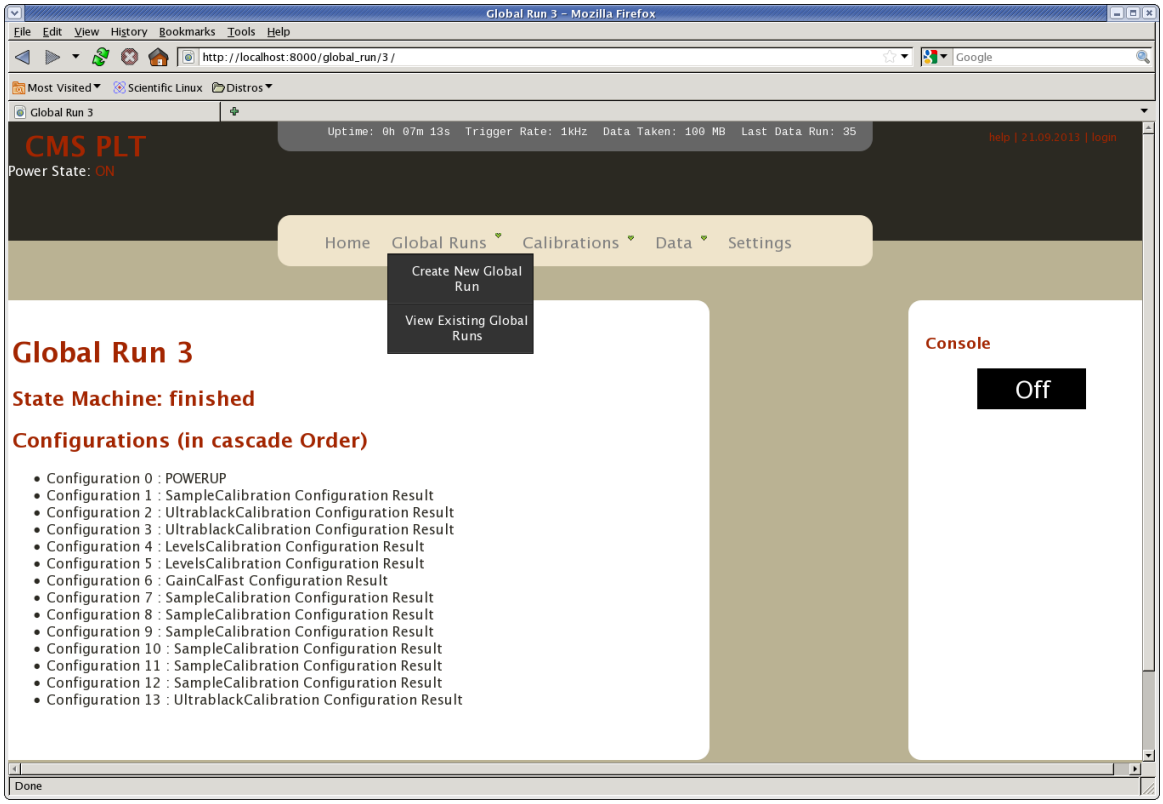


Figure 111: Submenu for the navigation bar's Global Runs link

Power and Trigger Utilities Page

There is a low level manual control page for performing power-up initialization of the hardware after power cycling the hardware, and for manual invocation of various triggering modes, and also for general script access. The means to get to this page is different from the means to get to all the other functionality described so far; it is not from the navigation bar, but by a separate link in the top left corner whose text indicates the power status, here circled in green.

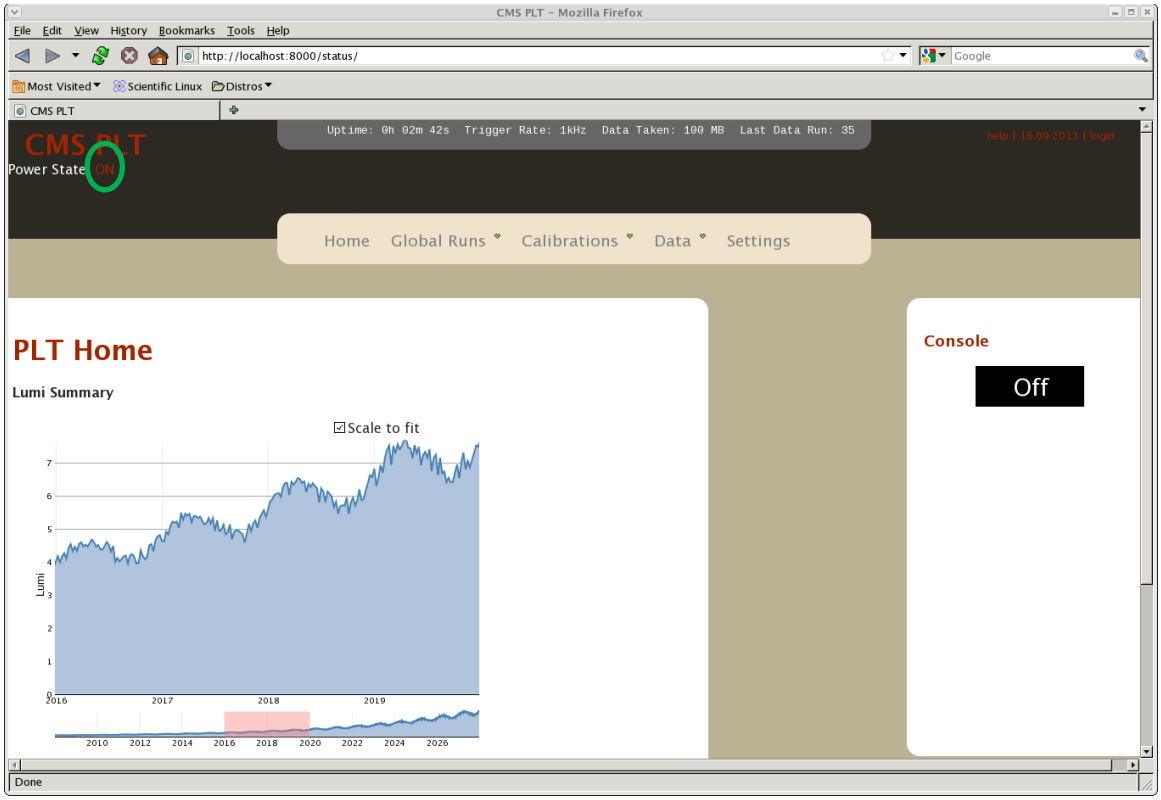


Figure 112: Home page with link to Power and Trigger Utilities page highlighted

When the server for the PLT Browser Interface is first started, it does not know the power state of the hardware and labels it as **Unknown**. That text will still be the link to reach the power and trigger utilities page.

Proceeding to the power and trigger utilities page…

Here everything is operated by a button, and generously labeled to be self explanatory.

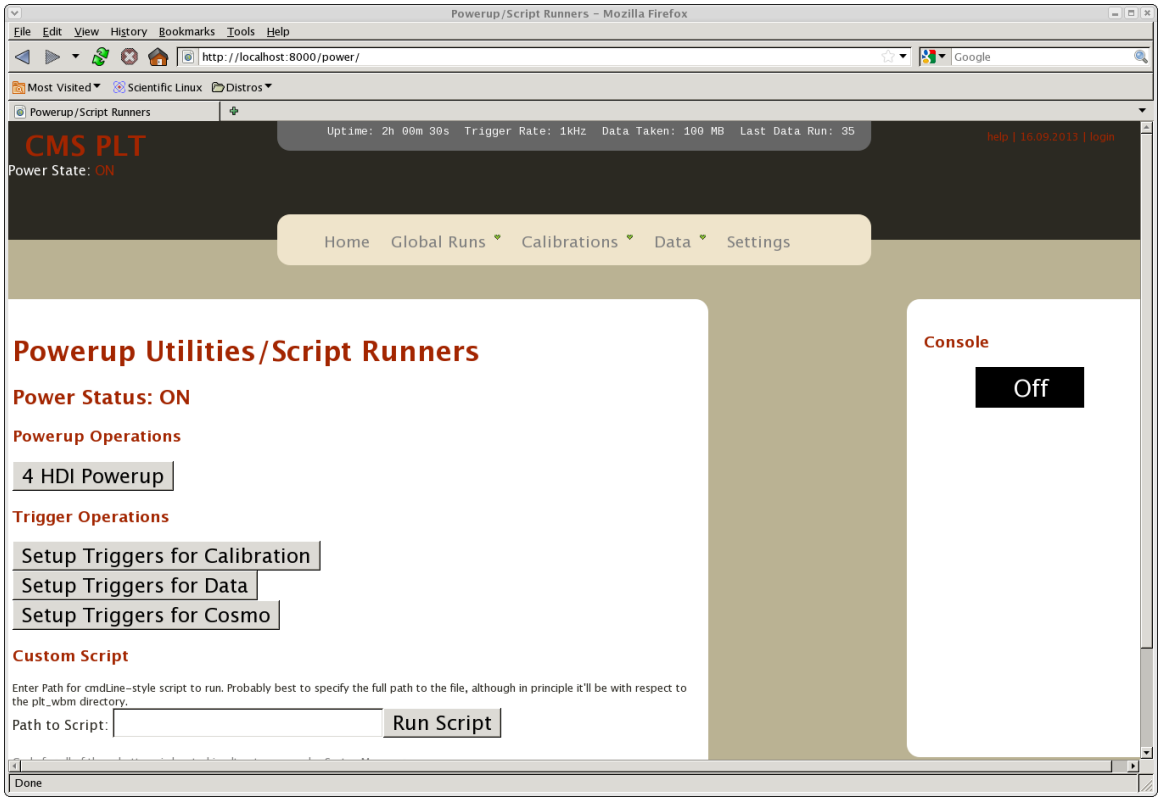


Figure 113: The Power and Trigger Utilities page

References

This chapter makes no external references.

CONCLUSION

There is a new detector in the CMS experiment at the LHC, dedicated to real-time luminosity monitoring and measurement. Its presence in CMS supplies luminosity information that is stable and achieves an elevated level of precision. It is being fully installed into CMS and will be taking data when the LHC resumes operation in 2015. It has a sophisticated software interface written for it that has been thoroughly developed, tested, and documented.

VITA

Mark M. Foerster was born June 21, 1965. He graduated from the University of Delaware in 1990 with a B.A. in computer science and worked in the software industry for fifteen years. He returned to school in 2005 and graduated from the Pennsylvania State University in 2010 with a B.S. in Physics. He pursued graduate studies at the University of Tennessee at Knoxville and graduated with an M.S. in Physics in 2015.