An Introduction to ATLAS Pixel Detector DAQ and Calibration Software Based on a Year's Work at CERN for the Upgrade from 8 to 13 TeV

A report
Submitted in partial fulfillment of the
Requirements for the degree of
Masters in Science

Updated 05/20/2016

Nick Dreyer
Department of Physics
University of Washington, Seattle, WA.

Committee:
Shih-Chieh Hsu
Jeffrey Wilkes

CERN-THESIS-2016-305
15/12/2016

An Introduction to ATLAS Pixel Detector DAQ and Calibration Software Based on a Year's Work at CERN for the Upgrade from 8 to 13 TeV

(Nicholas Dreyer 12/15/2016)

Abstract

An overview is presented of the ATLAS pixel detector Data Acquisition (DAQ) system obtained by the author during a year-long opportunity to work on calibration software for the 2015-16 Layer-2 upgrade. It is hoped the document will function more generally as an easy entry point for future work on ATLAS pixel detector calibration systems. To begin with, the overall place of ATLAS pixel DAQ within the CERN Large Hadron Collider (LHC), the purpose of the Layer-2 upgrade and the fundamentals of pixel calibration are outlined. This is followed by a brief look at the high level structure and key features of the calibration software. The paper concludes by discussing some difficulties encountered in the upgrade project and how these led to unforeseen alternative enhancements, such as development of calibration "simulation" software allowing the soundness of the ongoing upgrade work to be verified while not all of the actual readout hardware was available for the most comprehensive testing.

Table of Contents

1 Introduction

This work organizes and consolidates general information and documentation gathered and produced throughout an 11 month contract the author was fortunate to have had at CERN adapting existing pixel calibration software for the ATLAS pixel detector layer 2 upgrade in the winter of 2015/16. The upgrade is intended to double the DAQ bandwidth of the layer. This upgrade is part of a larger upgrade that also includes doubling layer 1 DAQ bandwidth. Since layer 2 was the first of the two expected to run out of bandwidth, the project was split over two years, with later layer 1 upgrade to be completed in the first half of 2017.

Pixel calibration software is tightly linked to all the DAQ hardware and highly integrated into data-taking software. As problems remained in the latter two up until the last minute before the detector had to resume full data taking operation in 2016 and shortly before the author had to leave CERN, the ability for testing the upgraded layer 2 calibration on the detector itself was never possible, and so the detector has been run throughout 2016 using its last layer 2 pixel calibration settings from the year before. The upgrade testing was completed to the extent possible in the CERN “SR1” test facility, but as only a very limited number of modules were accessible there, these tests could not come close to reproducing the much more intense conditions in the full detector. It appears that the setback of running the detector with a less than optimally tuned layer 2 was not the “end of the world”, since it was anyway a transitional period before the planned completion of the final layer 1 upgrade at the full 13 TeV energy.

This document is limited to the areas of software development covered by the author's work on upgrading calibration for layer 2. For that reason it is by no means an exhaustive set of documentation. However, it does cover some important features whose understanding is key for any efficient work on the pixel DAQ software, and it is hoped that some of the, at times highly focused descriptions are illustrative enough that a more general understanding of the entire DAQ software easily emerges from those examples.

The author believes that much time of his would have been saved had a document of a similar nature been available during his work on the project.

That being said, one can argue that the hard way is often the more thorough way of learning, so with that in mind - together with obstacles, such as unavailable hardware on which to test - almost surely a variety of new features to test intermediate portions of the DAQ (needed because others were missing) would never have been developed.

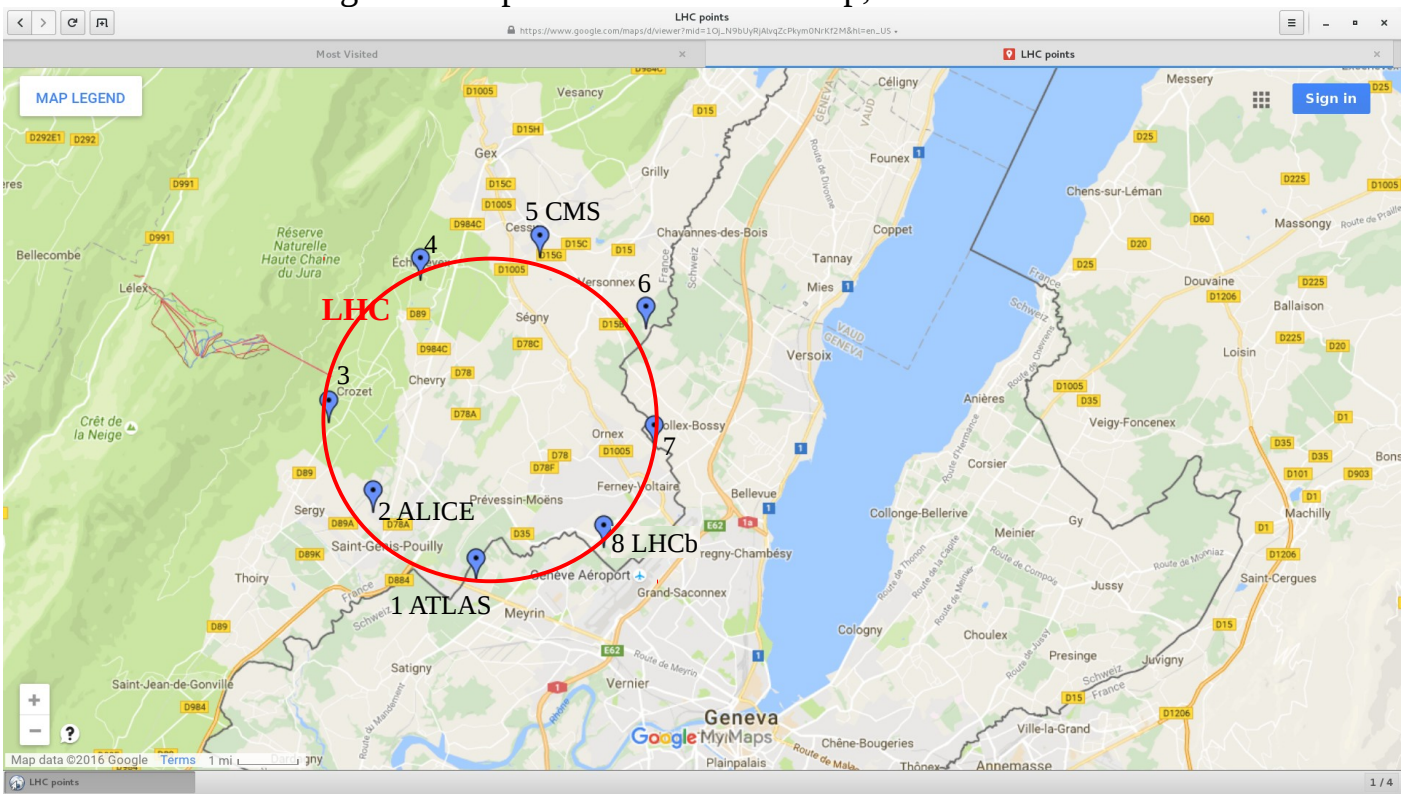
A Note on updates since the original thesis submittal:

The document in fact proved to be an invaluable reference during the further work for the layer 1 upgrade in 2017, with only minor corrections added to it as a result of that work.

2 Zooming in

2.1 Context of the ATLAS pixel detector within the LHC

The best way to visualize the position of the ATLAS experiment in the overall context of CERN is the following map, which also marks the location of four other major experiments located at four of the eight access points of the 100m deep, 27km circumference LHC tunnel:



In addition to passing through the unfathomably large LHC accelerator itself, the following (not-to-scale) diagram shows how the protons in a typical ATLAS particle collision will have first traversed the four preliminary accelerators, LINAC 2, PSB, PS and SPS:

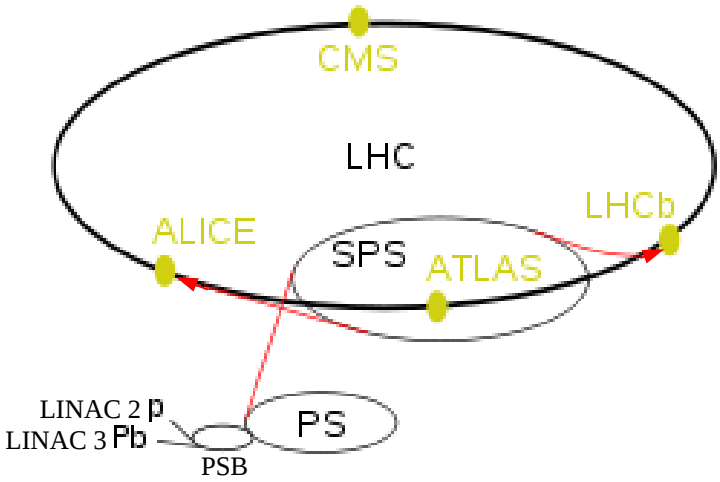


Figure 1 adapted from [1]

Next we find the 90-million-pixel, but comparatively minuscule 1.4m-long and 0.43m-across cylindrical pixel detector (circled in red) within the entire ATLAS detector:

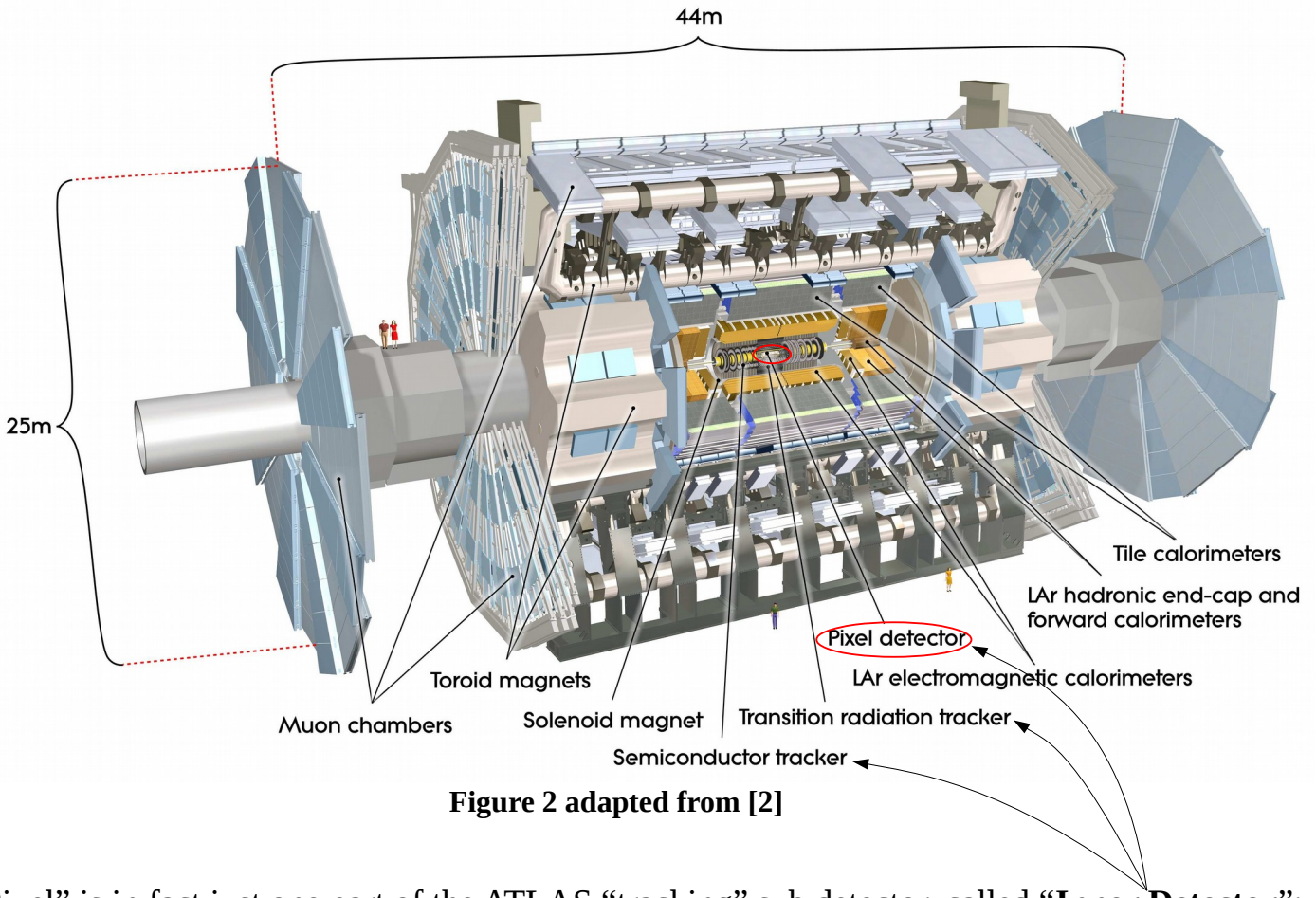


Figure 2 adapted from [2]

"Pixel" is in fact just one part of the ATLAS "tracking" sub detector, called "**Inner Detector**":

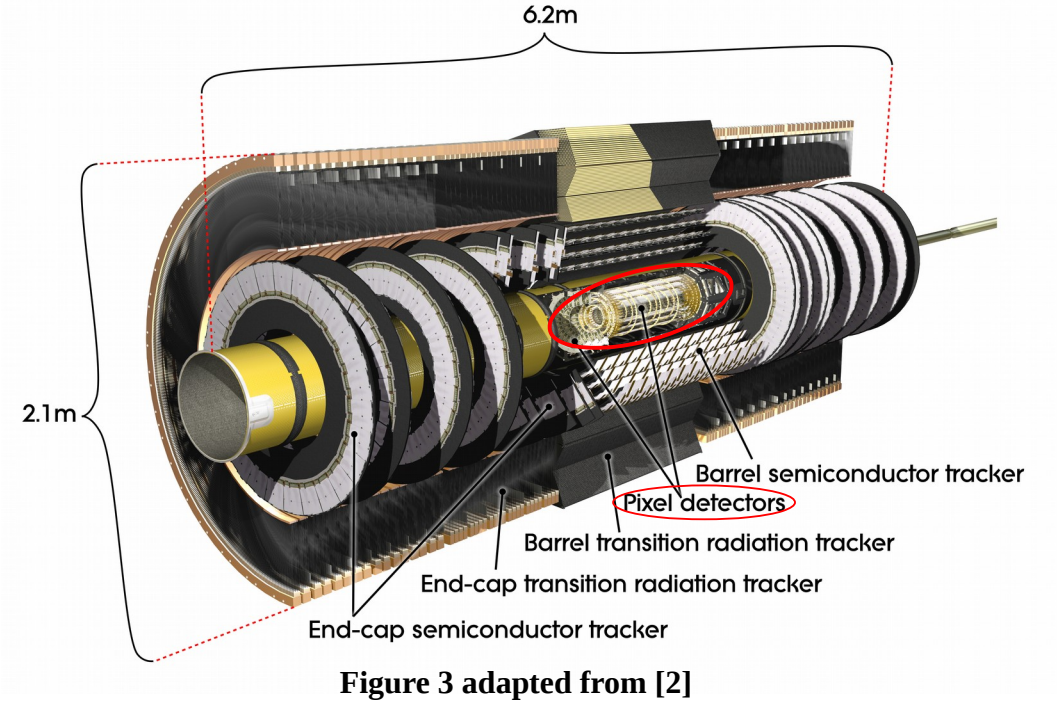


Figure 3 adapted from [2]

Zeroing in still further within the pixel detector itself we find the over 90 million pixels distributed over four concentric “barrel” layers and 6 “end-cap” disks (the latter not pictured):

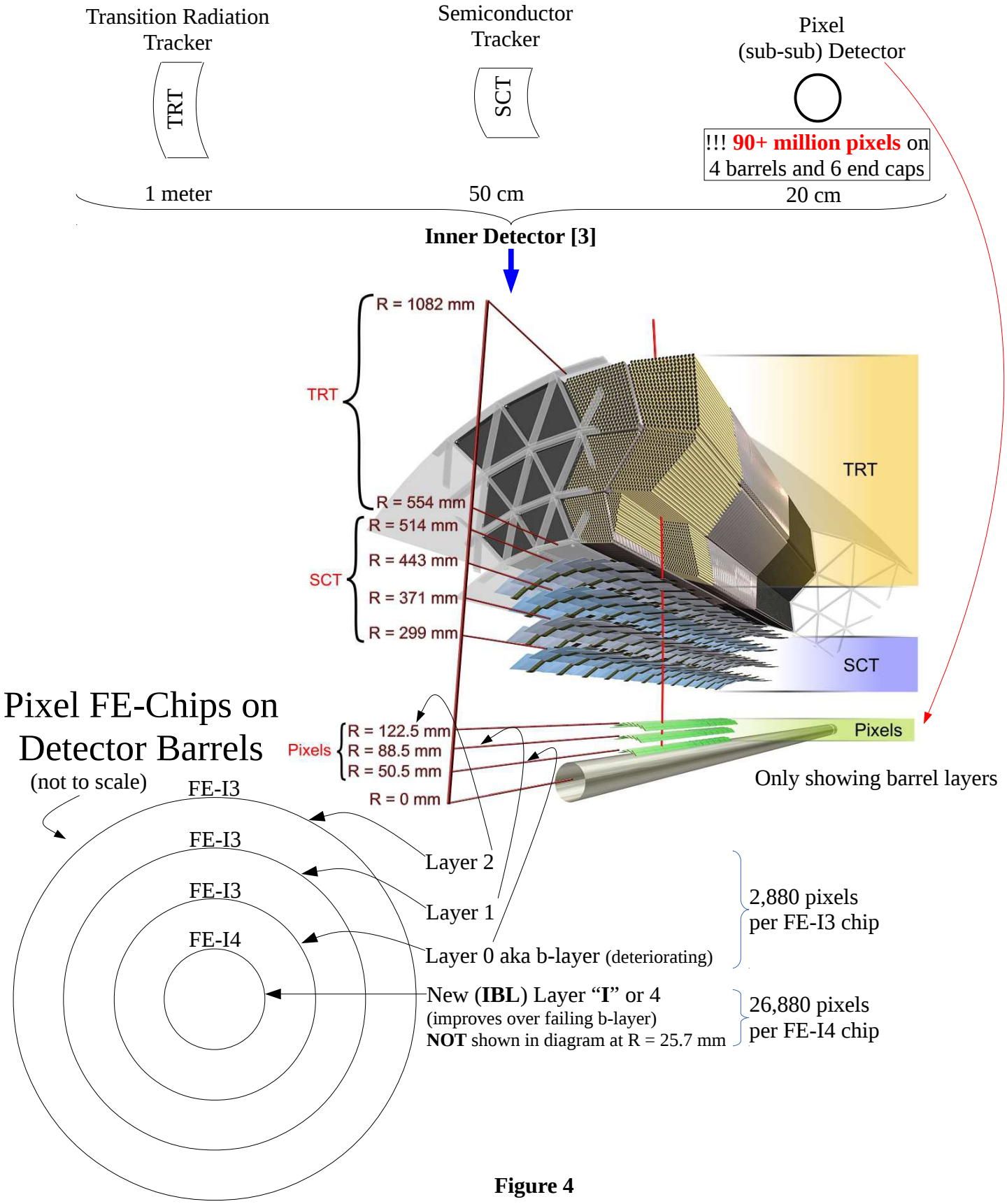
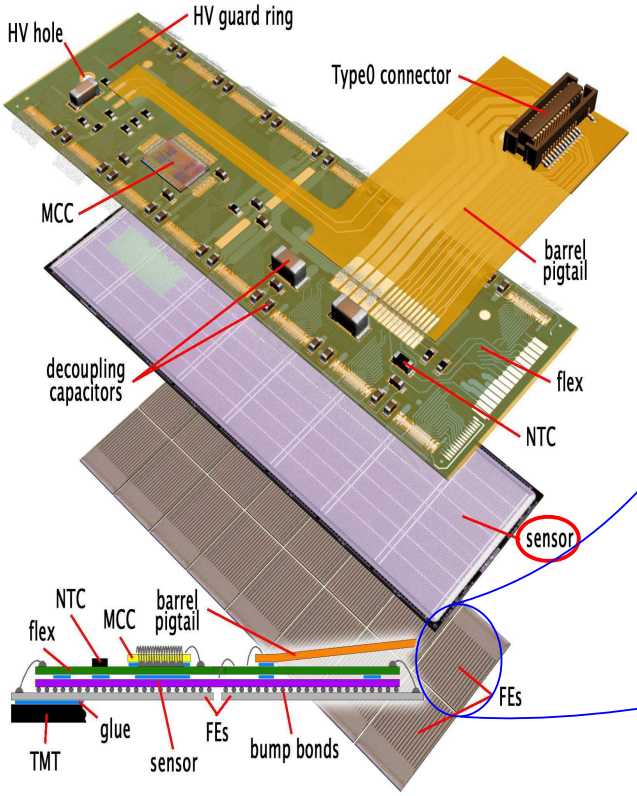


Figure 4

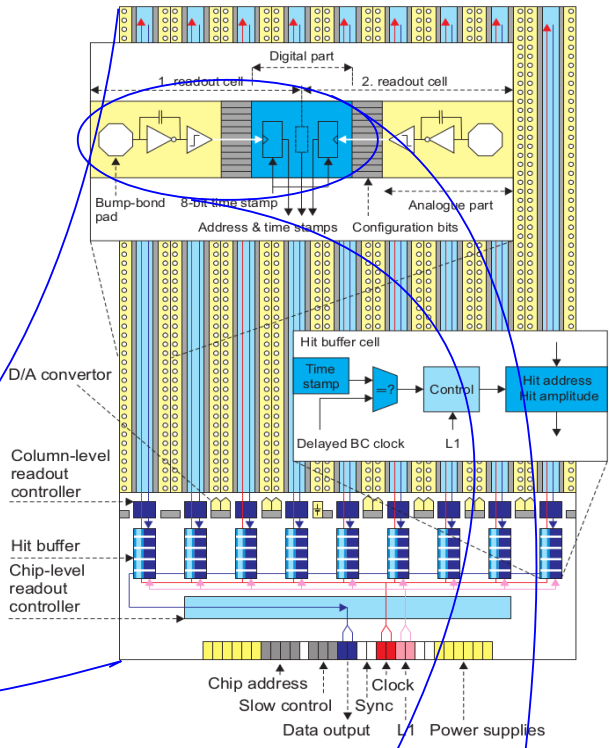
2.2 Module, Front End (FE) Chip and Pixel Hardware

A view at the ultimate resolution: sensors connect via bump bonds to FEs, aggregated in modules.

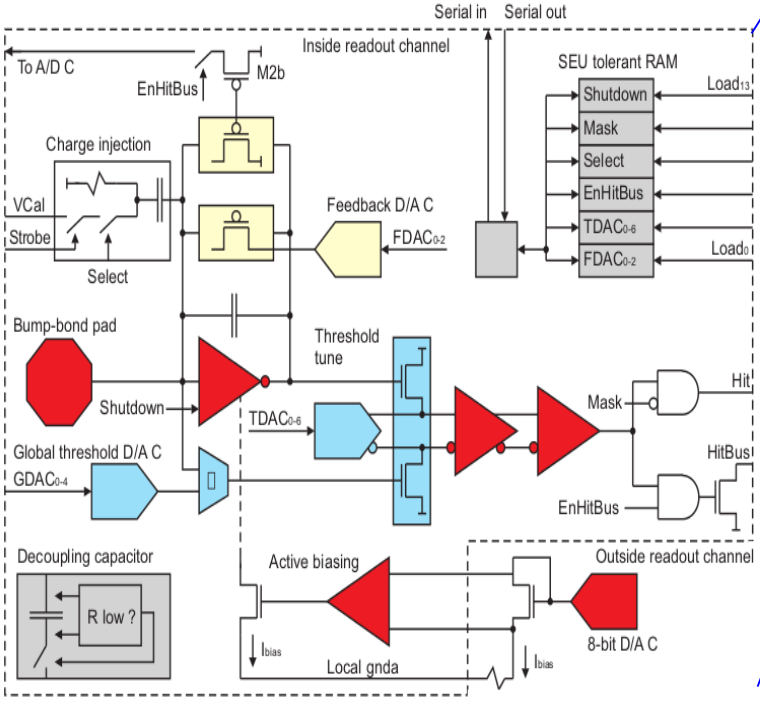
Module (adapted from [3])



FE Chip (adapted from [3])



Schematic plan of the front-end chip (FE-I3) with main functional elements. Not to scale.



Pixel cell block diagram. (adapted from [4])

Figure 5

Finally, it is useful to present some comparisons of the newer IBL FE-I4 vs. FE-3 chip:

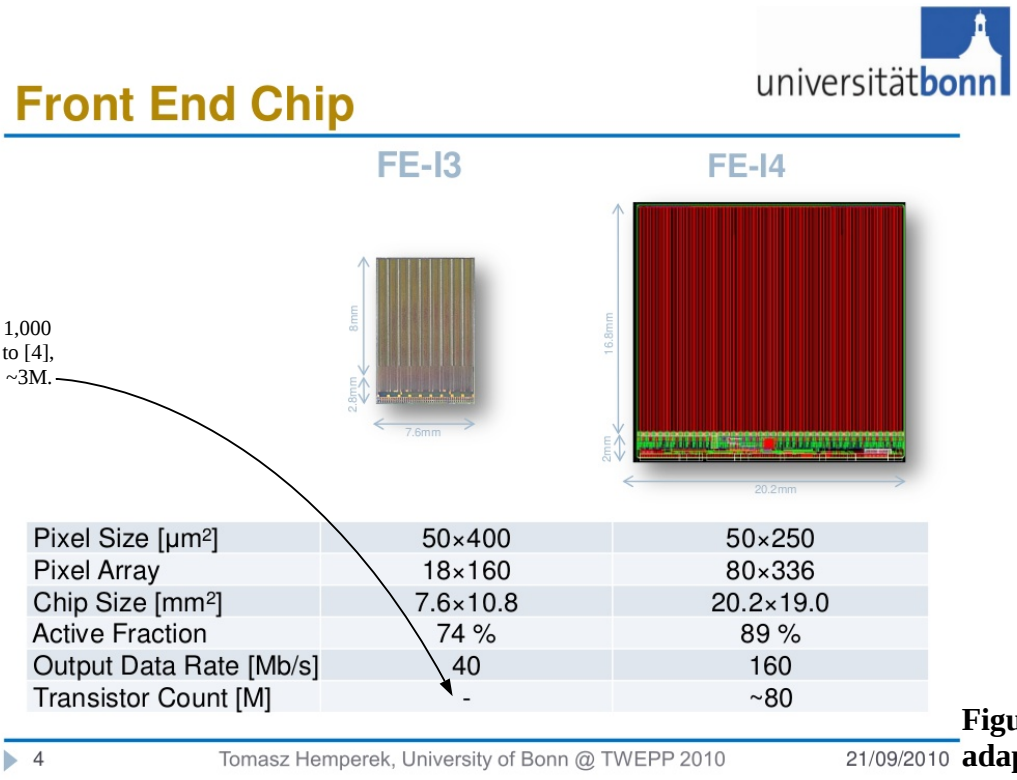


Figure 6 adapted from [5]

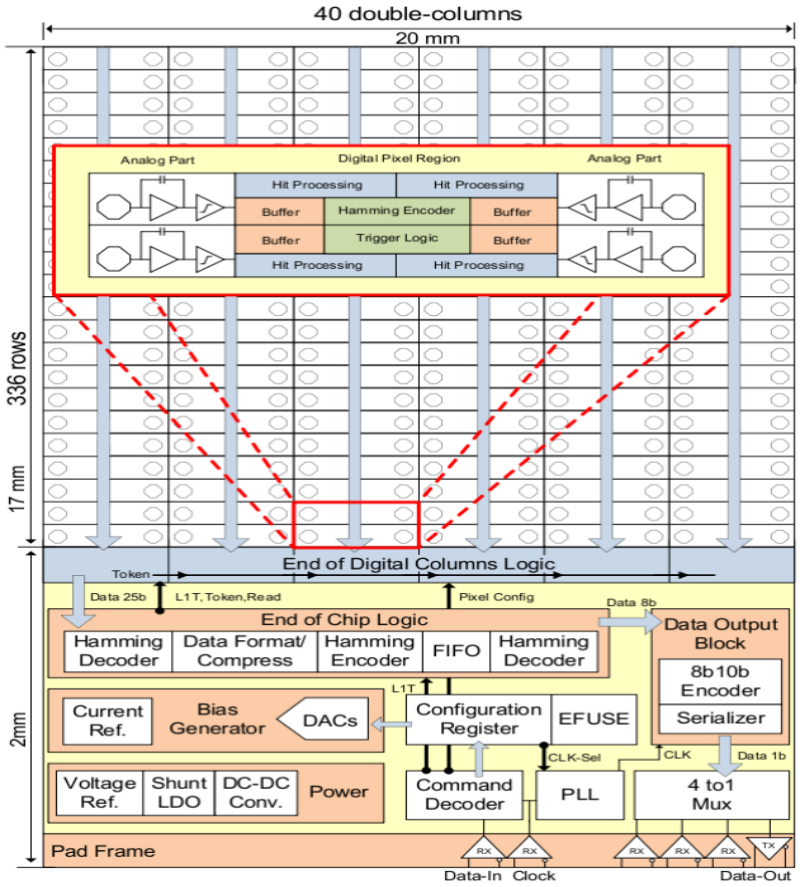


Figure 7 from [6]

FE-I4 chip diagram, not to scale. The coordinate origin is at the bottom left corner,

3 Hardware of pixel DAQ

The following diagram captures the essence of the DAQ data flow:

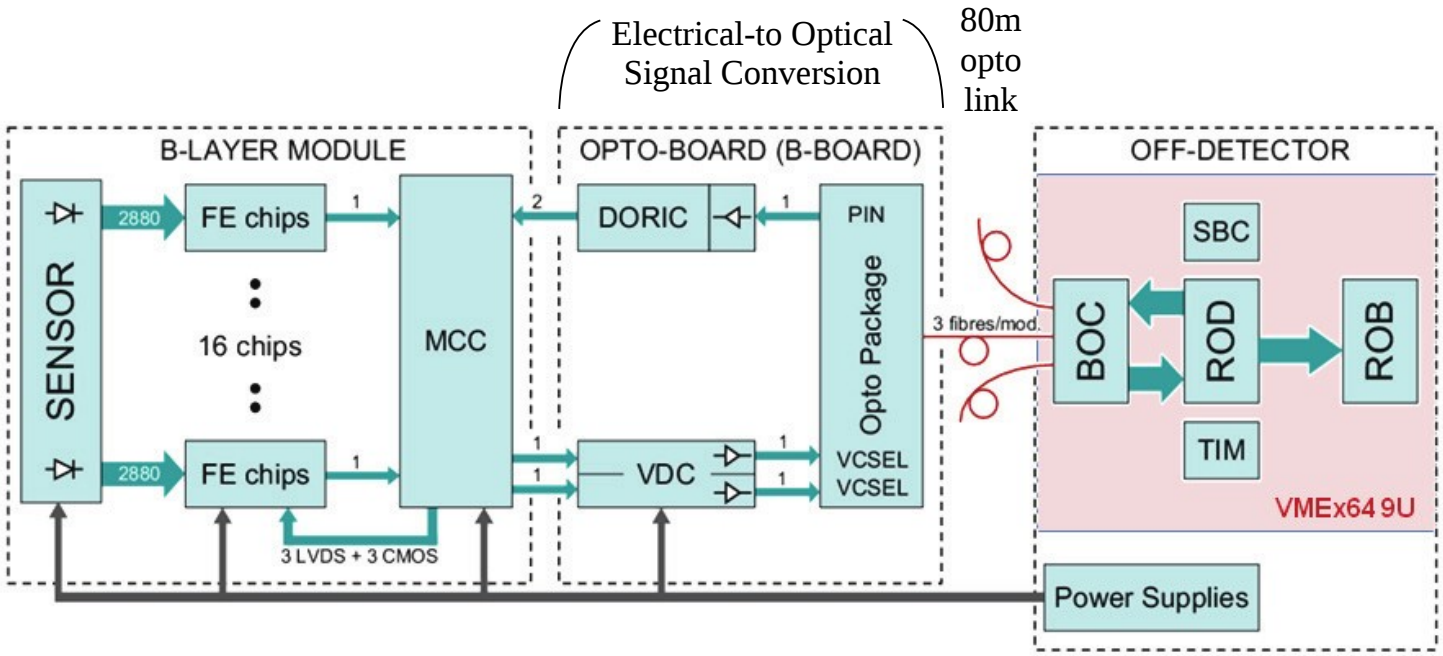


Figure 8: Schematic of Readout Hardware, adapted from [7]

While the components shown in Figure 8 are for the original FE-I3 readout system, except for some details of opto-board and IBL modules, the new system for Layers 2, 1 and IBL is not fundamentally different, except that there is no MCC for the IBL FE-I4 FE chips.

Here is another view of these hardware components highlighting the parts replaced for the L1/L2 upgrade and their relation to the revamped calibration software via the ROD:

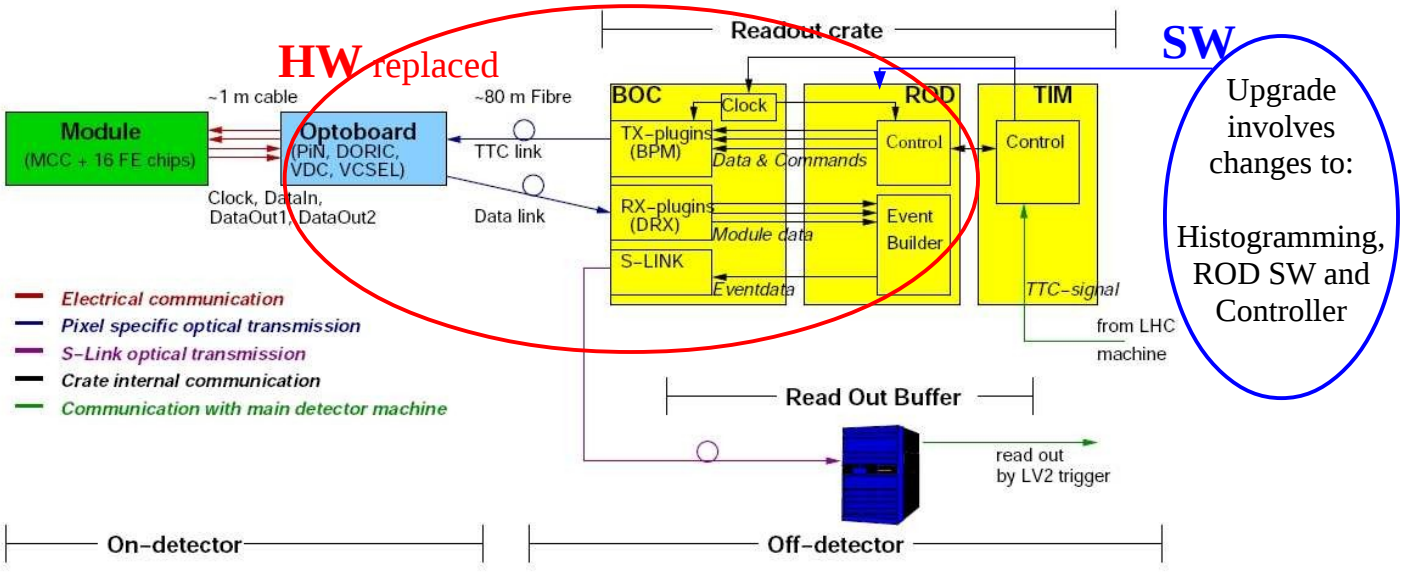


Figure 9
adapted from [8]

4 Pixel DAQ bandwidth upgrade - why?

Having started their journey in LINAC 2, the charged particles, into which the protons are transformed by their high energy collision at the center of the pixel detector, trigger a current in its semiconductor sensors that is propagated as signals through all the intricate electronics in the pixel chips and subsequent DAQ systems. This is the raw data for particle trajectory determination. The upgrade from 8 TeV to 13 TeV collision energy requires DAQ bandwidth expansion whose limitations in the original DAQ are expressed in the following table:

- The extrapolated pixel occupancies can be used to determine the **occupancy per CP and BC**:

Observed **pre**-upgrade

Occupancy projections per column pair and BC					
	mu	B-Layer	Layer 1	Layer 2	Disks
50 ns	37	0.17	0.07	0.04	0.06
25 ns; 13 TeV	25	0.15	0.06	0.04	0.05
	51	0.27	0.11	0.07	0.09
	76	0.39	0.16	0.10	0.13

- From this the estimated MCC→ROD **link occupancy** at 75 and 100 KHz LVL1 rate can be calculated:

Time between BC's (**was** 50 ns **pre**-upgrade)

Link occupancy at 75 kHz L1 Trigger					
	μ	B-Layer	Layer 1	Layer 2	Disks
50 ns	37	39%	34%	52%	30%
25 ns; 13 TeV	25	35%	31%	48%	27%
	51	53%	59%	66%	39%
	76	71%	73%	111%	64%

Link occupancy at 100 kHz L1 Trigger					
	μ	B-Layer	Layer 1	Layer 2	Disks
50 ns	37	51%	45%	69%	40%
25 ns; 13 TeV	25	47%	42%	65%	37%
	51	71%	67%	88%	52%
	76	95%	97%	148%	75%

Figure 10 adapted from [9]

Key to jargon:

CP = Column Pair (2 of 18 FE columns = 320 pixels = 1/9th of FE-I3 chip)
BC = "Bunch Crossing" = Point where/when packet of particles in beam intersect
mu (μ) = Collisions per BC
MCC/ROD = On/Off-detector electronics (Module Control Chip / Read Out Driver)
LV1 rate = Rate of "event" detection, i.e. ("Level 1") trigger rate of read-out system

Note that the upgrade also reduces BC from 50 to 25. It is clear that Layer 2 is the first to reach saturation, with Layer 0 and the disks just hanging on and not needing to be part of the upgrade. The necessary on-detector hardware for the upgrade (new laser fibers and associated electro-optical conversion) was already installed during the 2014 IBL insertion phase. Off-detector hardware and related software for Layer 2 were installed during the winter 2015/16 shutdown, and a virtually identical revamp for Layer 1 is planned for winter 2016/17.

5 History of all pixel DAQ upgrades

With the doubling of luminosity, the DAQ data flow rate doubles, and as just illustrated, this overwhelms the original system. Since Layers 1 and 2 were originally only wired for half of their built capacity, the necessary detector intervention for adding the additional wires was used as an opportunity to replace the on-detector original, less-than-optimal electro-optical conversion system. Overall technical parameters concerning the entire history of pixel detector DAQ upgrades are summarized in the following table:

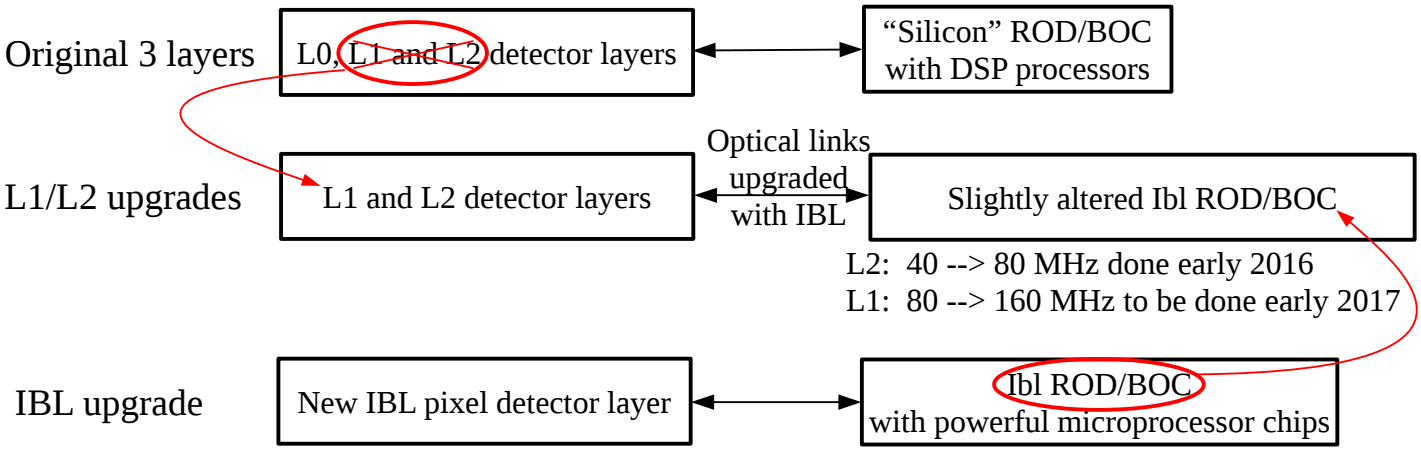
Detector Layer	Opto-links (incl 80m fiber)	Pixels/link ="module"	Original MB/s	Upgraded MB/s	Off-detector Upgrade
IBL	Installed in 2014	53,760	160	160	New "IBL" ROD/BOC
B (aka "L0")	Upgraded in 2014 at time of IBL upgrade	46,080	160	160	--
L1		46,080	80	160	2016/7 major upgrades: Adapt to IBL tech.
L2		46,080	40	80	

Original ATLAS pixel detector
2014 IBL upgrade
2016 (almost done) upgrade
2017 planned

6 Hardware of L1/L2 upgrade

6.1 What changes?

The most complex changes of the upgrade are in the off-detector DAQ electronics, where ReadOut Driver (ROD) and BackOfCrate (BOC) card-pairs, developed for the IBL, replace the original "pixel" ROD-BOC pairs, the latter sometimes also referred to as "silicon" ROD-BOC, as they are the same ones used for the "silicon" SemiConductor Tracker (SCT) DAQ. Although similar to IBL, the L1/L2 ROD and BOC hardware were slightly modified, along with significant firmware revamps for their FPGAs. Here is the relation L1/L2 to the rest:



A major complication in implementing IBL ROD-BOC technology for L1/L2 is that the IBL RODs do not run the many activities of detector calibration scanning, tuning and statistical operations, including curve fitting that are done on the pixel RODs dedicated DSP processors. These activities have to be moved to a "PC Fit Farm" as illustrated in Figure 11:

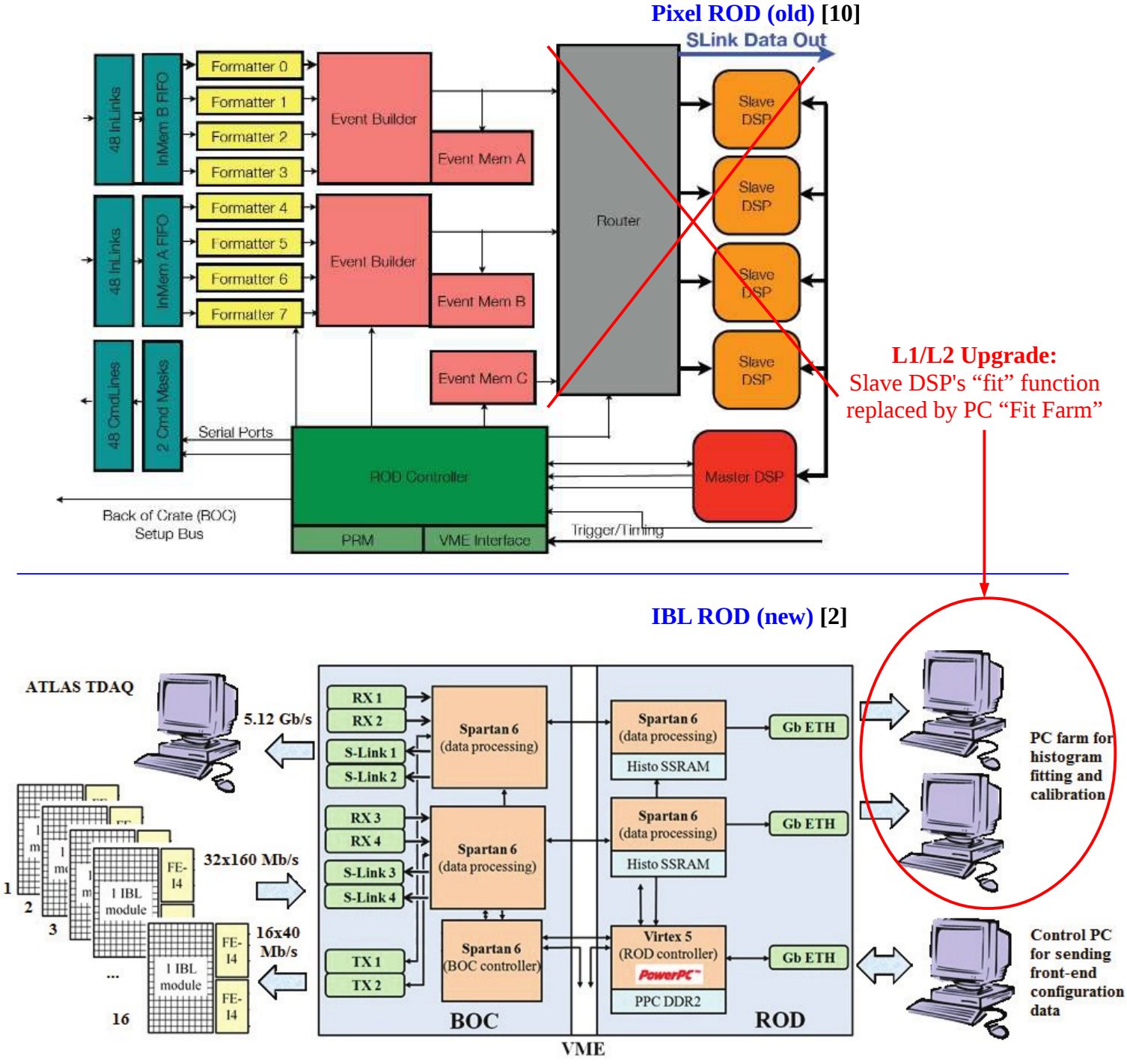


Figure 11

While other software tasks concerning data-taking and optical link tuning also are affected by the upgrade, only pixel tuning and calibration will be discussed in this paper.

Moving L1/L2 calibration software from the pixel RODs to the IBL Fit Farm approach required a thorough understanding of both so all the tools of the former could be included in the necessary modified form of the latter. Fortunately, most of the key pieces of electronics and the principles behind their operation for calibration are not much different between the FE-I3 and FE-I4 chip, so much existing code could be reused in the revamped structure.

6.2 The "SR1" test facility at CERN

Before diving into a description of the main aspects of pixel calibration, a brief mention is provided here of the CERN test facility called "SR1", where all development work had to be done so as not to endanger irreplaceable (because inaccessible) detector equipment. This facility consists of copies of the same hardware used in the detector (in a setup, physically identical other than consisting of only a fraction of the channels). It offers developers access to a limited number of actual front end chips via a fully mimicked set of tools. This includes having access to the electronics via the "Detector Control System" (DCS) which also offers monitoring capabilities of the cooling system and the critical safety interlock mechanisms.

An overview of the intricate power and signal interconnections in SR1 is provided in [11]. The document consists basically of photographs taken by this author that have been superimposed over the extensive connectivity and flow diagrams complied in [12]. It is most usefully viewed in an electronic "presentation" mode that allows instant flipping between entire pages, so that each system subpart displayed can be easily placed within the higher level displays repeated each time immediately preceding it. The naming scheme of both patch panels and cable types follows this simple scheme: Each is numbered in order of increasing distance from the detector, starting with 0 for the nearest ones, i.e. with cable type 0 connecting detector modules to patch panel 0 (PP0), cable type 1 connecting PP0 to PP1, etc.

7 Calibration

To calibrate 90 million pixels in a reasonable amount of time (a few hours), successive layers of parallel actions are spread over 10 VME crates housing a total of 148 RODs each connecting on average to about a dozen modules of roughly 50,000 pixels each (46,080 FE-I3, 53,760 FE-I4). Dozens of Linux processors control these RODs, including a "Single Board Computer" (SBC) in slot 1 of each VME crate. There is also a growing number of the "Fit Farm" PCs for handling calibration processes formerly handled by the silicon RODs, as these are gradually being replaced in the L1/L2 upgrade by the newer IBL-style RODs.

Most DAQ activities are controlled and overseen through the "Calibration Console" GUI, from which it is possible, with just a few clicks of the mouse, to "drill down" from a single panel view of all the pixel detector modules, to inspect the response of a single pixel to a scan.

To describe all the calibration procedures, it is not necessary to deal with the complications associated with all the parallel operations and database interactions required for the various tuning parameters. For the most part understanding how a single pixel is scanned and tuned is sufficient, with only a brief look at making pixel responses uniform over entire modules.

8 Pixel calibration fundamentals

The ultimate purpose of the pixel detector is to accurately determine the trajectories of charged particles generated by the proton collisions at its center. In other words the pixel sensors on each cylindrical "barrel" layer surrounding the collision point have a similar function as the pixels in a digital camera in that they use signals, originating from the effect of passing electromagnetic fields within the semiconductor pixel sensors, to reconstruct a "picture" of the source of those fields, which for the pixel detector, is a particle trajectory.

Because proton-collision particles move through the pixel sensors with velocities approaching the speed of light, they are (if charged) what are called Minimal Ionizing Particles (MIP), meaning the sensors register only the energy of the electric field of the particles and not their kinetic energy (or a least not enough to be detectable). These passing particle fields can affect sensors beyond the one actually traversed. Because of this, using the distribution of signals from an MIP in all the pixels of one detector barrel layer, it is possible to calculate where such a particle intersected that layer to a greater precision than the size of the pixels themselves.

Sensor manufacturing has been done so as to to ensure that (within acceptable tolerances) identical field disturbances send the same current into the electronics of each pixel. At least initially, before significant radiation damage starts to set in, this means that no additional tuning is needed to account for hardware-induced signal irregularities preceding the pixel electronics. All the tuning discussed here therefore concerns adjustments to the electronics past the "bump bond" point of entry into the pixel electronics (see figure 5 above).

Pixel "calibration" involves three distinct activities:

1) **Scans**: Measure signal response to inputs generated by special pixel circuitry designed for calibration purposes.

2) **Tunes**: Adjust pixel electronics settings to ensure uniform response across pixels.

3) **Calibrations**: Iterate through (1) and (2) with different tunes, either for more precise results, or to generate values for parameters needed in reconstruction algorithms other than those available from raw data.

9 Scans

In order of increasing complexity, only these four most important scans will discussed here:

1) Digital Scan: Tests functionality of “digital” part of readout chain for each pixel.

2) Analog Scan: Tests pixel discriminator circuity, in addition to “digital” part of readout.

3) ToT Scan: Measures pixel discriminator response (ToT) to a known current injection.

4) Threshold Scan: Determines charge level of pixel discriminator threshold.

Subsequent descriptions will reference the following **very** simplified pixel circuit diagram:
(In fact there are approximately 1,000 transistors in each pixel circuit! See [4].)

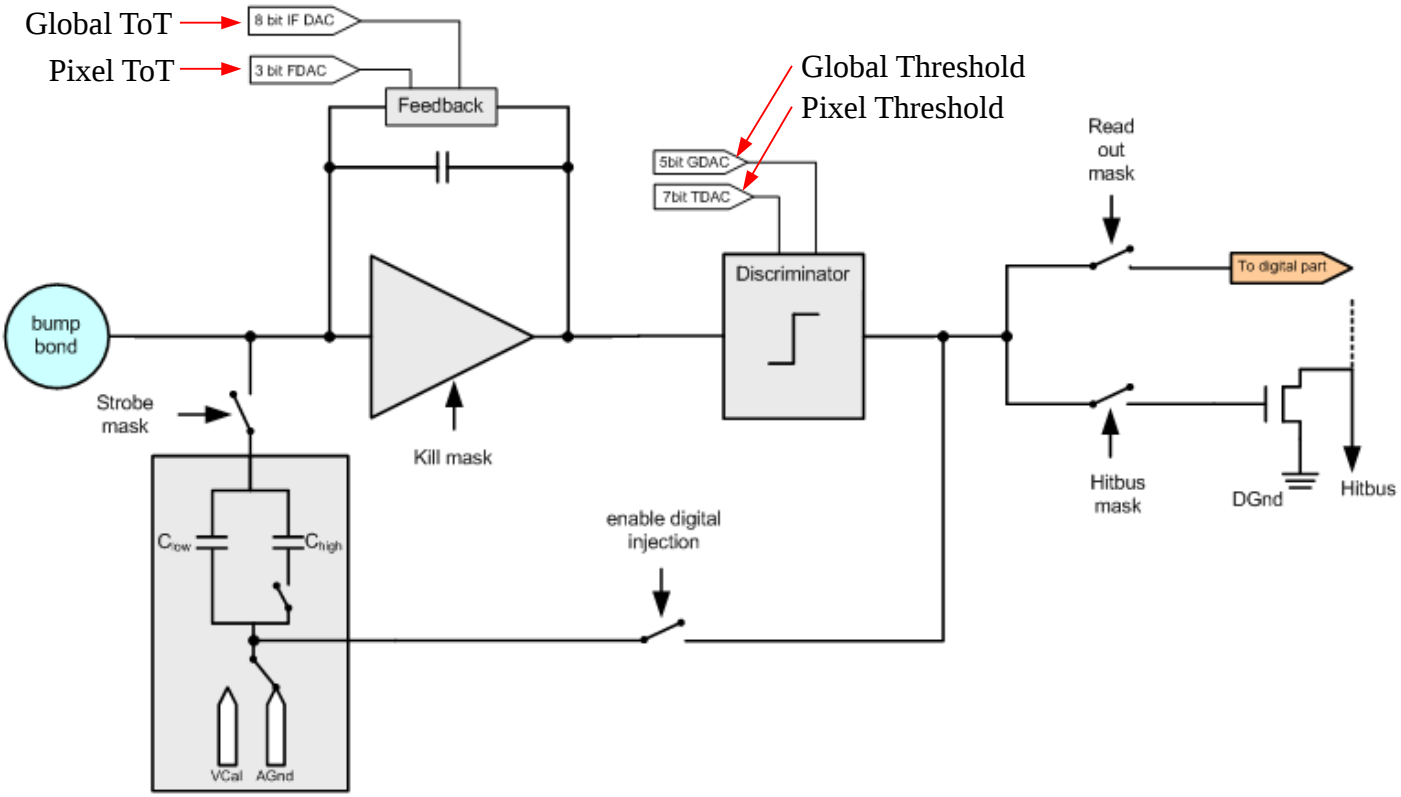


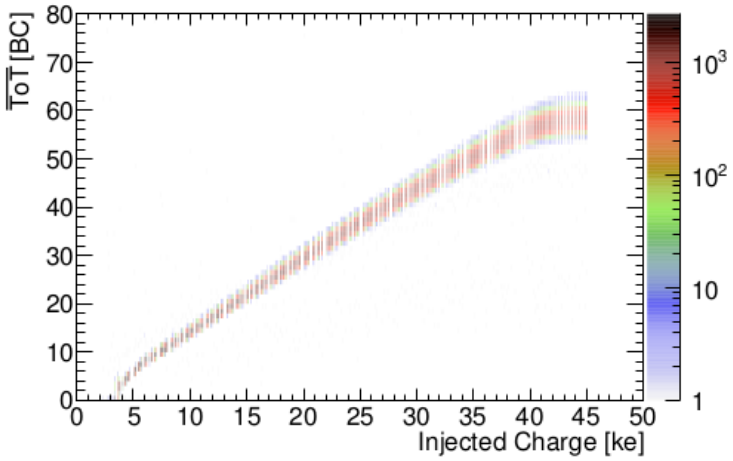
Figure 12 adapted from [13]

9.1 Digital scan (Console Scan Name = DIGITAL_TEST, SR1 preset = 00Preset):
A number (typically a few hundred) of pulses, or "injections"/"events"/"hits" of a certain duration (typically 50 BCUs, or Bunch Crossing Units of 25 ns each) are injected into the circuitry just after the discriminator, to simulate discriminator output of that same number of events, each of the same duration recorded as "ToT". If everything, including count logic in the hitbus, is functioning correctly, the same number of events should be recorded as "occupancy" counts in the histograms of the scan. The existence of problems anywhere along the readout chain after the pixel discriminator are thus easily revealed by occupancy counts deviating from the injected number of pulses. Also, if ToT histograms are requested, should these not correspond to the duration of the injected pulse, this indicates problems such as inability to properly determine leading or trailing edge of the pulse or a failure in the ToT-determination circuitry.

9.2 Analog scan (Console Scan Name = ANALOG_TEST, SR1 preset = 0000Preset):
Repeated injection of a known amount of charge into the adjustable amplifier leading into the pixel discriminator circuit results in occupancy histograms identical to those for the digital scan, from which (after factoring out any anomalies revealed by the digital scan) problems in this "analog" portion of the pixel circuit are revealed. As in the digital scan, ToT values are also transmitted through the readout chain in an analog scan, but ToT histograms are not produced in the official "preset" for this scan. The following separate preset is used for that:

9.3 ToT scan (Console Scan Name = ANALOG_TEST, SR1 preset = 0000Preset_ToT):
In an analog scan, the ToT output by the discriminator depends on the amplifier pulse shape, a setting that is determined through the ToT tuning procedure to be described below. ToT, which stands for Time over Threshold, is the core physical measurement produced by the pixel detector, so it is critical to understand how it is produced, and what it is used for.

Figure 13 shows how it is nearly a linear function of the injected charge:



The ToT versus charge for all non-ganged pixels on a single front-end chip in ToothPix.

Figure 13 [14]

The conversion of a current pulse into a ToT value of time is illustrated in figure 14:

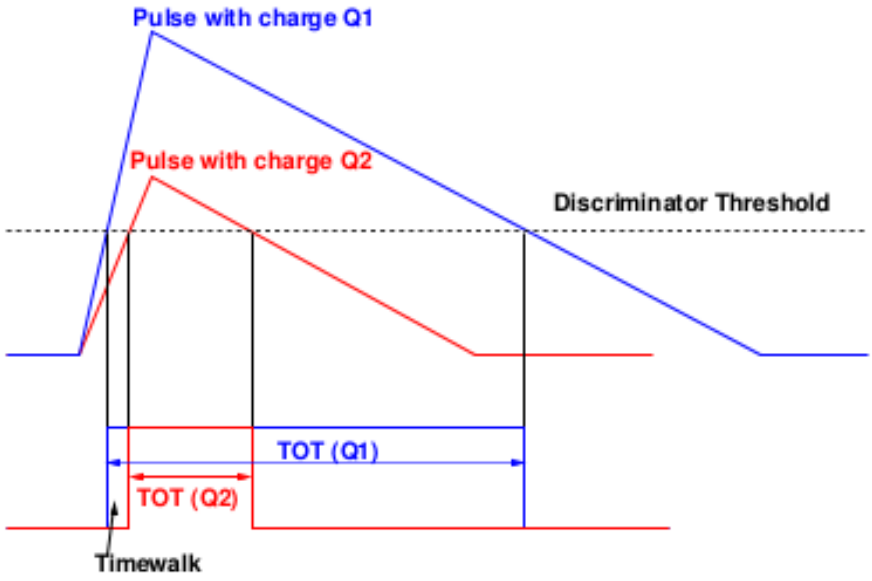


Figure 14 [15]: Schematic of hits with different energy deposition in the detector.

Obtaining the most precise estimate of ToT for the charge produced in pixel sensors by a particle passing through a pixel barrel layer is critical for all physics analysis done using data from ATLAS. Therefore minor corrections are needed to account for some of the details visible in these figures. The slight deviations from linearity seen in figure 13 are accounted for by the results from a ToT calibration not discussed in this paper. The "Timewalk" highlighted in figure 14 may require special handling for very low values of ToT (when Timewalk is most pronounced) by means of a DAQ feature called "ToT doubling": ToT values below (FE electronics-) configurable levels can be copied to the preceding time unit allowing better after-the-fact event reconstruction from the raw data.

From figure 14 it is clear that ToT for a given charge level depends on slope of the trailing edge of the charge's pulse and the discriminator threshold level, which are respectively adjustable via the amplifier feedback registers (labeled "IF DAC" and "FDAC" in figure 12) and the threshold level registers "GDAC" and "TDAC". The former are determined by the ToT tuning and the latter by Threshold tuning procedures to be described below.

But of what use is ToT in event reconstruction? Frequently particle detector ToT measurements are used to calculate kinetic energy. However the high speed "MIP" particles in the pixel detectors are not suited for that, since they generate more or less the same ToT response regardless of their kinetic energy. What matters for the pixel detector is to be able to accurately pin-point the position of charged particles, and for this it is only important that all pixels produce a uniform ToT response per unit charge generated in the semiconductor senor from the electric field of the passing particles. This field of course peaks in the sensor the particle traverses, but neighboring sensors also react to the field, producing ToT values that

decrease with distance from the pixel layer intersection point. If all pixels are uniformly tuned, it is then possible to use the ToT values of the "cluster" of pixels on a barrel layer associated with a given particle "event" to find it's position at a higher resolution than the dimensions of the pixels themselves! So this is how ToT is used in the pixel detector and why its uniform tuning across the entire detector (or at least individual layers thereof) is so crucial.

9.4 Threshold scan

(Console Scan Name = THRESHOLD_SCAN,
SR1 preset = 000Preset_full):

The pixel discriminator threshold is defined as the charge level for which the discriminator registers a "hit" on average 50% of the time. This is visualized in the typical "S-curve" plots of figure 15. Both are plots of pixels whose thresholds are tuned at a charge of 4000:

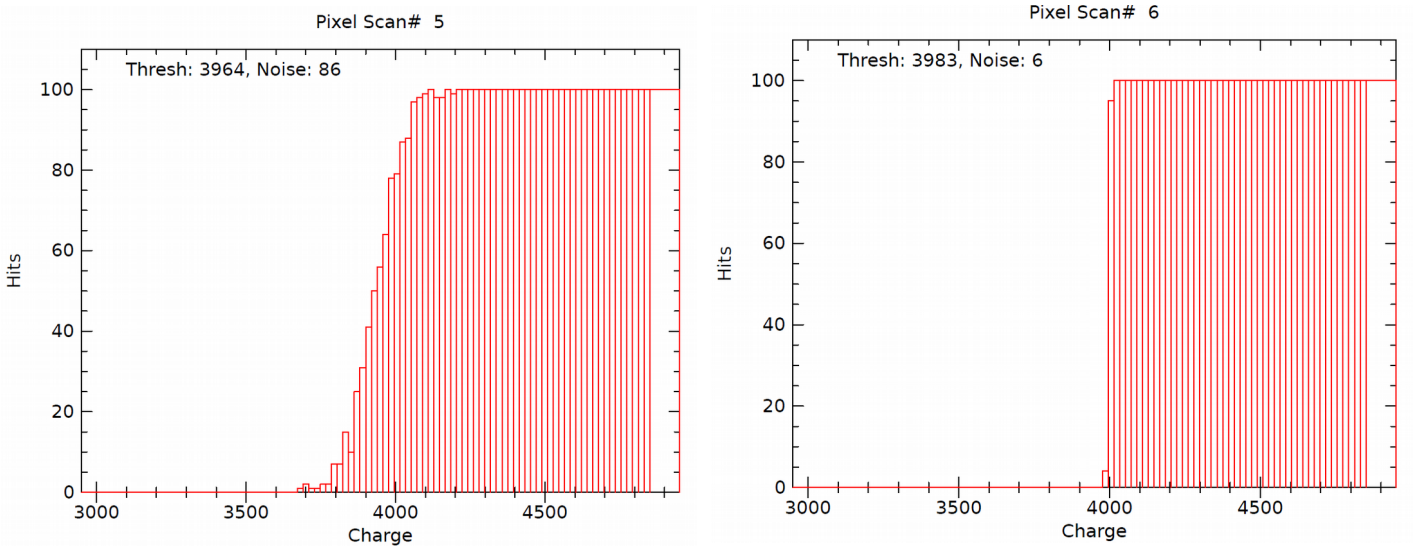


Figure 15 [16]: Emulated pixel hits. Large noise (left), small noise (right)

To find the threshold charge, one runs a number of analog scans over a range of charge levels, *V*, and finds the best S-curve $f_{\mu,\sigma}(V)$ for the average occupancy of the scans at each level *V*. The S-curve parameters μ and σ so found are the expected threshold and its sigma respectively.

This is the most complex as well as time consuming of the scans, as it not only requires running a large number of analog scans, but also because of the curve fitting. The fitting involves solving for the best two parameters μ and σ in $f_{\mu,\sigma}(V)=N\left(1+erf\left(\frac{(\mu-V)}{\sqrt{2}\sigma}\right)\right)$.

where *V* is the voltage parameter (labeled VCal in figure 12) used to set the charge injection level, *N*, the number of injections and *erf*(*x*) the error function in figure 16:

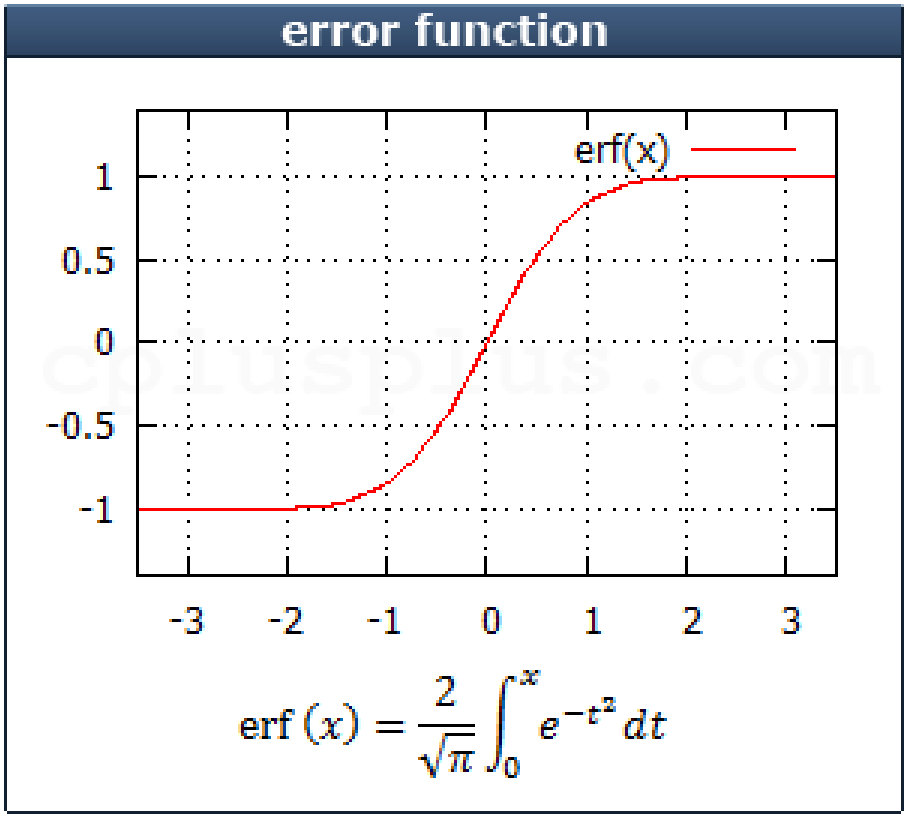


Figure 16

If, for a given pixel, $Oc(V)$ is the average occupancy count over a reasonable number (usually 20-30) analog scans of N injections at VCal = V, μ and σ are determined by minimizing $RMS = \sum_V \left(Oc(V) - f_{\mu,\sigma}(V)\right)^2$ using Levenberg-Marquardt least square fitting.

Note that the fitting is done with V in the actual VCal units of the pixel circuitry. Each FE chip comes from the manufacturing process with the four parameters for the following 3rd-degree polynomial that converts VCal (V) to charge (q) for the capacitance, C, of the analog injection circuitry (see C_{low} and C_{high} in figure 12):

$q = 6.241495961\ C\left(d + cV + bV^2 + aV^3\right)$. For each of the FE chips, C, a, b, c, and d are retrieved along with a large number of other parameters from a module configuration database before the scan is run.

Note: The same equations for q apply to **all** pixels on a given front end chip.

10 Tunes

The tunes discussed here are the two most significant ones, i.e. Threshold and ToT, which set the corresponding registers indicated in figure 12. Note that both a "Global", FE-chip level and a separate pixel-level register have to be set for each of the two types of tunes.

10.1 Threshold tune (For more in-depth coverage, see sections 2.2 and 2.3 in [17])

First the pixel-level TDAC registers are tuned via TDAC_FAST_TUNE, targeting the threshold for each pixel at a charge level preset for this tune (usually 3,500 or 4,000 e^-).

The chip-level GDAC register only needs to be adjusted after TDAC if any chips end up with a large number of pixels having TDACs tuned to the maximum value. In that case GDAC_TUNE needs to be run, followed by another TDAC_FAST_TUNE and another check to confirm that there are no more chips with large numbers of pixels at maximum TDAC.

A good threshold Tune will yield a threshold scan SCURVE_MEAN module-level histogram with sigma of well under 100 e^- as seen in figure 17:

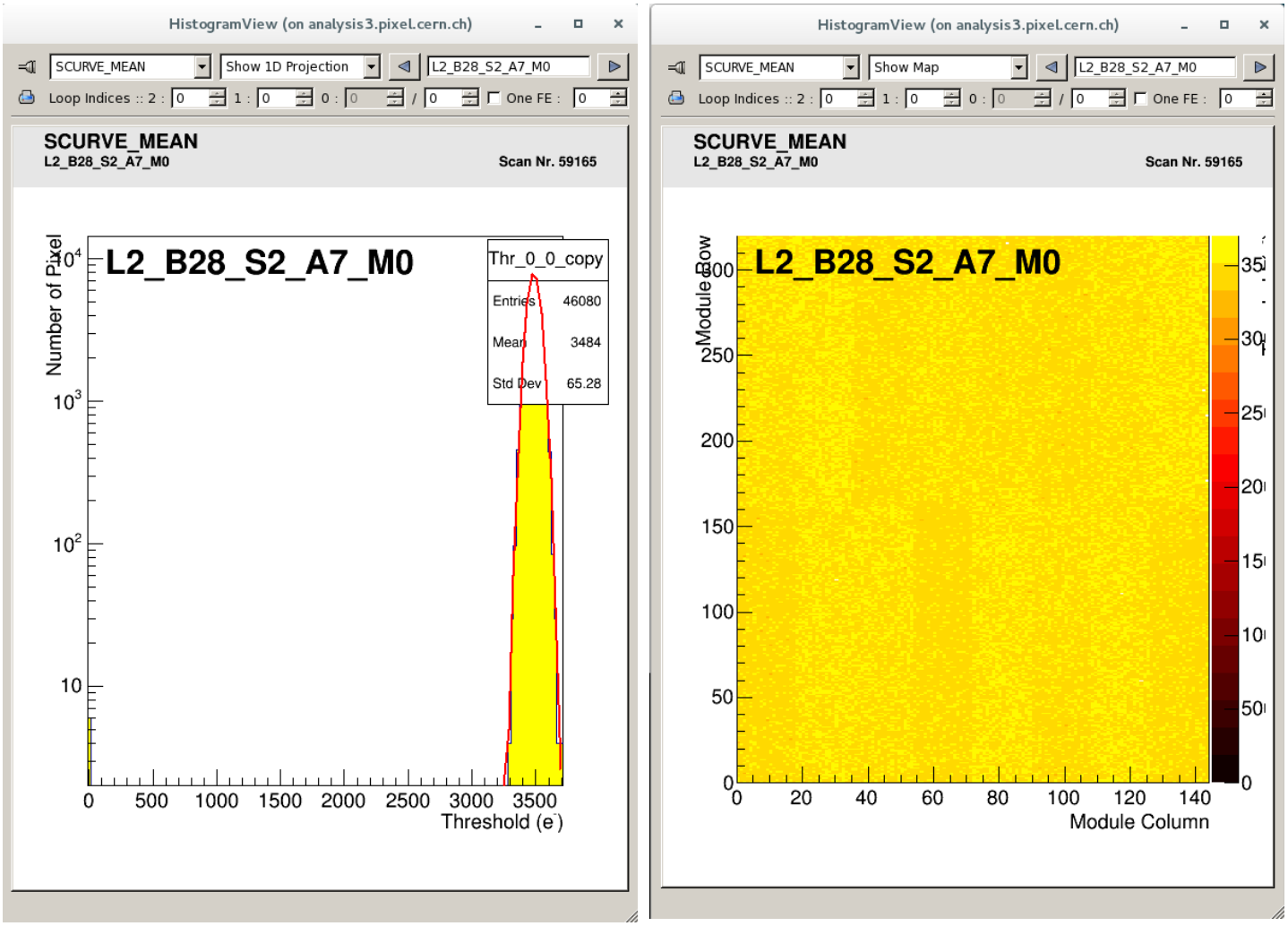


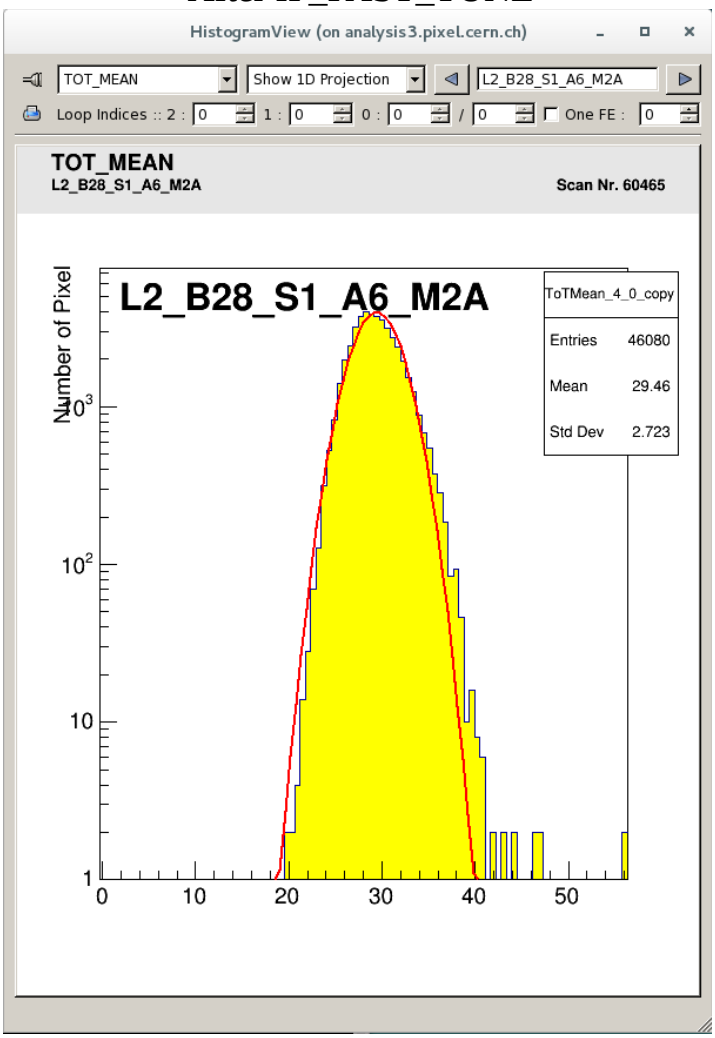
Figure 17

10.2 ToT tune (For more in-depth coverage, see sections 5.1 through 5.4 in [14])

First the chip-level IF DAC register is tuned via IF_FAST_TUNE, targeting an average ToT of 30 BCU over all pixels on each module at an injected charge level of 20 ke^-.

The pixel-level FDAC registers are subsequently tuned with FDAC_TUNE, targeting the same 30 BCU at 20 ke^-. This should reduce the module-level ToT dispersion by a factor of 3-4 as illustrated in figures 18 and 19:

After IF_FAST_TUNE



After FDAC_TUNE

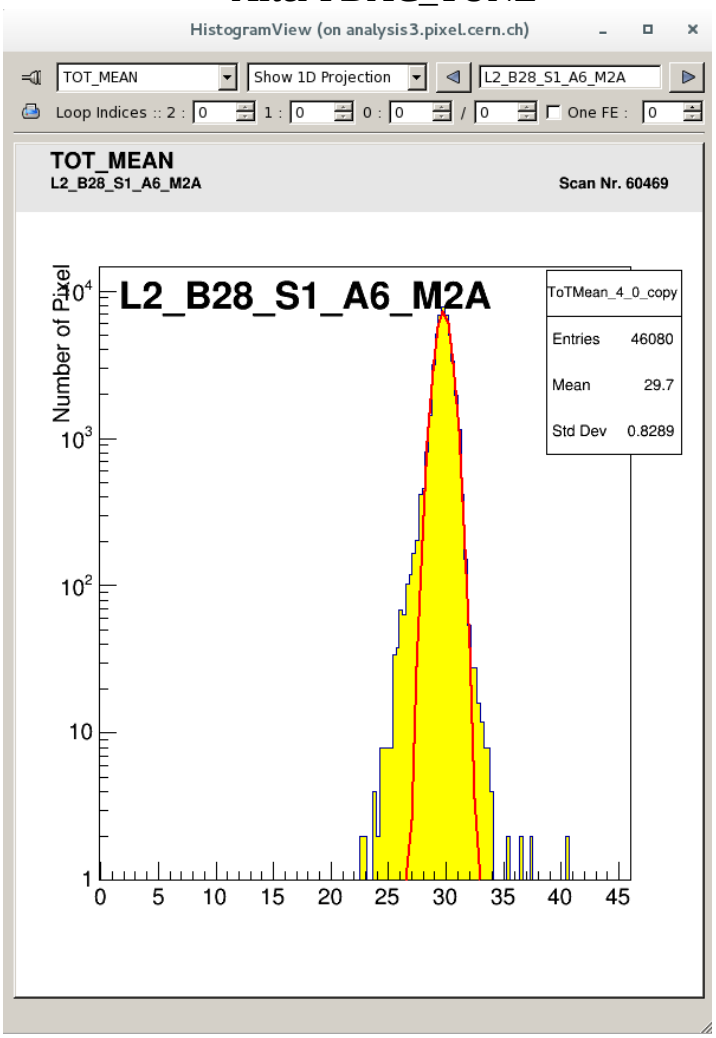
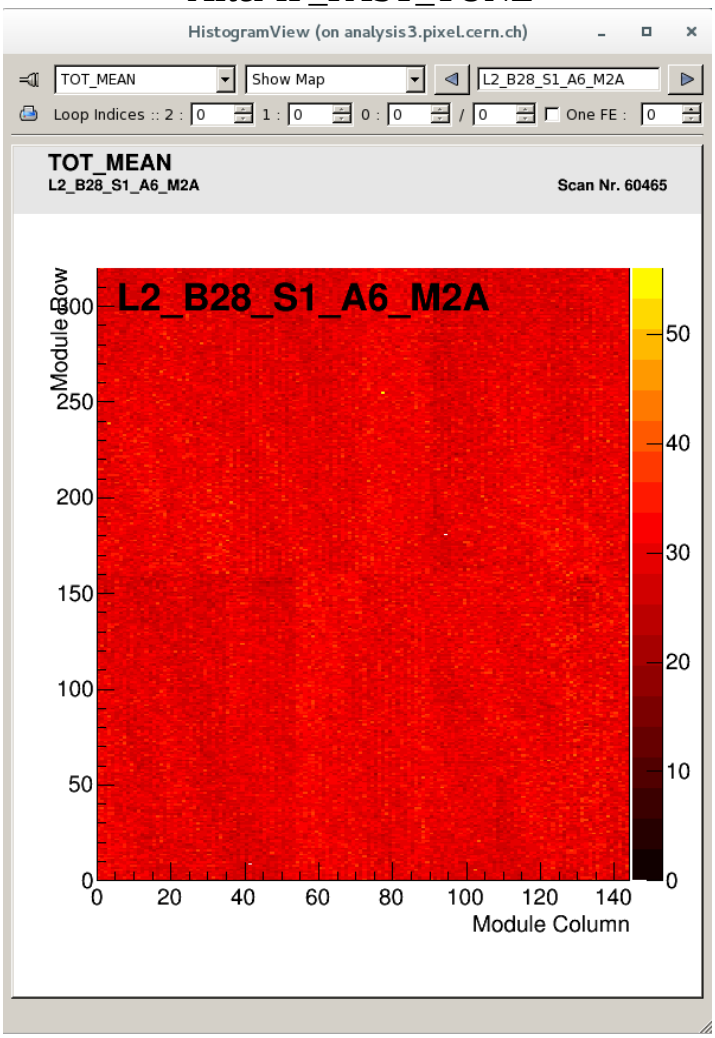


Figure 18

After IF_FAST_TUNE



After FDAC_TUNE

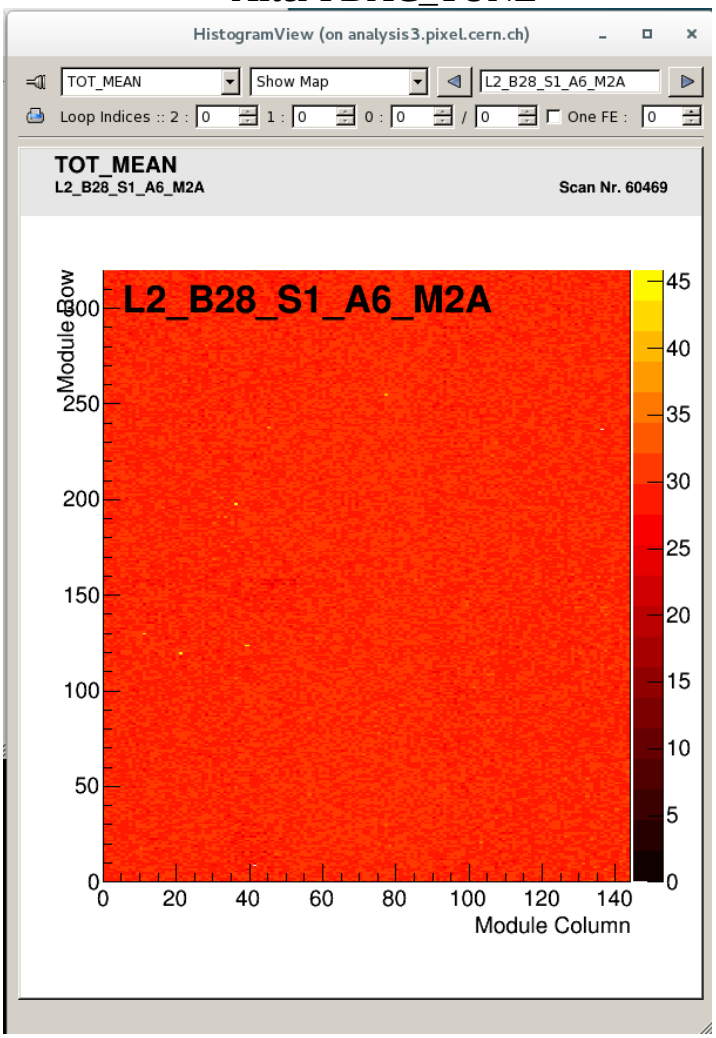


Figure 19

11 Relation of calibration software to hardware

The main software interface to the pixel detector for DAQ purposes is a graphical user interface (GUI) called the "Calibration Console", or sometimes just "Console" for short. Other user interfaces include DCS, dealing with more fundamental aspects of the detector such as power and cooling, along with certain specialized tools like "PixDbBrowser" or "TagManager", used to manage configuration parameter databases. Only the Console will be briefly described here, since it is the principle vehicle for controlling detector calibration.

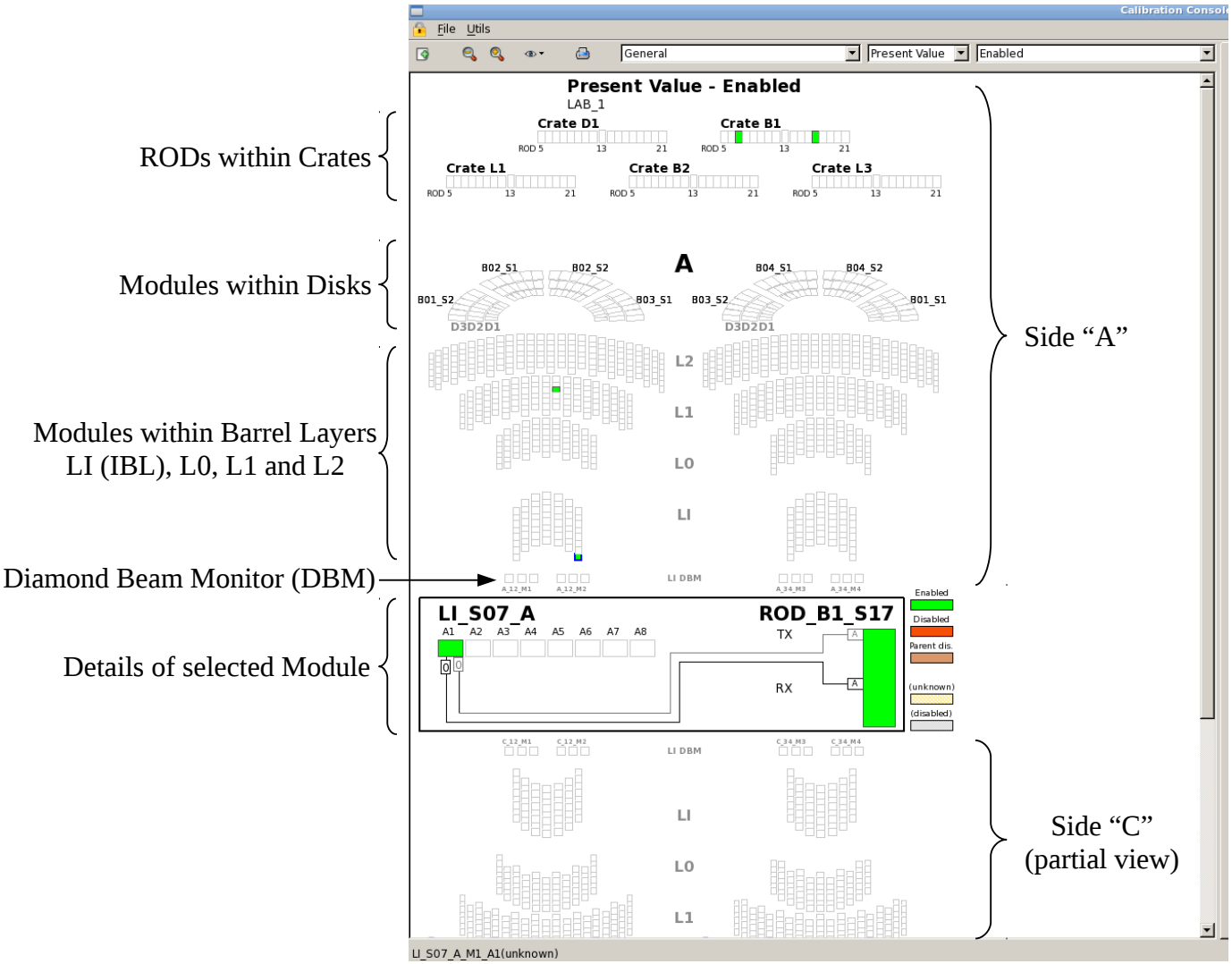


Figure 20

Figure 20 was created in the Seattle lab using a "dummy console" configuration containing just two modules: One green on L1 and one blue ("selected") on the IBL layer "LI". On the actual running detector, of course, almost all modules would typically be green, i.e. "enabled". The console has utilities for viewing configuration parameters of individual FEs and panels from which to configure, initiate and view the results of scans, tunes and calibrations. The Console software is not the subject of this paper, but rather all the "machinery" initiated by it.

The highest level of interaction between the various calibration software components and hardware is illustrated in figure 21:

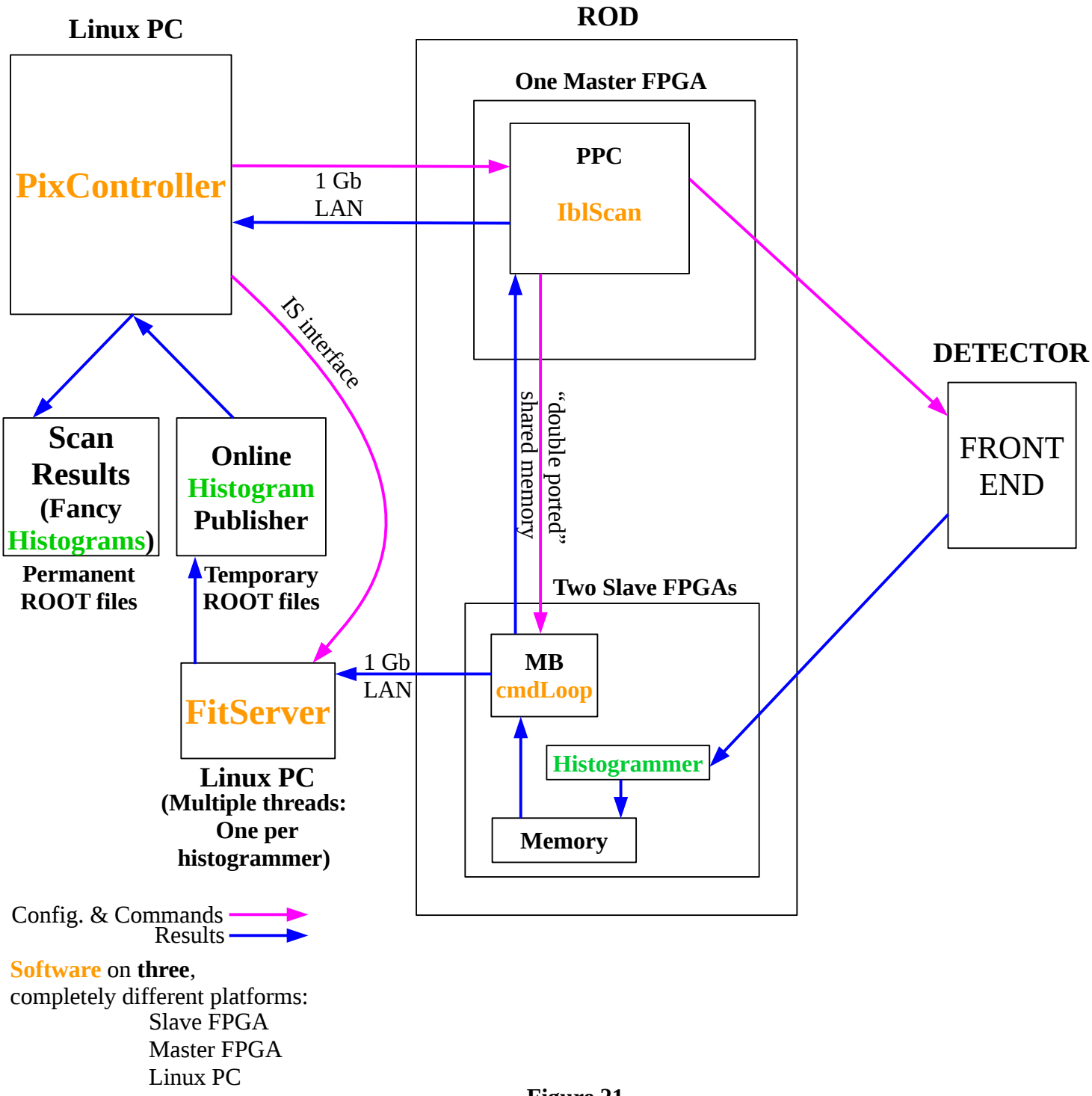


Figure 21

In addition to the numerous and varied types of communication channels dealt with by the software, the diagram also shows that 3 different processor and histogram types are involved.

The complexities of communication (a vast topic in and of itself) will be left aside, but it is necessary to understand how the different processor platforms affect the software, and why histograms for the same underlying data (just occupancy counts, ToT μ and σ) are built in three different locations.

Everything outside the RODs runs on 64-bit Linux (SLC6) operating systems. The Fit Farm consists of dedicated machines (usually running a single FitServer instance) but there are many more machines for tasks not of interest, and therefore not shown here. The master FPGA on the ROD contains a Virtex-6 "Power PC" (PPC) processor that runs 64-bit C++ code in a special Xilinx environment. The two slave FPGAs on the ROD each contain a Microblaze processor that runs 32-bit standard c code (**not** C++), also in a special Xilinx environment. The standard output (stdout) stream for all three ROD processors feed to a common UART port, only one of which can be monitored at a time, via a setting controlled by a register in the master FPGA.

It is apparent that the slaves have the most primitive of the processors, and the software is correspondingly simple, doing nothing but start their histogrammers (themselves very simple Digital Signal Processors, or DSPs) and then, when they are done, sending the results via a network connection to their respective FitServers. All these activities of the slaves are directly controlled through the PPC.

The master can run more sophisticated object-oriented and multi-threaded code, which it needs to in order to be able to simultaneously communicate with a variety of host controller processes, the FEs (via the BOC) and the slaves.

The Linux "host" machines are generally the most powerful and versatile Linux-capable processors commercially available. About 100 independent, simultaneous processes run on an indeterminate number of these. We will collectively refer to them all as "host", treating them as a single processor running multiple concurrent processes. In fact, how the processes are distributed across Linux machines is pretty much arbitrary and configurable via xml files, so in theory they all could run on a single machine and thus nothing is lost in our discussion by treating them as if they were. Coordination of everything running on "the host" is handled by a system (of course also running on "the host") called "Inter Process Communication" (IPC) based on "Coordinated Object Request Broker Architecture" (CORBA). There is also a more simply accessible "Information Service" (IS) involved. Aside from the occasional, (as needed brief) mention, none of these will be any further elaborated on in this document.

Similar to the three progressively more powerful processor platforms, calibration scan results are aggregated into three levels of increasingly complex forms of histograms. This is because of the different levels of information available at each processing stage:

The slaves can only identify each pixel by its histogrammer unit, FE chip, row and column

number coming to it from the (up to 7 modules worth of) FEs to which each “histoUnit” has been assigned. There is no information at this stage about the relation of these pixels to the modules they are on, much less the detector as a whole. Three separate values for each pixel, namely “occupancy” (or hit) count, ToT and ToT-squared are accumulated, as the “hit” data packets for a given chip-row-column pixel “address” within the module on the link reach the histogrammers. No more information is needed for building all subsequent necessary statistics, and so, to keep slave FPGA processing to a minimum, this is all that it is designed to handle. Once the slave histogrammers have completed their job for all the pixels in a scan it was asked to process, the slave sends the three accumulated values just described (in one big block for all those pixels) on to the next level of histogramming in the FitServer.

The FitServer is able to associate all RODs and communication channels to the module-level data packets received from the slave, and generate sets of the following three different generic Online Histrgraming (OH) chip-level root histograms for all pixels per channel: Occupancy, ToT (mean and sigma) and S-curve (mean, sigma and chi-square). Because these histograms are chip-level only, they are too unwieldy for regular inspection other than during debugging. If all goes as intended, the PixController immediately converts then (yet once again) to the more useful histograms accessible directly from the console.

The PixController specific to each ROD creates those final, fancier “console histograms” out of the generic chip-level FitServer ones by aggregating the chip data by module so it is viewable with the standard (and quite versatile) histogram viewing tools available from the console (which are powerful enough to allow the values for individual pixels to be instantly seen by simply hovering the mouse pointer over the pixel in displays mapping entire modules).

The creation by the FitServer of temporary “OH” histograms (which is not done by the DSP histogramming code of the old Si-RODs) has been questioned, but in the end it was agreed that the advantage of keeping the FitServer simple (by not passing to it, and then have it process all the auxiliary information needed to build the console histograms) outweighs the increased disk storage requirements of the “OH” histograms, especially as they are temporary anyway.

12 ROD processor code initiation and control

Since there is no direct human interface with the two types of ROD processors, code on both is loaded and started/initialized either via a Xilinx JTAG connector or the VME crate interface to the ROD using the respective special scripts iblRodCtrl and iblRodCtrlVme. Both run custom processes that wait in infinite loops for commands to control them, with actions for the slaves from the host additionally requiring their commands to be routed through the master.

After a quick summary of the processor interfaces, some illustrative code will be introduced by pointing out the function of key processes that are initiated, and in the case of the most complex host actions, tracing the long sequence of functions started and major classes instantiated along the way. Separate sections follow with details of the most significant host-based (under "PixLib") and ROD-based "Host Commands" classes.

13 Processor interfaces

13.1 Slave-master interface

The key pieces of code [18] handling the slave-master command interface are:

IblRodSlave::sendCommand	To send command to slave from master.
histClient_main.c	To initialize slave and start infinite-loop wait for commands.
cmdLoop.c	To interpret and act on slave commands sent from master.

The "cmd" variable in the slave's cmdLoop.c routine points to a special type of memory called "double ported memory" that the master also accesses through IblRodSlave::sendCommand. This ROD master processor function places the commands it wishes to pass on from the host-based commands it received via the master-host command interface. It will also retrieve any results the slave may place into the double ported memory in response to such commands.

13.2 Master-host interface

The master-host command interface is an extremely powerful (and therefore complex) set of tools, generically referred to as "Host Commands", that can be used in anything from simple command line programs to the most complex detector calibration and live monitoring routines. Processing of these commands runs through the "ServerInterface" abstract class, with details implemented in the "Server", "IblRodModule" and "Client" classes. More details are described in later sections 15 and 21, but the basics of the two most common implementations can be understood by digging into the following illustrative pieces of code, and (especially in the second) learning how they are used:

Listing all available Host Commands by running a simple command line program on the host:
HostCommandPattern/tools/compareCommands.cxx (code to start on host command line)
CmdFactory/Production/src/ListCommands.cxx (code that runs on **both** master and host)

Calling a Host Command from routines running on host while detector is operating:
BarrelRodPixController::sendCommand

14 Getting ready for calibration on host: Processes traced from PixActionsServer

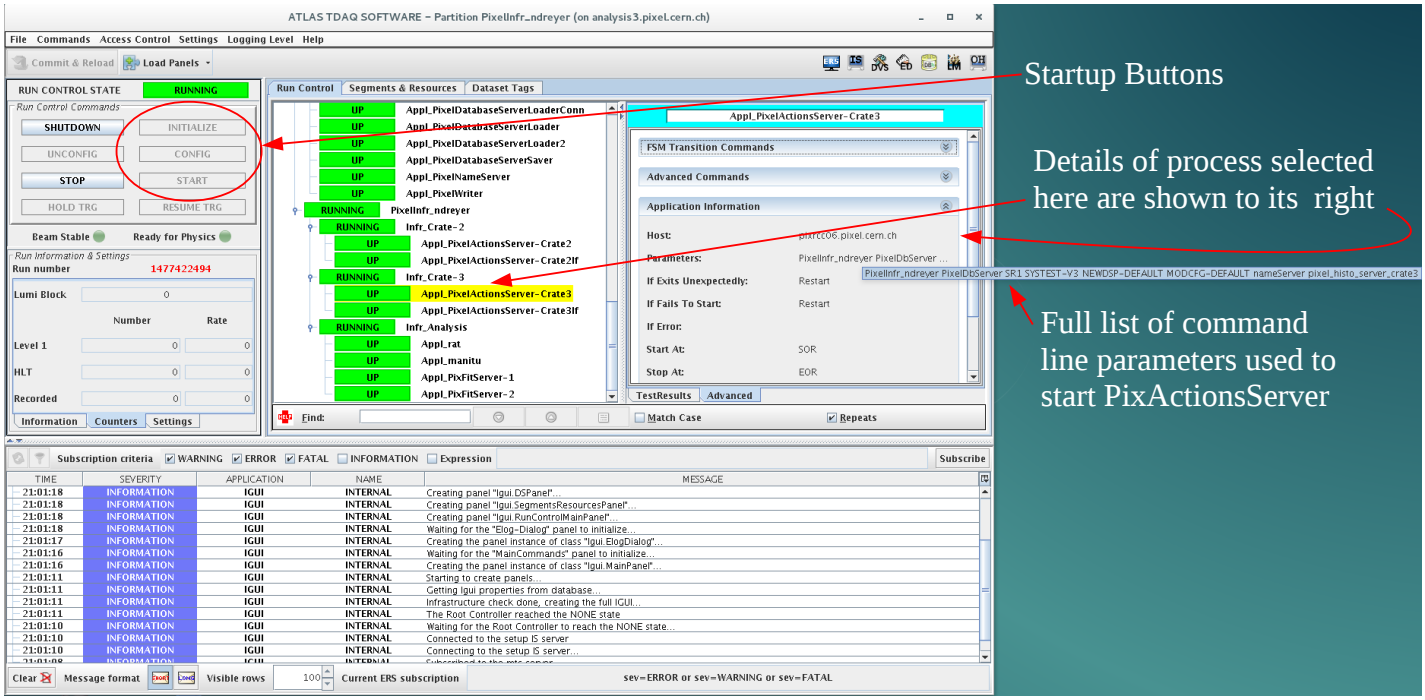


Figure 22

The script “start_infr” launches the detector software “Infrastructure”, which must then be “initialized”, “configured” and “started” by clicking, in that order, the corresponding buttons on the GUI panel (see figure 22). The “START” button ultimately brings up the critical software components controlling the ROD. For calibration only (much else brought up along the way being ignored), these pieces are traced out here, in startup order (as indentations move back “out” to the left, the calling function is found from the preceding alignment at that level):

PixActionsServer	This is the “main” program from which all calibration activities (and anything else directly involving the ROD) are ultimately launched. It runs on each crate SBC and instantiates
PixBrokerSingleCrate	Its constructor then, for each ROD, if the variable rodBocType is CPPROD or L12ROD, i.e. is “IBL-style”, instantiates
PixActionsSingleIBL	This class inherits **PixActionsSingleROD** and **PixActions**.
setupDb	is called for its ROD, which creates a new instance of
PixModuleGroup	whose constructor runs
initDbServer	whose function initConfig instantiates a “general” ConfGroup object and whose ctrlType for the ROD, having been read from the connectivity database, determines the version of
xxRodPixController	to instantiate, where **xx** = “” by default and **xx** = “**Barrel**” for ctrlType == “L12ROD”, the only ROD type discussed here.
allocate (via IPC call)	“reserves”/allocates **PixActionsSingleIBL** object/ROD. Console scan initiation causes IPC to start up these functions

initCal (via IPC call) which calls loadConfig, which calls
m_moduleGroup->downloadConfig which calls writeConfig, which calls
m_pixCtrl->writeModuleConfig which sends front end configurations to the ROD.
setPixScan (via IPC call) which creates a new instance of
PixScan which contains all scan parameters and generic scan functions used by **PixModuleGroup**, **xxRodPixController** and the **ROD software**. Note that the FitServer, because running in an independent thread, has to create its own instance of PixScan, of course getting the same parameters loaded from the same scan configuration root file used in the "Actions".
scan (via IPC call) which instantiates
PixScanTask from where initialize() runs these **PixModuleGroup** functions
prepareScan(scn) which does
scn->initScan(this) which does miscellaneous scan setup activities.
scn->scanLoopStart(2, this) which only does things for ToT tunes:
prepareFDACTuning(2, this) for FDAC and
prepareIFTuning(2, this) for IF DAC tunes.
for(n=2; n>=0; n--){
m_scn->loop(**n**) which does housekeeping
m_grp->prepareLoop<**n**>(m_scn, …) which does
scn->prepareStep(n, this, …) which mainly handles scan loops not on ROD/DSP.
if (**n** = 0) {
scanExecute(scn) which for a NORMAL_SCAN does
m_pixCtrl->writeScanConfig(*scn), which sets up ROD as described below*.
m_pixCtrl->startScan(), which starts scan on ROD, to be described below*.
} else {
scn->scanLoopStart(**n**-1, this) which only does things for ToT tunes:
prepareFDACTuning(**n**-1, this) for FDAC and
prepareIFTuning(**n**-1, this) for IF DAC tunes.
}
}.

After **PixScanTask**::initialize() has completed, whose launching via **PixThread**::setNewTask is the last thing done by **PixActionsSingleIBL**::scan, **PixScanTask::**execute() is somehow caused to be launched from (I think??? . . .) the routine **PixThread**::run(), which gets started via startExecution() in /daq/slc6/tdaq/tdaq-05-05-00/DFThreads/src/DFThread.cpp.
I have not been able to confirm it definitively, but all log outputs indicate this must somehow be done via console software just after **PixActionsSingleIBL.cxx**::scan is called through IPC.

PixScanTask::execute() drives scan cleanup, including the making the of "fancy" histograms.

* The "workhorse" code of NORMAL_SCAN types, **which are all we discuss here**, are in the

xxRodPixController class on the host and in the master ROD software it interacts with. The two most significant of the Controller functions, writeScanConfig and startScan, which are called within prepareLoop0, drive the ROD software and incorporate the bulk of the special features highlighted in this document. The more detailed description of this part of the software that follows will lead us naturally into a more complete discussion of the Host Command interface between ROD master and host processors, as well as a few of the other more complex class constructions.

The "n" designations on the "loop" routines listed above are the nesting levels of the scanning loops, the innermost being "0". On the IBL-style RODs only 0-level looping is done on the ROD, and this is always the mask stepping, which is also all that has to be handled by the FitServer. Thus for simple digital or analog scans, the host software does no looping at all (all loop counters are 1), i.e. those scans are complete at "step count 1" of level 1, and so it is at that level that the **PixScanTask::**execute() cleanup process creates the console histograms. On the other hand threshold and ToT scans are completed at level 2.

From now on **BarrelRodPixController** will be the only controller class discussed, and will sometimes be abbreviated as **PixController**, as done in figure 21, or even just "controller". While this could become confused with the actual **PixController** class, which is the base class inherited by all the controllers, we will not get into such technical distinctions here, and so doing this will not hurt the discussion that follows.

15 PixScan configuration sent to ROD with BarrelRodPixController::writeScanConfig

Before this is run within prepareLoop0, all the many front end settings will have already been sent from host to ROD via the controller function writeModuleConfig. This is quite a different, and in some ways even more complex interaction between host and ROD involving completely different sets of nested module and FE classes on each platform. A limited attempt to describe how writeModuleConfig works is found in section 21 at the end of this document. In relation to the scan configuration in this section, we just mention here that of the vast array of FE settings pulled out of the module configuration database and sent to the ROD by writeModuleConfig, the few that must be specifically controlled by the scan are **re**sent to the ROD in the separate **PixScanBase** class by writeScanConfig, to be updated once again in the ROD-side FE classes and then propagated to the actual hardware registers by startScan just before it launches the ROD scan process, as described in section 16.

PixScanBase, which is inherited by **PixScan,** is the class containing all the scan parameters and the simple methods to manage them. On the host these variables are all loaded from a root file whose name is exactly that of the preset chosen (from the console) in a folder whose name is the exact name of the scan selected (from the console) that itself is contained within the folder /daq/db/scan-cfg. This scan "database" of root files is read by the function

PixScan::readconfig in **PixScan**'s constructor and the values in **PixScanBase** then find their way through a rather devious "back door" of pointers to them, setup in a **Config** class object named "ScanConfig" created right beforehand in function **PixScan**::initConfig. Looking at writeScanConfig, one sees that it does not seem to do very much. In fact it just copies all the scan parameter variables from the host into the RODs instance of the **PixScanBase** object. What is complicated is the edifice of **Config** and Host Command class structures (see figures 23, 24 and 25) that make the code in writeScanConfig doing the passing of scan parameters from host to ROD look so simple. A good way to understand these complex classes is therefore offered by tracing their use in the example of writeScanConfig. We look first at how scan configurations make their way into **PixScanBase** on the host (actually done before writeScanConfig is run) using a **Config** object "ScanConfig", and then how **PixScanBase** gets to the ROD via the Host Command mechanism.

15.1 Config class: How scan parameters are loaded into PixScanBase on host

The **Config** class allows creating an almost unlimited variety of objects for configuration parameters, and is used all over the pixel DAQ software. This limited description of its use in **PixScan** should be helpful for understanding its more general features, and to that end we also include a simplified map of the **Config** class structure at the end of this section. The description of its use in **PixScan** will be made even simpler by just tracing how a single parameter, the number of triggers per scan sent to the pixels, gets into the private **PixScanBase** variable m_repetitions.

As already indicated in the section tracing the initialization of PixActionsServer, we know how **PixScan** is created on the host. Pursuing this down to the individual parameter level, the

PixActionsSingleIBL function
setPixScan (called via IPC from the console, as already indicated above) creates a new instance of **PixScan** whose constructor calls
initConfig which builds a pointer to m_repetitions with these statements:
m_conf = new **Config**("ScanConfig")
Config& conf = *m_conf
conf.addGroup("general")
conf["general"].addInt("repetitions", m_repetitions . . .) which puts an instance of **ConfInt** into the vector conf["general"].m_param whose constructor
ConfInt::ConfInt(std::string nam, int &val . . .) does
m_value = (void *)&val

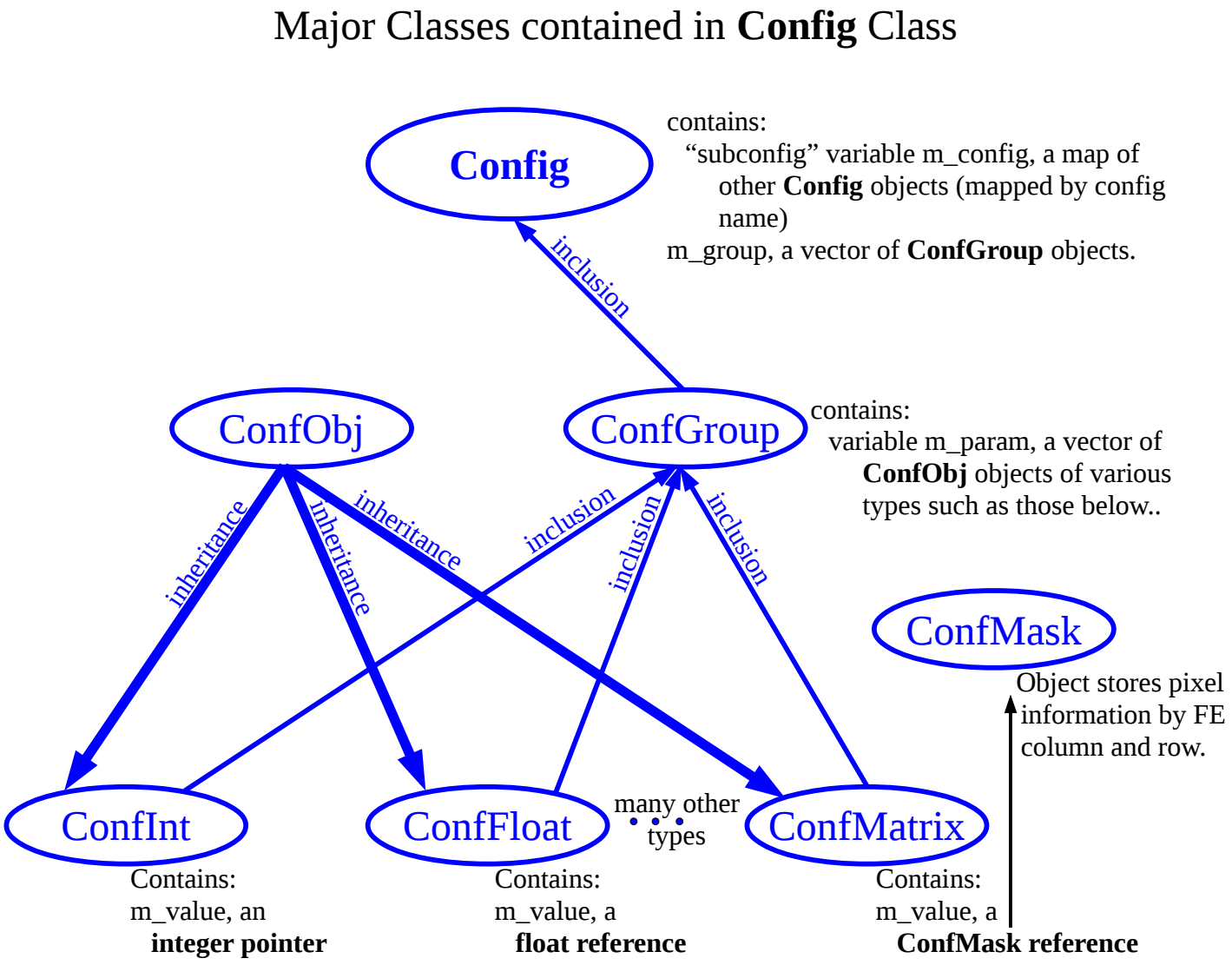
m_repetitions passed by reference as val to ConfInt

& means val passed as **reference** parameter

& means take **address** of val

readConfig which gets the scan configuration root file value into m_repetition with
conf["general"]["repetitions"].read(**DbRecord** *) which reads the root-file value for "repetitions" and does
*((int *)m_value) = value where value is the desired root-file value.

How does this work? The last statement assigns the database value to the value pointed to by the **ConfInt** object's pointer m_value. But looking at the statements presented above, one sees that m_value was made to point to m_repetitions by passing the latter **by reference** to the **ConfInt** constructor and making the **pointer** assignment m_value = (void *)&val. By this "back door" method all parameter variables of **PixScanBase**, which is inherited by **PixScan**, are given the values in the scan configuration root file. Figure 23 gives a more general (though still rather simplified) view of the **Config** class.



m_value can point/refer to a variable anywhere in the PixLib host code (from whereever the ConGroup m_param ConfObj constructors are called from).

Figure 23

15.2 Command class: How scan parameters are copied from host to the ROD

BarrelRodPixController::writeScanConfig(**PixScan&** scn)

recasts the PixScan pointer &scn to another pointer scnBase of the **PixScan** parent class, **PixScanBase** (with the assignment **PixScanBase*** scnBase = &scn), then creates a **SerializablePixScanBase** object serScnBase, using scnBase in its constructor. The scan parameters in the copy of scnBase in serScnBase will get copied to the RODs own **PixScanBase** object by the serialize/deserialize functions working on yet another copy of scnBase within yet another object of class, **PixScanBaseToRod**, whose function setPixScanBase is what makes this copy.

In short, after the host command class **PixScanBaseToRod** object pixScanBaseToRod is created, the last two statements in writeScanConfig are

pixScanBaseToRod.setPixScanBase(serScnBase)	creating the copy of scnBase from serScnBase to pixScanBaseToRod
sendCommand(pixScanBaseToRod)	serializing scan parameters on the host, transmitting then over the network and deserializing them on the ROD

A close look at the **PixScanBasetoRod** class reveals that it is a fairly generic shell with nothing very specific to do with the scnBase variables other than making a copy of them via its setPixScanBase function. The sophistication is buried behind the sendCommand functions, of which there are four: The one in **BarrelRodPixController** calls the one in **IblRodModule**, which calls the one in **ServerInterFace**, which calls the one in **Link**. The last one manages the network transfer, calling the host **PixScanBasetoRod** serialize function (which calls the **SerializablePixScanBase** serialize function) just before transfer, and right after transfer, the corresponding deserialize functions on the ROD. The central position of the **Link** class is seen in figure 24, which shows - except as noted - how the entire host command class structure exists on **both** host and ROD. Except for minor alterations in **Link** because of some networking details, the identical C++ code for all those classes is compiled on both platforms.

15.3 Command class: Some secrets for how to create a new host command class

Instructions and the sample classes built inside of Command.h give more precise details.

Command.h and Serializer.h must be included in the class header file one way or another. RodCommand.h includes Command.h, so the latter is not needed if the former is already there.

Also, EasySerializable.h, GerneralResults.h and RodCommand.h all include Serializable.h.

Command line tools (as in HostCommandpattern/tools) must include Client.h, either directly or indirectly through DefaultClient.h or SilentClient.h.

Host command class inheritance
(except as noted, all code in PPCpp/CmdPattern/)

Thick arrows: Actual inheritance **Thin arrows**: Inclusion via header, except as noted.
Labels with ".h" suffix indicate classes defined in header files only.

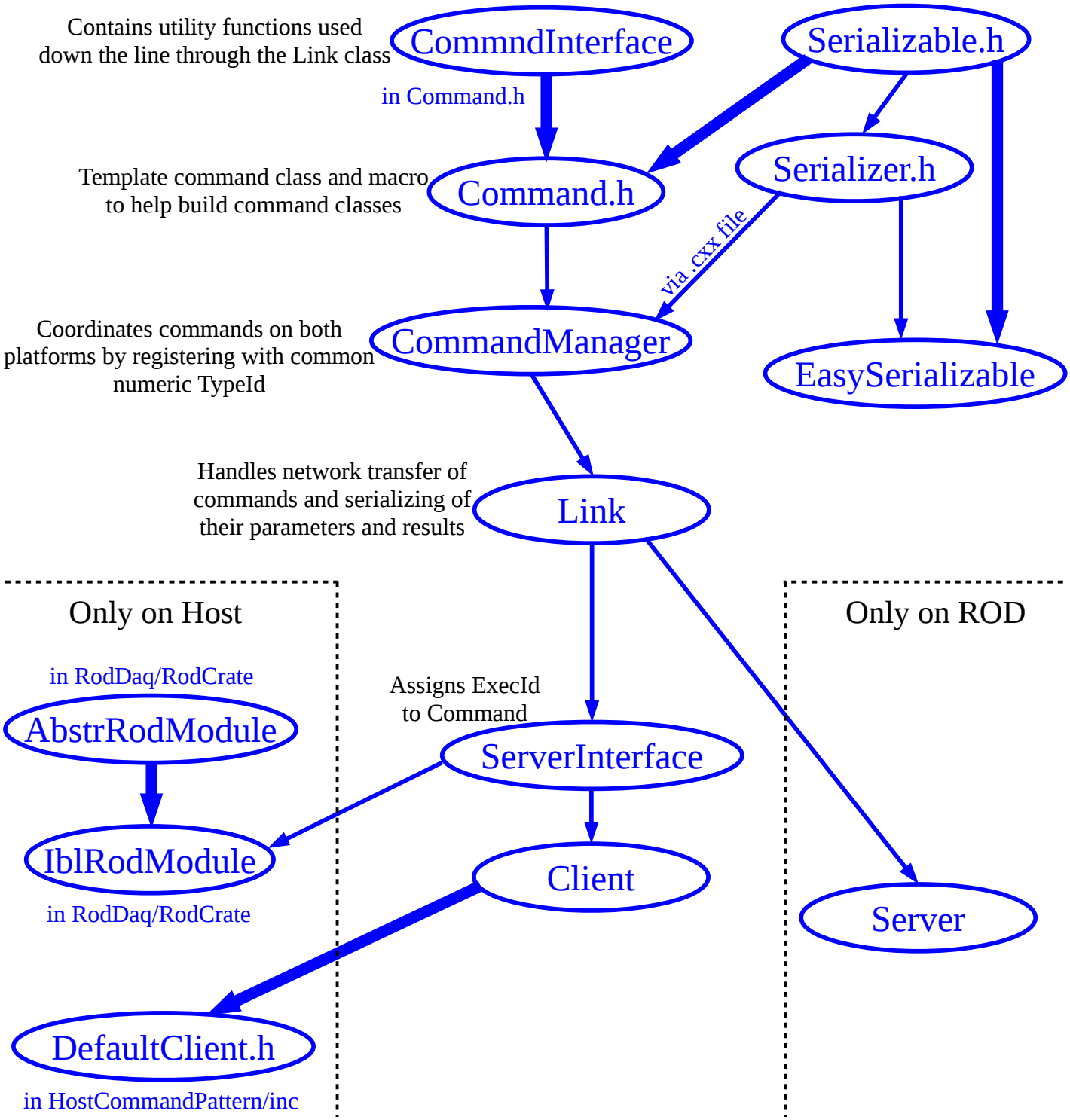


Figure 24

15.4 Command class: Serializing with the EasySerializable class

To illustrate how serialize/deserialize works, we once again trace the handling of just one scan parameter variable, namely the **PixScan** variable m_repetitions (the number of scan triggers). We already know how m_repetitions gets into the host command class object pixScnBaseToRod. The **Link** version of sendCommand runs

pixScanbaseToRod.serialize which (via template class **Serializer**<**SerializablePixScanBase**> as defined under "Specialization for classes" in Serializer.h, included in **EasySerializable**, inherited by **PixScanbaseToRod**) runs the function

serScnbase.serialize which is inherited from **EasySerializable** and calls serial() const, which calls

EasySerializable::prep(m_repetitions) const, which calls

Serializer<int>::serialize on the integer m_repetitions.

The deserializing on the ROD does the same things, except of course using instead of serialize, the **Serializer**<int>::deserialize function and the non-constant, instead of the const versions of the serial and prep functions.

The call to prep in the serScnBase serial function is built from the **EasySerializable** macro SERIAL. Identical calls to prep for all the other scan parameters are included in the same SERIAL macro call in SerializablePixScanBase.cxx (which, amazingly slick, contains absolutely no other code whatsoever), and whose use is nicely described in the comment at the top of EasySerializable.h. A simplified overview of all these interconnections is offered in figures 25a and 25b.

Major functions of **Serializable** Class
(code in PPCpp/CmdPattern/)

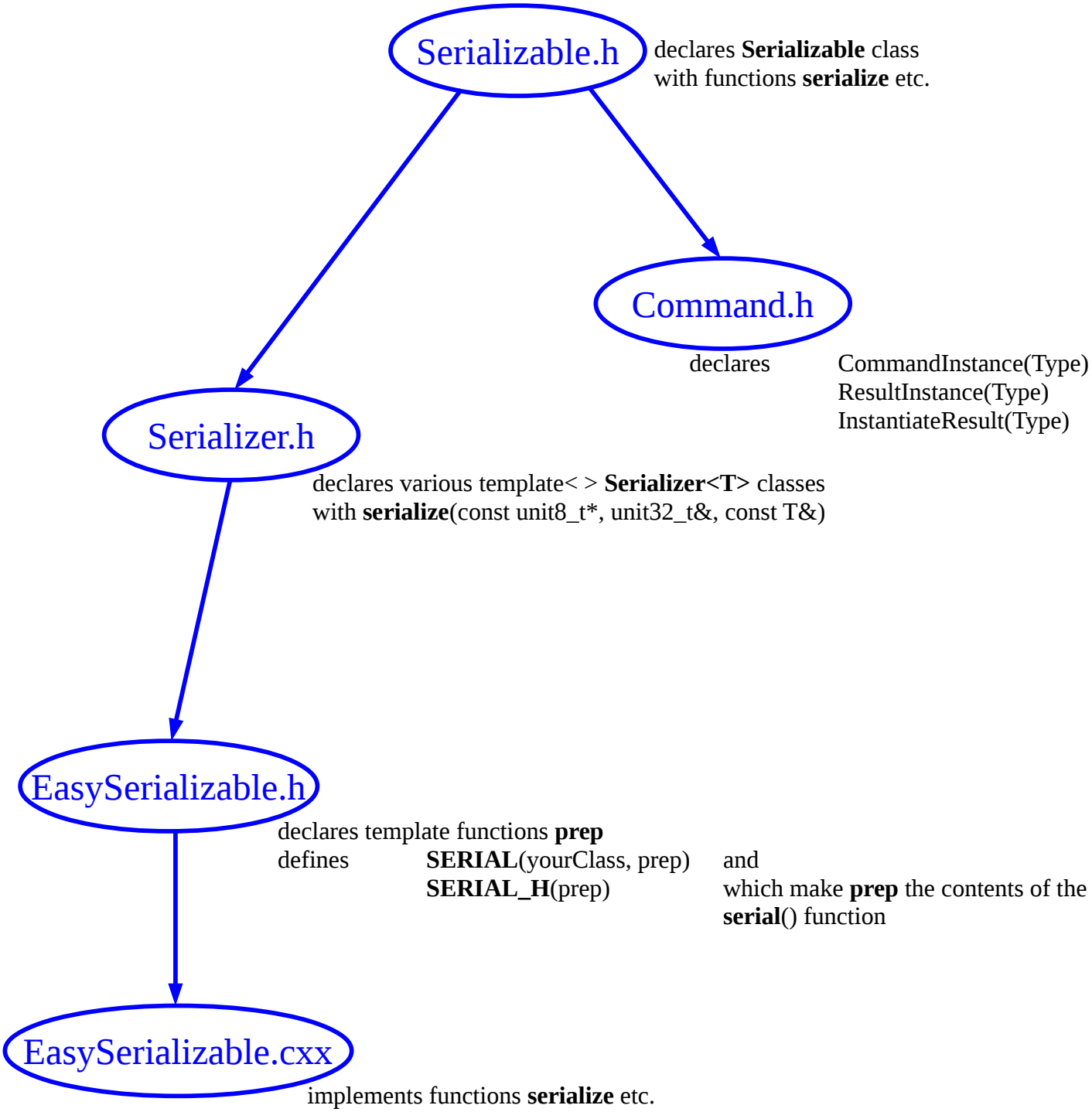


Figure 25a

Graphical Depiction of Principle Classes Used to Send Scan Configuration Parameters to ROD as Described in Sections 15.2 and 15.4

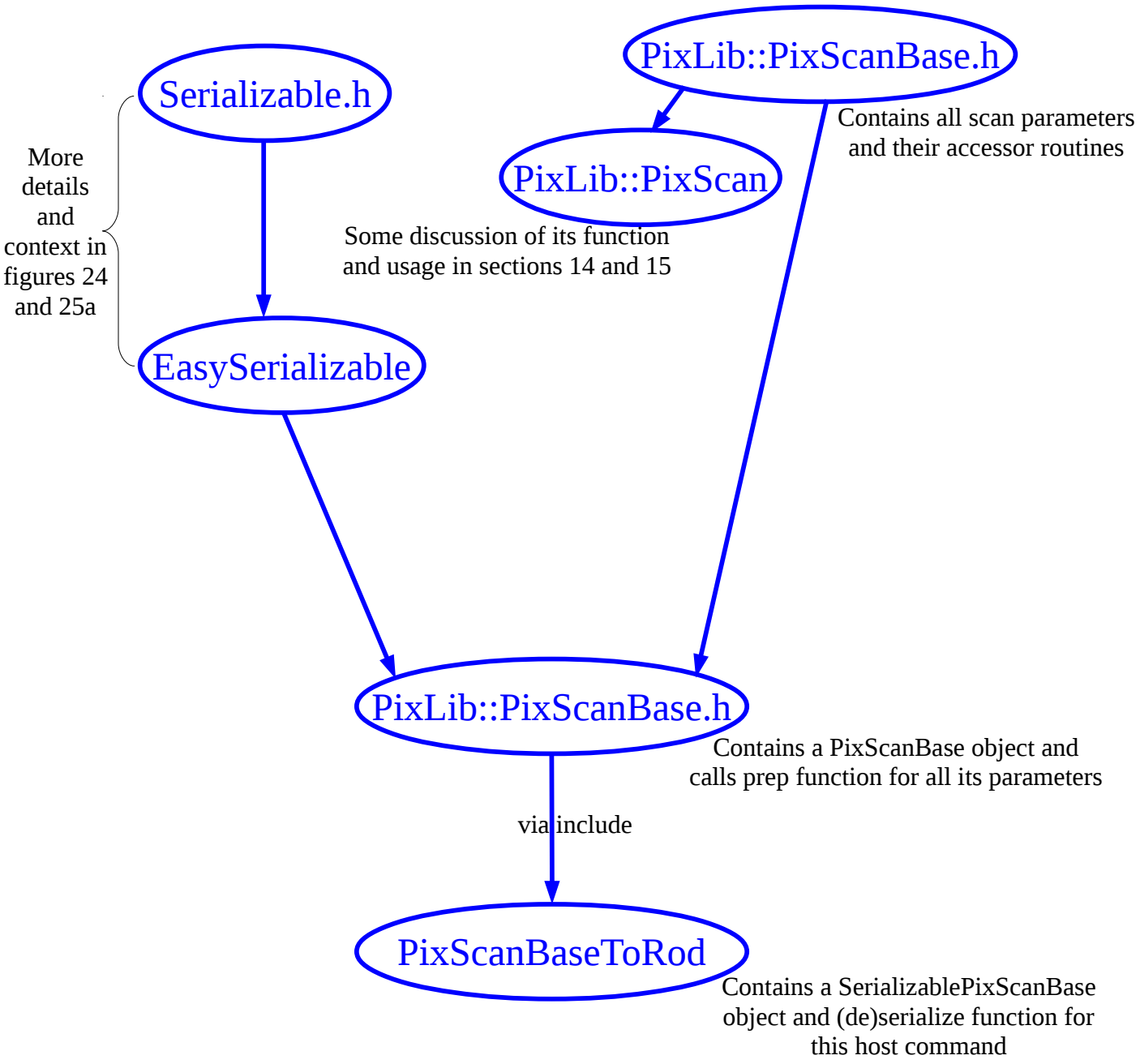


Figure 25b

16 Starting a scan running on ROD with BarrelRodPixController::startScan

The ultimate goal of **BarrelRodPixController**::startScan is to launch IblScanL12.cxx on the ROD. Some context is provided by figure 26 that highlights scan startup activity:

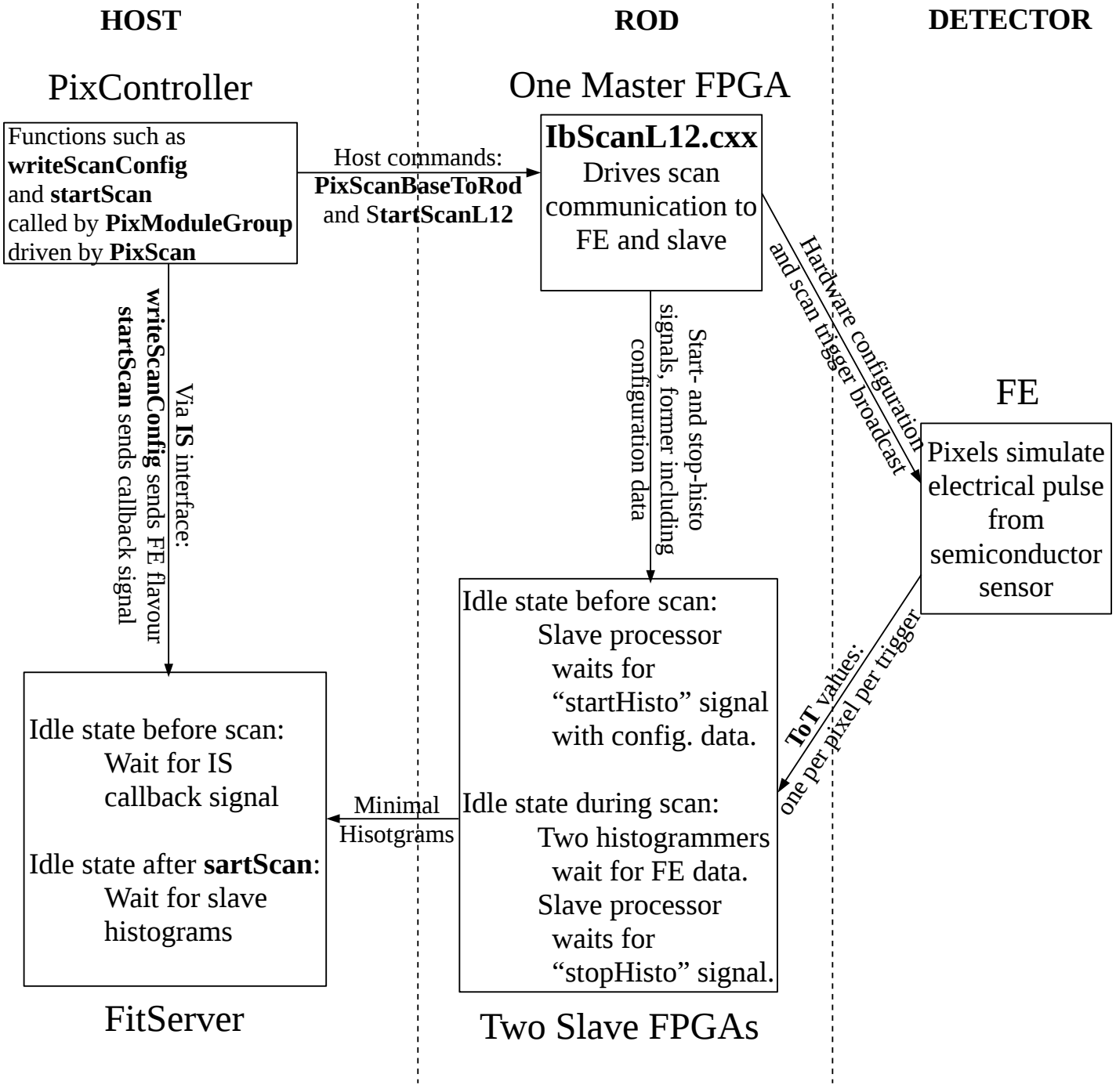


Figure 26

Most of what the ROD needs to know to run the scan has already been sent to it in **PixScanBase** by the host command **PixScanBaseToRod** in the function writeScanConfig. Now the function BarrelRodPixController::startScan must first alert the FitServer (via an IS callback mechanism) to begin waiting for the appropriate histograms from the slaves, then setup the scan to run on the ROD with the host command **StartScanL12** that launches **ScanBoss**::startScanL12. This latter function manages the entire scan process on the ROD through calls to the following three functions in a unique, global ("singleton") instance of the **IblScanL12** class (created in a **ScanBoss** object when the ROD software is first launched):

IblScanL12::initScan()	configures FEs for the scan.
IblScanL12::runScan()	alerts the slaves to start waiting for the (fake) "hit" data about to be generated in the FEs, then sends all the triggers needed to generate those fake hits, and finally tells the slave to stop waiting and ship all accumulated histogrammer results on to the FitServer.
IblScanL12::endScan()	ensures various variables monitored by host processes are set to correctly signal the scan status to them when the scan is done on the ROD.

While each "scan" at the ROD level is a single "level 0" loop step of the entire scan from the host perspective, there are actually a couple of further loop levels in **IblScanL12**::runScan().

The innermost of these loops is over the number of triggers in the already mentioned **PixScan** variable m_repetitions, which is usually between 50 and 200 for the simplest scans like DIGITAL and ANALOG. In each step of this loop, a single trigger command is broadcast to all enabled pixels (not masked out for the particular mask step of the scan) within all enabled front ends within all enabled modules attached to the ROD.

The outermost of the ROD-level loop is referred to as "mask stepping" which is intended to mask out all but the maximum number of pixels a front end can handle without overwhelming the electronics with too many simultaneous trigger signals.

Code has also been put in place for an intermediate loop level enabling only single pairs of FE columns per trigger broadcast (referred to as "double column" or "DC" looping), which is however mainly just useful (in fact necessary) for the FE-I4 chip where much fewer (8 vs. 32) mask steps are required, and therefore many more pixels per mask step enabled. For use in some early L1/L2 FE-I3 code-testing, an additional 4th ROD-level loop (between the mask stepping and DC loops) was created in IblScanL12.cxx that allows stepping through powers-of-2-sized groups of the 16 front ends per module. This last loop can currently only be enabled by changing the variable nFEgroup in the code from 1 to the desired power of 2 and then recompiling, in other words it remains a feature purely for testing purposes.

Except for mask stepping, all of these ROD-level loops impact no other aspect of the scan

beyond the interaction between front end and ROD. Mask stepping on the other hand must be handled by both slave and FitServer as well, and involves the following considerations:

From the slave's point of view each mask step is considered an entire “scan”, to be processed in its entirety before all results of that step are passed on to the FitServer, which expects each such result “package” to arrive in the order of the mask loop stepping. While the actual step number of any given mask step is not even included in the result package sent from the slave, it can be deduced by the Fitserver from the position of the mask step “scan” in the looping sequence.

17 Slave histogrammer function

The histogrammers built into the slave FPGA (there are two per slave, or a total of four per ROD) perform the minimal operations required to build the required statistics for each pixel in a scan. This is illustrated in figure 27. The three values per pixel, listed at the bottom of the diagram, are transmitted (to the FitServer in one long block from memory for all pixels) by the slave processor once it receives a signal from the master software that the scan is done (recall that a “scan” for the histogrammer, involves only the 1-out-of-32 pixels of a single mask step).

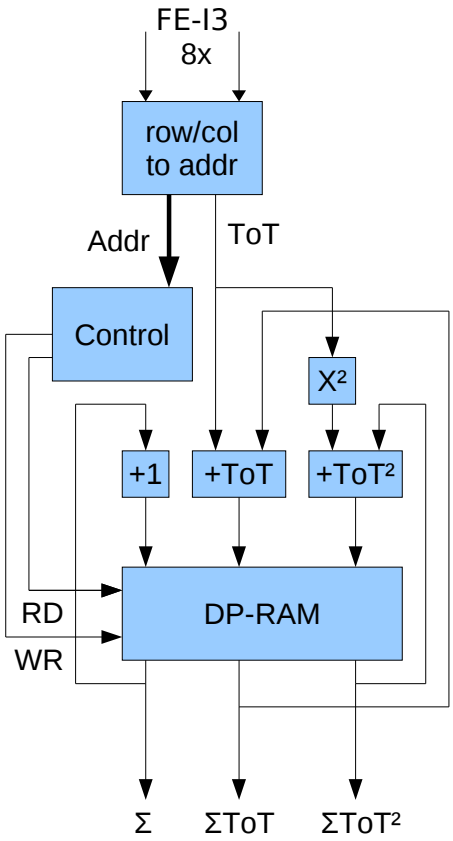


Figure 27 adapted from [19]

18 Slave histogrammer scan types

The histogrammer can be configured to not do the sums involving ToT if not needed in a scan, so only occupancy is calculated. For each of these two possibilities there is a compact and less compact result format. The alternatives are labeled as follows in the enum histoOpMode in RodDaq/IblDaq/common/DspInterface/PIX/iblSlaveCmds.h:

ONLINE_OCCUPANCY
OFFLINE_OCCUPANCY (compacted to four instead of one pixel per word in output format)
SHORT_TOT (compacted to one instead of two words per pixel in output format)
LONG_TOT

The compact forms OFFLINE_OCCUPANCY and SHORT_TOT have not been implemented in the L1/L2 ROD slave firmware and, while implemented in the IBL ROD slave firmware, are nowhere used in IBL software other than in some test programs. Because FE-I3 ToT is 8 instead of 4 bits, it is not possible to fit any ToT scan results into a single word anyway, so SHORT_TOT wouldn't even be useful for L1/L2. As of this writing, the L1/L2 slave firmware has a bug that prevents OFFLINE_OCCUPANCY from working even if it were desired. Here is a summary of the main features of the four output formats:

ONLINE_OCCUPANCY
24-bit occupancy, so results from a single pixel require an entire 32-bit word. It is possible to use this format for generating histograms with large occupancies during data taking, “online”.

OFFLINE_OCCUPANCY (bug in L1/L2 firmware, and apparently not used in IBL either)
8-bit occupancy, so results from four pixels fit in a single 32-bit word.

SHORT_TOT (not useful for L1/L2 , and apparently not used in IBL either)
ToT expected to be well above threshold, so occupancy count can be reduced to only the number of missed hits, a number that can fit into 4 instead of 8 bits, small enough to allow (for FE-I4 only) all three sums to fit into a single 32-bit word per pixel. In order to calculate the “missing” hits, the expected occupancy has to be supplied as an additional parameter to the histogrammer in a scan startup parameter.

LONG_TOT
Full 8-bit occupancy, together with 12-bit sum of ToT (for IBL, 16-bit for L1/L2) and 16-bit sum of squared ToT (for IBL, 24-bit for L1/L2) requires two 32-bit words per pixel.

19 FitServer thread and work-queue management

For each of the four histogrammers on each ROD assigned to a FitServer in the connectivity database, a separate **PixFitNet**::loop() thread is launched when the FitServer starts up. After that, the following three additional threads are started:
PixFitWorker::loop(), **PixFitAssembler**::loop() and **PixFitPublisher**::loop().

All thread loop() functions immediately go into an idle mode using the thread's getWork() function that does a wait for an object variable of type boost::condition_variable_any to have its notify_one() function unlock the thread with a call from the thread's addWork() function. These addWork()/getWork() functions add/remove “work requests” to/from the queues used to control the thread activities. How they interact can be deduced from the following table:

Class	getWork Queue	addWork Queue	addWork Queue type
PixController	--	RODqueue via IS	pair<string, string>
PixFitManager	RODqueue	m_scanConfigQueue*	PixFitScanConfig
PixFitNet	m_scanConfigQueue*	fitQueue	RawHisto
PixFitWorker	fitQueue	resultQueue	PixFitResult
PixFitAssember	resultQueue	publishQueue	PixFitResult
PixFitPublisher	publishQueue	--	--

All queue names are the ones given in **PixFitManager**, except for (*), which is the name in **PixFitNet**. For any particular scan, the flow of tasks performed by the FitServer can be traced in the table from top to bottom as “work requests” are added and removed from each queue:

PixController uses IS to initiate a callback function in **PixFitManager** so all the **PixFitNet** threads set up for those histogrammers connected to the modules in the scan can start polling their network port for incoming results.

Once the histogram data has arrived and been converted into a **RawHisto** class object form more compatible with the type of scan, the object is added to the **PixFitWorker** fitQueue.

Only threshold scans require real “work” by **PixFitWorker**, which just passes the other scans' fitQueue object unaltered into a new resultQueue object's **RawHisto** variable. In the threshold scan case, the lmfit function, in the **PixFitFitter_lmfit** class, does the necessary S-curve fitting and returns the required object to be placed in resultQueue.

PixFitAssembler transforms the results from the **RawHisto** form into root histograms that go into a publishQueue object's root histogram variables. Interestingly, both the assembler and

publisher queues are **PixFitResult** vectors.

PixFitPublisher handles the logistics of assigning the necessary ROD/Rx identifiers to the histograms before publishing them in OH.

20 Layout of pixel and FE geometry in modules

Since it is by no means obvious and in fact, without anything to use as a guide, this author made an incorrect guess the first time around, it is worth pointing out how pixels are labeled in front ends and how the 16 front ends are arranged and oriented on their module.

Pixels are indexed by their row and column placement on the front end, whereby row numbering increases from bottom to top and column numbering from left to right. The front ends' numbering increases from left to right on the bottom module row (oriented with its very first FE at bottom left), but increases from right to left on the top module row. In both top and bottom module rows the front ends are oriented with their “bottom” row on the module edge. In this way the front end numbering increases from left to right in both module rows as long as it is oriented so the row in question is at the bottom. This complicated layout is probably most easily captured by the following diagram (all numbering starting at **0**):

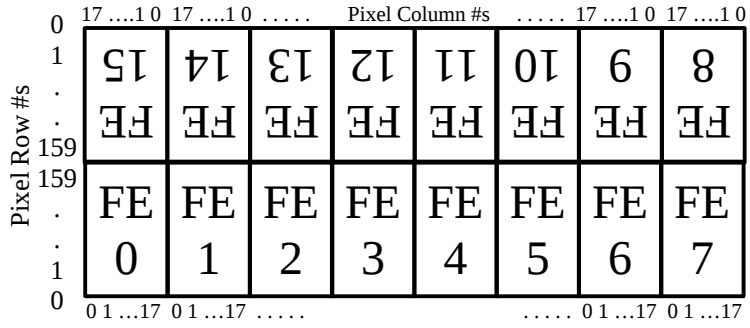


Figure 28

Perhaps even less clear at first sight than module pixel indexing is the way pixels are selected for each mask step. In the diagram below, the color for all pixels enabled in a given mask step is the same:

First 6 FE Column #s (starting at **1**)

First 96 FE Row #s (starting at **1**)

FE-I3	1	2	3	4	5	6	
96	32	1	32	1	32	1	Step 1
95	31	2	31	2	31	2	Step 2
94	30	3	30	3	30	3	Step 3
...	...	...	...	...	...	...	Step 4
68	4	29	4	29	4	29	...
67	3	30	3	30	3	30	Step 29
66	2	31	2	31	2	31	Step 30
65	1	32	1	32	1	32	Step 31
64	32	1	32	1	32	1	Step 32
63	31	2	31	2	31	2	
62	30	3	30	3	30	3	
...	...	...	...	...	...	...	
36	4	29	4	29	4	29	
35	3	30	3	30	3	30	
34	2	31	2	31	2	31	
33	1	32	1	32	1	32	
32	32	1	32	1	32	1	
31	31	2	31	2	31	2	
30	30	3	30	3	30	3	
...	...	...	...	...	...	...	
4	4	29	4	29	4	29	
3	3	30	3	30	3	30	
2	2	31	2	31	2	31	
1	1	32	1	32	1	32	

Figure 29

The differing pattern in odd and even columns is due to the way the serial bit mask identifying each of the 2880 pixels on a front end begins at the lower right corner (row=0, col=0), moves up the first column, then down the second, back up the third column and so forth with the last bit representing the top rightmost pixel (row=159, col=17). Each mask step simply enables every 32nd bit in this 2880-bit stream, beginning at bit 0 for mask step 0, and on through bit 31 for mask step 31 (numbering starting at **0**).

21 FE configuration sent to ROD with BarrelRodPixController::writeModuleConfig

21.1 Loading module configurations into host command class WriteConfig

Similar to the scan configuration parameters in **PixScan**, it is useful here to trace how a single front-end-level register setting, in this case DAC_VCAL (the analog scan VCal value that determines charge level), reaches the ROD from the connectivity database. This will be followed by a brief look at the more elaborate setting of pixel-level registers such as the 7-bit TDAC.

First a vector, with one **PixModule** class element for each module attached to the ROD, is filled with those modules' connectivity database values into the **Config** class object of that module's **PixModule** vector element, accessed by its config() function. The filling from the database is done by the function **PixModuleGroup**::initDbServer, called at the very beginning of each scan by the **PixModuleGroup** constructor, as presented in the scan startup sequence in section 14 above.

BarrelRodPixController::writeModuleConfig then sends the **PixModule Config** values to the ROD using the **Fei3ModCfg** host command. The point at which writeModuleConfig is called at the start of each scan is also shown in section 14 above. Here a broad overview of the sequence of activity in writeModuleConfig is presented:

In essence all that happens is that the parameters having earlier been retrieved from the connectivity database into the **PixModule** objects for each module are transferred to an **Fei3ModCfg** host command object that is then sent to the ROD, later to be actually transfered to the front ends in IblScanL12.cxx with the function **Fei3ModProxy**::sendPixelConfig.

After moving a few module-level parameters from **PixModule** to **Fei3ModCfg,** such as the eleven module register values, followed by some highly customized maneuvers to assign the correct communication channels to the module, front-end-level and pixel-level parameters are moved over, as illustrated here for the DAC_VCAL and TDAC settings respectively:

Front-end-level DAC_VCAL:

For each of the 16 **PixFe** objects, fe, in the **PixModule** object passed to writeModuleConfig, the DAC_VAL value (voltage level for analog scans used by all pixels on front end) is copied to the corresponding entry in the front end configuration array, m_chipCfgs, in the **Fei3ModCfg** object feMod with an assignment like this:

```
feMod.m_chipCfgs[(*fe)->number()].writeRegGlobal(GlobalRegisters::VCal,
                                                 (*fe)->readGlobRegister("DAC_VCAL"));
```

Pixel-level TDAC:

For multi-bit pixel registers such as TDAC, each bit is loaded one at a time into all pixels of a front end, all at once in the same serial "snaking" order (up first column, down second, etc.) as seen for the mask step pixel-select/enabling in the preceding section, 20. Thus we see that for the 7-bit TDAC, writeModuleConfig has to rearrange every single one of the 7 bits in the 2880 pixel TDAC values in the **PixFe** fe object of each front end in 3 levels of looping, roughly like this:

```
ConfMask<short unsigned int> &tdacReg = (*fe)->readTrim("TDAC");
for (unsigned col = 0; col < nCol; col++) {
  for (unsigned row = 0; row < nRow; row++) {
    for (unsigned ibit = 0; ibit < 7; ibit++) {
      feMod.m_chipCfgs[number].writeBitPixel(tdacBits[ibit], row, col,
                                             (tdacReg.get(col, row) >> ibit) & 0x1);
    }
  }
}
```

This fills in the array pixcfg[NPIXELREGISTERS * nDC * nWordsPerDC] in **Fei3PixelCfg** bit by bit for each of the seven pixel "bit-registers" TDAC0, TDAC1, . . . TDAC6. The seven other of the NPIXELREGISTERS=14 pixel "bit-registers" are:

HitBus, Select, Enable, FDAC0, FDAC1, FDAC2 and Preamp, where the 3-bit FDACs are of course handled the same as the TDACs, and "Select" and "Enable" will both get set to "1" for only those pixels in a scan's mask step (done while scans are running on the ROD as IblScanL12.cxx loops through the mask steps).

21.2 Sending Fei3ModCfg to ROD: A look at more complex serializing

Since the transfer of module configurations to the ROD, which is the last thing done by **BarrelRodPixController**::writeModuleConfig, involves much more than required for the scan parameters in **PixScanBase** with the host command **PixscanBaseToRod** as previously discussed, it is worth showing here how the host command serializing in **WriteConfig** works. It takes care of serializing values in numerous arrays and entire class objects within objects:

Link calls

WriteConfig::serialize(uint8_t* out) on the host.

Here "out" is a pointer to the buffer to contain the serialized **Fei3ModCfg** data on the host that will be sent over the network to the ROD.

The serialize function is inherited by **WriteConfig** from **EasySerializable** and calls the serial() function built by the macro SERIAL_H which is identical to SERIAL except that it does not prefix serial() with the class name space, and thus can be used within header-file class definitions.

serial() does
prep(configType);
prep(i_mod);
prep(**Fei3ModCfg** moduleConfig);

The first two prep functions work straightforwardly just as discussed for m_repetitions in **PixScanBaseToRod** above in section 15.4.

The third is a template function for classes with base **Serializable** which **WriteConfig** inherits from **EasySerializable**. It does

Serializer<**Fei3ModCfg**>serialize(out, offset, **Fei3ModCfg** moduleConfig) which does
moduleConfig.serialize(out+offset) which does
serial() which does
prep on all **Fei3ModCfg** simple (32-bit, or less) variables. It also does the array version of prep on the **Fei3Cfg** array m_chip_Cfgs:
prep(**Fei3Cfg** m_chipCfgs, nFE) which does
Serializer<Fei3Cfg>serialize(out, offset, m_chipCfgs[iFE]) for each of the 16 chips, iFE = 0, 1, . . . 15, which do
m_chipCfgs[iFE].serialize(out+offset) which does
serial() which does
Fei3ExtCfg::serial() which does
prep on all of the **Fei3ExtCfg** simple (32-bit, or less) variables, as well as
Fei3GlobalCfg::serial() which does
prep on the front end global register bit array variable.
Fei3PixelCfg::serial() which does
prep on the front end pixel register bit array variable.

Note that the last three calls to serial() are not through prep, as opposed to the preceding two, which are. This is because the objects the former are serializing are instantiated as part of the object that inherits them and from where the call to serial is made, whereas the latter two are not inherited, but are member objects instantiated independently from the serial calling object (**Fei3ModCfg** from **WriteConfig** and **Fei3Cfg** from **Fei3ModCfg)**.

When **Link** calls **WriteConfig**::deserialize(uint8_t* in) on the ROD, the exact same sequence of serial() function calls is made, except that the deserialize, instead of serialize functions are used, and “in” is the pointer to the serialized **Fei3ModCfg** data received over the network from the host to be deserialized on the ROD.

Fei3 Host Command Class Inheritance

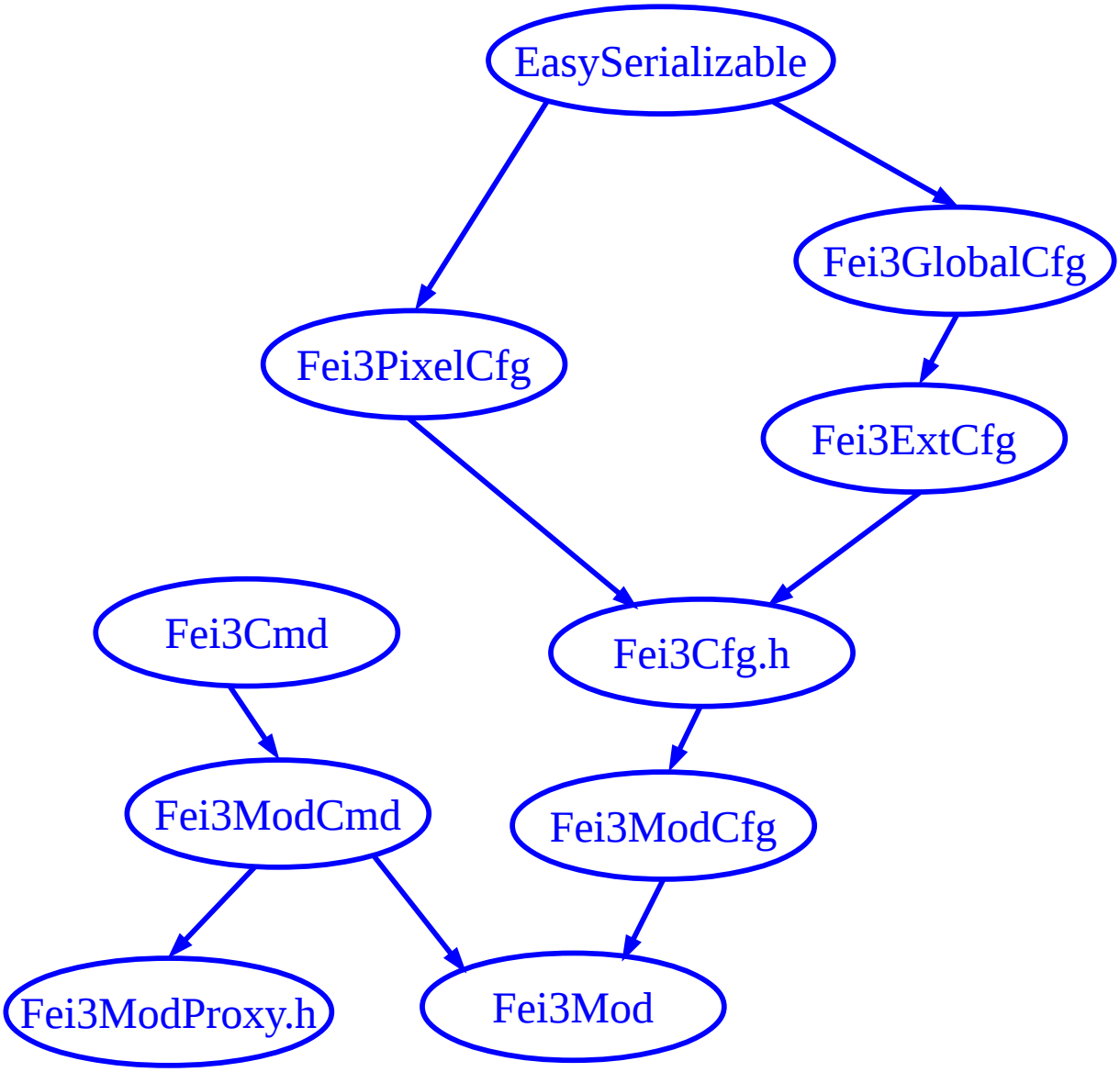


Figure 30

22 Extent of L1/L2 code divergence from IBL

One of the advantages of having used a new L1/L2 ROD design almost identical to IBL is the ability to reuse large amounts the IBL software. Even where code cannot be literally shared, at least the possibility of reusing many basic design principles exists.

The biggest divergence occurs in areas of greatest differences in hardware, which in our case involves the front ends. That is why we now do, and probably always will have completely separate class structures for FE-I3 and FE-I4, most notably in the ROD software "FrontEnd" folder.

On the other hand, the upgrade sees barely any changes in either slave or FitServer histogramming software requirements, so only slight generalizations in a few places of the existing IBL code was all that was necessary to make the same code base work for L1/L2 as well. This circumstance demonstrates the significant advantage of having kept histogramming as simple and generic as possible through the FitServer stage (and therefore require the controller to remake the FitServer OH histograms for console display). Even fewer changes were needed to the highest level, most generic host classes PixScan and PixModuleGroup.

An intermediate position between these two extremes is taken by the controller and master ROD software since both require more direct interaction with detector hardware specifics. Here some scope exists for improved redesign to allow more sharing of code for common activities. This has not been done due both to a desire to initially limit changes of existing IBL code already proven to work, along with the limited availability of coding skills to achieve the necessary, not entirely trivial redesign. Therefore BarrelRodPixController.cxx and PPCppRodPixController.cxx on the host and IblScanL12.cxx and IblScan.cxx on the ROD remain completely independent from each other for now, in spite of containing much code in common. In fact at the level of detail presented in this document, essentially everything said about the L1/L2 versions of the two applies equally to the other.

23 Special software developed to simulate scans in absence of actual hardware

Besides the very isolated, standalone software packages (labeled “0” and “0.5” in figure 31) already in existence when I began the L1/L2 calibration upgrade, it was not possible to test most new features without having every single other software and hardware component perfectly functioning. Nothing would work if even one of the many obscure little pieces was not available. To get around this, I developed software to test each of the progressively more complete data paths indicated in figure 31. In addition to allowing more productive debugging of the parts I was working on, the tests made possible by these new software components gave much more robust confirmation of the functionality of the targeted parts.

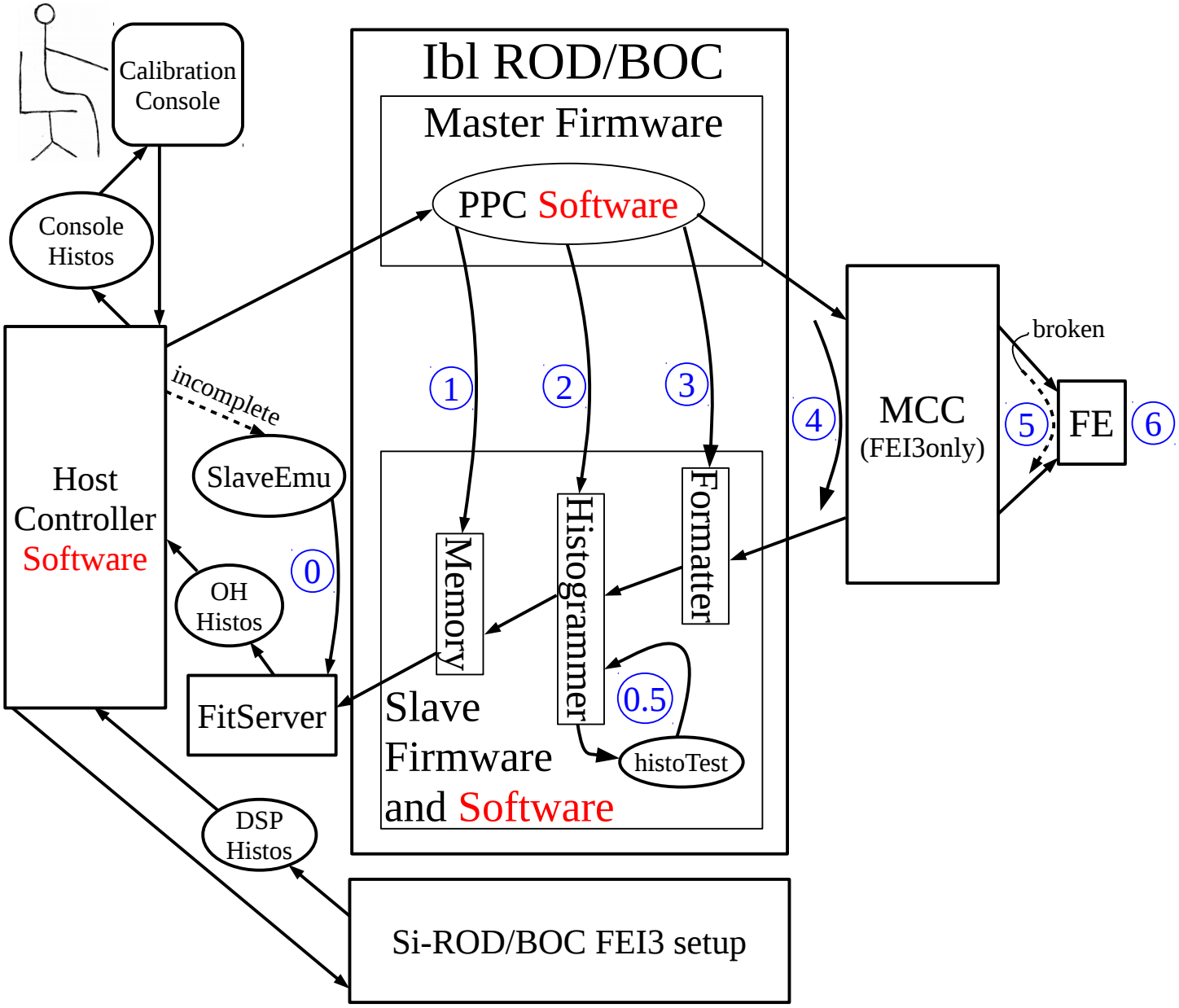


Figure 31

New Software to Validate Scan Software - and some Hardware too!

Level	Component	Key Software	Key Control Parameters (#define / enum used by IblScan(L12).cxx)	Run from CL = Host Cmd. Line All run in Console = GUI
0*	FitServer only	slaveEmu	all in file slaveEmu.cfg	CL: slaveEmu GUI: incomplete
1	Slave Memory	IblRodSlave:: fillDummyHisto	enum TestHistoStatus = HISTO_TEST_SETUP_NOHISTO	CL: dummyScan
2	Slave Histogrammer	IblRodSlave:: fillDummyHisto or IblRodSlave:: testRxBypass	enum TestHistoStatus = HISTO_TEST_SETUP or mode = 2 in -->	CL: dummyScan or testRxHisto --mode 2
3**	Slave Formatter	IblRodSlave:: fillDummyHisto	enum TestHistoStatus = HISTO_TEST_SETUP_EXT	CL: dummyScan
4	MCC (no FEI4 equivalent)	MCCTestScan	#define __MCC_TEST__ 1 __USE_FEI3__ 0	CL: testRxHisto --mode 1
5***	FEI3 (FEI4 equivalent missing)	Fei3TestScan	#define __MCC_TEST__ 1 __USE_FEI3__ 0	CL: testRxHisto --mode 0
6	Detector Tuning	to many pieces to list !!!	all in console	GUI only

* 0 still requires work to launch from GUI. Command line version works.

** 3 is implemented in software, but requires firmware fix.

** 5 worked for a while, but changes in low-level code have broken it.

1 - 5 first require #define changes in IblScan(L12).cxx and recompiling before they can be run from GUI. All but 6 can run from command line.

24 Conclusion: For good or bad, powerful custom software the only way to run ATLAS

One might be left wondering whether the maze of software presented here is really appropriate for the basic scanning and tuning tasks required by the ATLAS pixel detector. The purpose of this paper has been mainly to present **what is there now** in a compact and orderly enough manner so a careful reader will be able to avoid struggling as much as the author did to uncover some of the broader patterns and functionality within the key components of calibration software.

Since **what is there now** suffers from the inevitable chaos inherent in any large work in progress like the ATLAS experiment, a certain level of such disorder will permeate even the best form of presentation. If that remains a failing in some areas of this document, these concluding summary remarks can hopefully, where not actually able to tie together all the missing threads, at least explain the underlying difficulties. After taking a look at some areas that, in my view were (perhaps made unnecessarily) difficult, in conclusion a few places will be highlighted that ultimately hold up very well to scrutiny, in spite of being at first glance sometimes extremely hard to penetrate due to a lack of almost any kind of documentation.

The interdependence of hardware and software in cutting edge experimental work is hard to overestimate. The software to control and operate the hardware will not be optimal when adequate hardware to test it on is not available. On the other hand, without optimal software, the hardware cannot be operated to its fullest capacity, if at all. This wasn't always possible for the project discussed here. However, given the tight hardware-software interdependency, it is hoped that the emphasis in this paper on certain software intricacies offer better insight for a reasonable understanding of hardware operation, at least for scanning and tuning purposes.

Another difficulty encountered in upgrading existing software was that occasionally some of it turned out to be barely suitable for the task it was meant to do. What follows is a description of some of these challenges, and where they led to during the project described in this paper.

As outlined in section 23, the unavailability of some, but not all DAQ readout chain hardware still allowed for the creation of specialized software leading to confirmation of the expected behavior of those components that actually were operational. Taking advantage of this meant that once front end pixels were fully accessible to developers in the SR1 testing facility, only little extra work was needed to verify the functionality of the complete readout chain.

Unfortunately, not enough modules to connect RODs to their full capacity was ever available in SR1 before the detector itself needed to start operating with the upgraded layer 2 hardware in 2016. It turned out that in spite of successful tests on the limited SR1 setup, there was trouble operating the upgraded scanning software on RODs at their full capacity of 26 modules. Because of this, layer 2 was not tuned for the 2016 runs, and instead ran with the last tuning points from the previous year, before the DAQ upgrade. Together with the minor

changes needed for the 2017 layer 1 upgrade, fixing the problems in full detector scans will be the major focus of the concluding work on this project in 2017.

As already mentioned, some of the pre-existing software had been completed in great haste, and was left as it was in a state that just barely got what was needed done. The upgrade was a good opportunity to overcome such deficiencies, permitting the occasional bug-fix along the way. In this way, the IBL system was also improved.

Bigger software challenges were questions of how suitable overall designs were for the required purposes. In this regard, a significant obstacle was the proliferation of abandoned code, whose authors were long gone and whose relevance was almost impossible to confirm by those still around. Such "orphaned" code can cause huge distractions in dealing with code that is still relevant, and should become a future cleanup project for the sake of efficiency, at least in human understanding, if not always actual operation.

Such distractions aside, there are a number of core areas of the software with unquestionably robust design. In conclusion, the three most relevant to calibration touched on in this paper are summarized here:

I. Handling thousands of configuration parameters

Complicated device configuration parameter structures are effectively built using the highly versatile **Config** class structure in the **PixLib** function library. Quickly stored and retrievable from databases, these are good for handling everything from tracking the maze of hardware component connectivity (e.g. which modules connect to which RODs), to the pixel-level register parameters, the vast (over 90 million) number of which is compensated by their relatively small (8-bit maximum) numerical values. The basics of this design can be appreciated from figure 23 and the associated discussion in sections 15.1 and 21.1.

II. Interaction between distinct processor platforms providing minimal user hassles

Streamlined servicing is provided for commands requiring interaction between vastly differing processor types (Linux hosts vs. PPC) involving thousands of parameters and returned result values. Sections 15.2, 15.3 and 15.4, their figures 24 and 25, as well as especially section 21.2 give a sense of the many parameters and structures containing them that can be efficiently sent around with the host command serializing features. As briefly mentioned in section 13.1 the command line tool HostCommandPattern/tools/compareCommands.cxx offers an illustrative, simple example of how command results produced by the ROD are returned to the host.

III. Hardware processing on dozens of RODs converted to multi-threaded processing

FitServers replace dozens of histogramming and curve fitting processes previously running simultaneously on separate old Si-RODs. Linux host PCs do this well with sophisticated thread and work-queue management techniques, as is briefly outlined in section 19.

Appendix A - Glossary

Arduino Due	Italian designed sensor board (Arduino for Italian King from 1002 to 1014, Due for Duemilamonove)
ASIC	Application Specific Integrated Circuit
ATLAS	A Toroidal Lhc AparatuS
BBIM	Building Block Interlock Monitoring (Logical signals out of BBM to Interlock System)
BBM	Building Block Monitoring (Crates: Temperature and Humidity Monitoring)
BC	Bunch Crossing
BCR	Event Counter Reset
BOC	Back of Crate Card
CAN	Control Area Network (630 nodes of open protocol communication via ELMB interface)
CORBA	Common Object Request Broker Architecture
DAL	Data Access Library (in TDAQ software)
DAQ	Data AcQuisition system
DAVE	Digital ATLAS Vme Electronics
DCS	Detector Control System
DDC	DAQ-DCS Communication
DFM	Data Flow Manager
DIM	Distributed Information Management system
DORIC	Digital Optical Receiver Integrated Circuit)
DSP	Digital Signal Processing
ECR	Event Counter Reset
ELMB	Embedded Logical Monitoring Board (ATLAS standard front end unit for slow-control signals)
FIT	Front-end Integration Tools (software for DCS experts)
FPGA	Field Programmer Grid Array
FSM	Finite State Machine
IOM	Input-Output Manager
IPC	Inter Process Communication
IS	Information Service (implemented using IPC)
LTP	Local Trigger Processor
MAC	Media Access Controller
MCC	Module Control Chip
MUX	Multiplexer
OKS	Library to support simple, active, persistent in-memory object manager class definitions and instances stored in xml files (why “OKS” ?)
OPC	Openness, Productivity Collaboration (Servers that integrate hardware into higher level software)
PP0	Patch Panel 0
PRM	Program Reset Manger (FPGA)
RCD	ROD Crate DAQ
ROBIN	ReadOut Buffer INput
ROD	ReadOut Driver (9 crates with up to 16 RODs per crate)
ROI	Region Of Interest
ROL	ReadOut Links
ROS	ReadOut System (Atlas common design)
SCT	SemiConductor Tracker
SEU	Single Event Upset
SIT	System Integration Tools (software for shift workers, i.e. a detector-oriented view)
TDAQ	Trigger and Data AcQuisition system
TIM	TTC Interface Module
TRT	Transition Radiation Tracker
TTC	Trigger Timing and Control
VCSEL	Vertical Cavity Surface Expansion Layer
VDC	VCSEL Driver Chip

Appendix B - Scan configuration parameter names

Part 1

Groups in **Config** object “ScanConfig” of some concern to IblScanL12.cxx

ScanConfig

name (if applicable) in PixFeStructures::getGlobReg	FE Global Register name	ScanConfig ConfGroup	ConfObj / root Variable	PixScanPanel Var.	PixScanBase Variable	PixScanBase function/s	Used in IblScanL12 ?
ENABLE_CINJ_HIGH	HighInjCapSel	fe	chargeInjCapHigh	capLowHigh	m_chargeInjCapHigh	s/getChargeInjCapHigh	yes
ENABLE_DIGITAL	EnableDigInj	fe	digitalInjection	injectionMode	m_digitalInjection	s/getDigitalInjection	yes
1	HitBusEnable	fe	hitbus	HitbusEnabled	m_feHitbus	s/getFeHitbus	no
		fe	m_columnROFreq	PhiClock	m_columnROFreq	s/getColumnROFreq	yes
THRESH_TOT_DOUBLE	THRDUB	fe	m_totDHThr	dblTOT	m_totDHThr	s/getTotDHThr	yes
THRESH_TOT_MINIMUM	THRMIN	fe	m_totMin	minTOT	m_totMin	s/getTotMin	yes
MODE_TOT_THRESH	ReadMode	fe	totThrMode	TOTmode	m_totThrMode	s/getTotThrMode	yes
ENABLE_TIMESTAMP	TSI_TSC_Enable	fe	totTimeStampMode	TOTLEmode	m_totTimeStampMode	s/getTotTimeStampMode	yes
DAC_VCAL	VCal	fe	Vcal	setVal	m_feVCal	s/getFeVCal	yes
		general	Boc_Scan_DSP_nHits		m_bocScanNHits	s/getBocScanNHits	no
		general	Boc_Scan_DSP_pattern		m_bocScanPattern	s/getBocScanPattern	no
		general	bocScanDecodeMode		m_bocScanDecodeMode	s/getBocScanDecodeMode	no
		general	cloneCfgTag		m_cloneCfgTag	getCloneCfgTag	no
		general	cloneModCfgTag		m_cloneModCfgTag	getCloneModCfgTag	no
		general	maskStageMode	maskMode	m_maskStageMode	s/getMaskStageMode	yes
		general	maskStageSteps	nMaskSteps	m_maskStageSteps	s/getMaskStageSteps	no
		general	maskStageTotalSteps	maskStage	m_maskStageTotalSteps	s/getMaskStageTotalSteps	yes
		general	modConfig		m_modConfig	s/getModConfig	no
		general	modScanConcurrent	concurrMode	m_modScanConcurrent	s/getModScanConcurrent	no
		general	patternSeeds		m_patternSeeds	s/getPatternSeeds	no
		general	repetitions	nEvents	m_repetitions	s/getRepetitions	yes
		general	restoreModuleConfig	restorConfig	m_restoreModuleConfig	s/getRestoreModuleConfig	yes
		general	runType	runType	m_runType	s/getRunType	no
		general	saveModGroupConfig		m_saveModGroupConfig	s/getSaveModGroupConfig	no
		general	saveModulesConfig		m_saveModulesConfig	s/getSaveModulesConfig	no
		general	scanFEbyFE	sngFeMode	m_FEbyFE	s/getFEbyFE	no
		general	scanFEbyFEMask		m_FEbyFEMask	s/getFEbyFEMask	no
		general	scanType		m_scanTypeEnum	s/getScanTypeEnum	no
		general	swapInLinks		m_swapInLinks	s/getSwapInLinks	no
		general	swapOutLinks		m_swapOutLinks	s/getSwapOutLinks	no
		general	tagSuffix		m_tagSuffix	s/getTagSuffix	no
		general	tuningStartsFromConfigValues		m_tuningStartsFromConfigValues	tuningStartsFromCfgValues	no
		general	TxLaserCurrentBase		m_TxLaserCurrentBase	s/getTxLaserCurrentBase	no
		general	TxLaserCurrentProbe		m_TxLaserCurrentProbe	s/getTxLaserCurrentProbe	no
		general	useAltModLinks		m_useAltModLinks	s/getUseAltModLinks	no
		trigger	consecutiveLevl1TrigA_0	nAccepts_A0	m_consecutiveLvl1TrigA[0]	s/getConsecutiveLvl1TrigA	yes
		trigger	consecutiveLevl1TrigA_1	nAccepts_A1	m_consecutiveLvl1TrigA[1]	s/getConsecutiveLvl1TrigA	no
		trigger	consecutiveLevl1TrigB_0	nAccepts_B0	m_consecutiveLvl1TrigB[0]	s/getConsecutiveLvl1TrigB	no
		trigger	consecutiveLevl1TrigB_1	nAccepts_B1	m_consecutiveLvl1TrigB[1]	s/getConsecutiveLvl1TrigB	no
		trigger	lvl1HistoBinned	lvl1Binned	m_lvl1HistoBinned	s/getLvl1HistoBinned	no
		trigger	LVL1Latency	trgLatency	m_LVL1Latency	s/getLVL1Latency	yes
		trigger	self	selfTrigger	m_selfTrigger	s/getSelfTrigger	yes
		trigger	strobeDuration	strDuration	m_strobeDuration	s/getStrobeDuration	yes
		trigger	strobeLVL1Delay	l1Delay	m_strobeLVL1Delay	s/getStrobeLVL1Delay	yes
		trigger	strobeLVL1DelayOverride	overwL1delBox	m_strobeLVL1DelayOverride	s/getStrobeLVL1DelayOverride	yes
		trigger	strobeMCCDelay	strDelay	m_strobeMCCDelay	s/getStrobeMCCDelay	yes
		trigger	strobeMCCDelayRange	strDelRange	m_strobeMCCDelayRange	s/getStrobeMCCDelayRange	yes
		trigger	superGroupTrigDelay	superGrpDelay	m_superGroupTrigDelay	s/getSuperGroupTrigDelay	yes
		trigger	trigABDelay_0	delayAB0	m_trigABDelay[0]	s/getTrigABDelay	no
		trigger	trigABDelay_1	delayAB1	m_trigABDelay[1]	s/getTrigABDelay	no
		scans	thresholdTargetValue		m_thresholdTargetValue	s/getThresholdTargetValue	no
		scans	totTargetCharg		m_totTargetCharge	s/getTotTargetCharge	no
		scans	totTargetValue		m_totTargetValue	s/getTotTargetValue	no

Scan configuration parameter names (cont.)

Part 2

Use of FE Global Registers in IblScanL12.cxx

FEglobalReg

name (if applicable) in PixFeStructures::getGlobReg	FE Global Register name	ScanConfig ConfGroup	ConfObj / root Variable	PixScanPanel Var.	PixScanBase Variable	PixScanBase function/s	Used in IblScanL12 ?
0	SelfTriggerDelay						
0	EnableHitParity						
0	SelectDataPhase						
0	EnableEOEParity						
0	HitBusScaler						
0	EnableARegMeas						
0	ARegMeas						
0	EnableAReg						
0	EnableLVDSReferenceMeas						
0	EnableDRegMeas						
0	DRegMeas						
0	EnableTune						
0	EnableIPMonitor						
1	ARegTrim						
1	DRegTrim						
1	HitBusEnable	fe	hitbus	HitbusEnabled	m_feHitbus	s/getFeHitbus	
1	EnableBiasCompensation						
CAP_MEASURE	CapMeasure						
DAC_ID	ID						
DAC_IF	IF						
DAC_IL	IL						
DAC_IL2	IL2						
DAC_IP	IP						
DAC_IP2	IP2						
DAC_ITH1	ITH1						
DAC_ITH2	ITH2						
DAC_ITRIMIF	ITrimIf						
DAC_ITRIMTH	ITrimTh						
DAC_IVDD2	IVDD2						
DAC_VCAL	VCal	fe	Vcal	setVal	m_feVCal	s/getFeVCal	yes
ENABLE_BUFFER	EnableAnalogOut						
ENABLE_BUFFER_BOOST	EnableBufferBoost						
ENABLE_CAP_TEST	EnableCapTest						
ENABLE_CINJ_HIGH	HighInjCapSel	fe	chargeInjCapHigh	capLowHigh	m_chargeInjCapHigh	s/getChargeInjCapHigh	yes
ENABLE_CP0	EnableCol0						
ENABLE_CP1	EnableCol1						
ENABLE_CP2	EnableCol2						
ENABLE_CP3	EnableCol3						
ENABLE_CP4	EnableCol4						
ENABLE_CP5	EnableCol5						
ENABLE_CP6	EnableCol6						
ENABLE_CP7	EnableCol7						
ENABLE_CP8	EnableCol8						
ENABLE_DIGITAL	EnableDiginj	fe	digitalInjection	injectionMode	m_digitalInjection	s/getDigitalInjection	yes
ENABLE_EXTERNAL	EnableExtInj						
ENABLE_LEAK_MEASURE	EnableLeakMeas						
ENABLE_SELF_TRIGGER	EnableSelfTrigger						
ENABLE_TEST_ANALOG_REF	TestAnalogRef						
ENABLE_TIMESTAMP	TSI TSC Enable	fe	totTimeStampMode	TOTLEmode	m_totTimeStampMode	s/getTotTimeStampMode	no
ENABLE_VCAL_MEASURE	EnableVCalMeas						
FREQUENCY_CEU	CEU_Clock						
GLOBAL_DAC	GlobalTDAC						
LATENCY	Latency						
MODE_TOT_THRESH	ReadMode	fe	totThrMode	TOTmode	m_totThrMode	s/getTotThrMode	yes
MON_ID	TestDAC_ID						
MON_IF	TestDAC_IF						
MON_IL	TestDAC_IL						
MON_IL2	TestDAC_IL2						
MON_IP	TestDAC_IP						
MON_IP2	TestDAC_IP2						
MON_ITH1	TestDAC_ITH1						
MON_ITH2	TestDAC_ITH2						
MON_ITRIMIF	TestDAC_ITrimIf						
MON_ITRIMTH	TestDAC_ITrimTh						
MON_IVDD2	TestDAC_IVDD2						
MON_MON_LEAK_ADC	MonLeakADC						
MON_VCAL	TestDAC_VCal						
MUX_DO	Select_DO						
MUX_EOC	EOC_MUX						
MUX_MON_HIT	Select_MonHit						
MUX_TEST_PIXEL	TestPixelMUX						
THRESH_TOT_DOUBLE	THRDUB	fe	m_totDHThr	dblTOT	m_totDHThr	s/getTotDHThr	yes
THRESH_TOT_MINIMUM	THRMIN	fe	m_totMin	minTOT	m_totMin	s/getTotMin	yes
WIDTH_SELF_TRIGGER	SelfTriggerWidth						
		fe	m_columnROFreq	PhiClock	m_columnROFreq	s/getColumnROFreq	no

References

1. https://commons.wikimedia.org/w/index.php?curid=679693

2. The ATLAS Collaboration, “ATLAS Insertable B-Layer Technical Design Report”, CERN-LHCC-2010-013, https://cds.cern.ch/record/1291633/files/ATLAS-TDR-019.pdf.

3. ATLAS Collaboration, The ATLAS Experiment at the CERN Large Hadron Collider, Institute Of Physics Publishing And SISSA (2008)

4. ATLAS Collaboration, ATLAS pixel detector electronics and sensors, Institute Of Physics Publishing And SISSA (2008)

5. https://indico.cern.ch/event/83060/contributions/2101686/attachments/1069923/1525755/FE-I4.pdf

6. The FE-I4 Collaboration, The FE-I4B Integrated Circuit Guide, manual, version 2.3 (Dec. 30, 2012)

7. Polini A. et al., Design of the ATLAS IBL Readout System, Physics Procedia 37 (2012) 1948-1955

8. George, M., Implementation of an Electrical Read-Out System for Multi-Module Laboratory Tests of the ATLAS Pixel Detector, PhD Thesis, Georg-August-Universität Göttingen (2009)

9. Flick, T., L1/L2 Review Introduction & HW Status, Report for Upgrade Project Review Part 1, CERN, 29.-30. September 2015

10. Jedrzej B., The Implementation and Performance of ROD DSP Software in the ATLAS Pixel Detector , ATL-INDET-INT-2010-006, 2010

11. https://indico.cern.ch/event/372043/contributions/882248/attachments/1180439/1708663/NickDreyer2015-11-02Report.pdf

12. https://indico.cern.ch/event/447711/contributions/1112754/attachments/1159899/1674591/Sr1_Status_Update.pdf

13. Weingarten, J., System Test and Noise Performance Studies at the ATLAS Pixel Detector, PhD Thesis, Universität Bonn (2007)

14. Garelli, N. et al., The Tuning and Calibration of the Charge Measurement of the Pixel Detector, ATL-COM-INDET-2010-017, 2009

15. Flick, T., Studies on the Optical Readout for the ATLAS Pixel Detector, PhD Thesis, Bergische Universität Wuppertal (2006)

16. IBL-WG4, Calibration of the ATLAS IBL detector using the new VME readout architecture, project document, Rev. No. 0.2 (2013)

17. Behara, P. et al., Threshold Tuning of the ATLAS Pixel Detector, ATL-COM-INDET-2010-001, 2010

18. https://gitlab.cern.ch/atlas-pixel/atlaspixeldaq/tree/sr1/PixelDAQ-02-04-00

19. The ATLAS Collaboration, “IBL ROD BOC Manual” (development draft version), https://espace.cern.ch/atlas-ibl/OffDecWG/_layouts/WopiFrame.aspx?sourcedoc=/atlas-ibl/OffDecWG/Shared Documents/Readout System/IBL ROD/iblRodBocManual_v1.2.1.odt