1 Introduction

This note is intended to describe the design and implementation of the pixel online software. The pixel online software is used for controlling and calibrating the CMS pixel detector. In particular, it performs the following functions:

- Configure the detector
- Perform online calibrations
- Analyze calibration data in online farm (CMSSW)
- Monitor the detector during data taking

In Section 3 the main software components are introduced. Next the software organization is discussed. In Section 4 the different software components are discussed in more detail. In Section 13 the different calibrations are described. Other information about pixel online software is available at the pixel online software wiki pages [1].

2 Pixel DAQ System

The intention of this section it to give an overview of the CMS pixel DAQ system. For more details the reader can follow the references given in the text. But the introduction given here should be sufficient to understand the main goals of the pixel software.

2.1 Overview of DAQ components

The CMS pixel DAQ system consists of a number of components as illustrated in Fig. 1. This figure is specific to the FPix detector, but the modification for the BPix are minor and not really relevant to the discussion here unless otherwise mentioned. Starting from the detector itself we have the Read Out Chip (ROC) [2] and the token bit manager (TBM) [3]. Some properties of the ROC are described below as needed to understand the calibrations. The ROC reads out 4,160 pixels and the TBM coordinates the communication with a group of 8 to 24 ROCs. The TBM is electrically connected to the portcard via extension cables ≈ 2 feet in length.

The portcard receives and sends optical signals to the FrontEnd Controller (FEC) and FrontEnd Driver (FED) respectively. The FEC is used to program the TBM and ROCs and the data read out from the detector is send to the FED to be digitized. The portcard is the home of several discrete components. We have the Digital Optic Hybrid (DOH) that receives data from the FEC and the Analog Optic Hybrid (AOH) that transmits data to the FED. On the portcard we have in addition the (t)PLL, Delay25, and gatekeeper chips.

Of particular interest is the Delay25 chip. The communication to the portcard via the DOH is done at 40MHz on a serial line. We have a clock line and the data line (and return clock and return data). In order to have this communication working the timing between

the clock and data lines must be right. The purpose of the Delay25 chip is to adjust delays to make this communication work. The setting of the parameters on the Delay25 chip as well as the other components on the portcard is done using the I2C protocol from the CCU. The CCU is again controlled from a Tracker FEC (TkFEC).

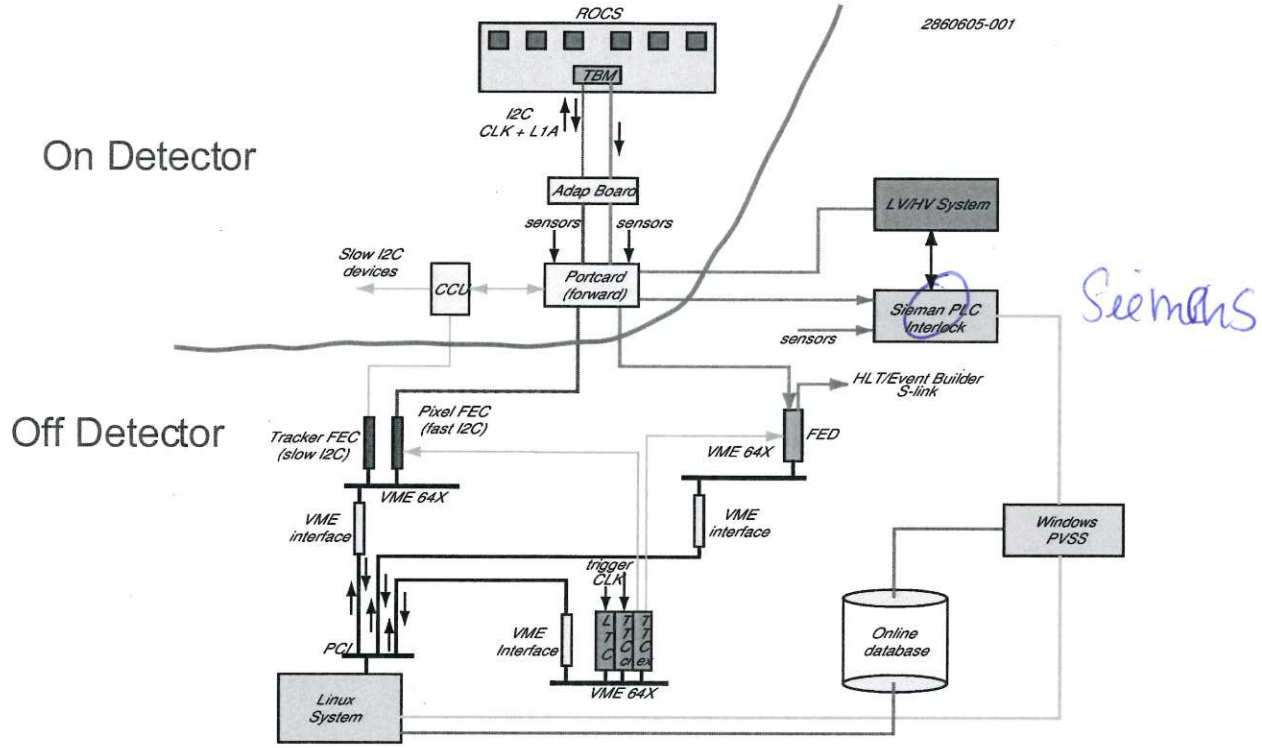


Figure 1: The main components in the CMS pixel DAQ system.

2.1.1 The Read Out Chip

Each Read Out Chip (ROC) [2], ilustrated in Fig. 2, collates data from $52 \times 80 = 4{,}160$ pixels and contains about 1.3 million transistors ~~in all~~. It amplifies and zero suppresses data using 4 trim bits that specify a threshold for every pixel. The chip buffers hit data until the trigger decision arrives. It is developed at PSI and manufactured by IBM.

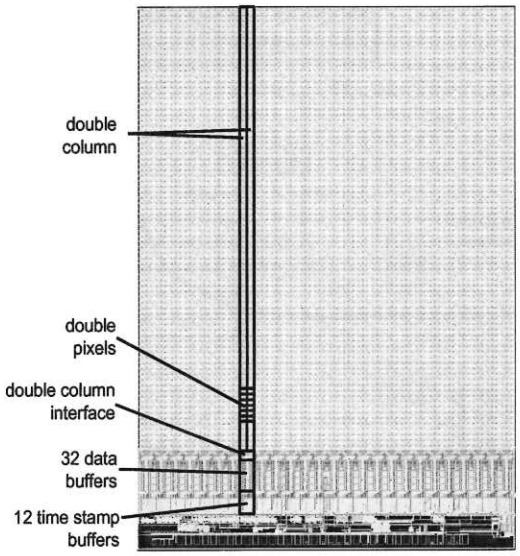


Figure 2: The Read Out Chip of the CMS Pixel Detector.

Table 1: Pixel online PCs at P5

Slot in rack	Node name	Front	Label	Size	Function	Comments	OS
-	vmepcs2b18-17	Pixel	-	?U	Histoviewer		SLC4
19	vmepcs2b18-16	Pixel	1/10	1U	VME/S1G01i	BPix FECs	SLC4
17	vmepcs2b18-15	Pixel	10/10	1U	VME/S1G01e	FPix FECs	SLC4
16	vmepcs2b18-14	Pixel	9/10	1U	VME/S1G04e	BPix FEDs do	SLC4
14/15	vmepcs2b18-13	Pixel	8/10	2U	VME/S1G04i	BPix FEDs up	SLC4
13	vmepcs2b18-12	Pixel	7/10	1U	VME/S1G03i	FPix FEDs	SLC4
12	vmepcs2b18-11	Pixel	6/10	1U	PixelSuperFPix	FPix Supv.	SLC4
11	vmepcs2b18-10	Pixel	5/10	1U	PixelSuperBPix	BPix Supv.	SLC4
10	vmepcs2b18-09	Pixel	4/10	1U	DCS	Fpix daq	Win
9	vmepcs2b18-08	Pixel	3/10	1U	DCS/Siemens		Win
8	vmepcs2b18-07	Pixel	2/10	1U	DCS/CAEN		Win
	vmepcs2b16-10				TTC+LTC Supv.	TTC+LTC	SLC4
	cmsrc-pixel				L1 FM	Pixel FM	SLC4
	fmmpc-s1d12-08				FMM PC	FMM PC	SLC4
	cmspsx				PSX server	PSX server	SLC4
	srv-c2c02-05				DB server		SLC4
	srv-c2c02-06				DB server		SLC4

2.2 Installation at P5

Table 1 lists the online PCs that we have at P5 for pixel online software.

3 Online Software Overview

The pixel online software is based on the xdaq toolkit and is built from a number of different components (applications). The different xdaq based components are shown in green in Fig. 6. The top level application is the PixelSupervisor. This application is responsible for the overall coordination of the pixel DAQ. The PixelSupervisor talks to the supervisors that directly control the hardware. For example we have the PixelFECSupervisor that provides the interface to the pixel FECs. Similarly the PixelFEDSupervisor controls FEDs. In production at P5, there are multiple instances of the PixelFECSupervisor and PixelFEDSupervisor; one per VME crate.[1]

The PixelTKFECSupervisor controls the tracker FEC hardware. The pixel system uses the tracker FEC hardware slow I2C to initialize the fast I2C used for the download of most configuration data.

The PixelTTCSupervisor controls the pixel TTC module used for trigger and timing. Among other things the TTC module is used during calibrations to generate triggers. In modern releases of the software, the PixelTTCSupervisor has been deprecated in favor of the TTCciControl, which is a standard package maintained by the TTC group.

[1]The strip tracker uses a design where there is one supervisor per VME board.

refer to these tables in the text

Table 2: FED connections for FPIX

Rack	Crate	Slot	VME addr.	Channels	FED id	Name(Official)	Name(Construction)
S1G03	upper	6	0x13000000	13-24	33	BpO_D(1,2)_BLD(1,2,3)	HC+Z1 1.1 2.1
S1G03	upper	6	0x13000000	1-12	33	BpO_D(1,2)_BLD(4,5,6)	HC+Z1 1.2 2.2
S1G03	upper	7	0x14000000	13-24	34	Bp0_D(1,2)_BLD(7,8,9)	HC+Z1 1.3 2.3
S1G03	upper	7	0x14000000	1-12	34	Bp0_D(1,2)_BLD(10,11,12)	HC+Z1 1.4 2.4
S1G03	upper	5	0x12000000	1-12	32	BpI_D(1,2)_BLD(1,2,3)	HC+Z2 1.4 2.4
S1G03	upper	5	0x12000000	13-24	32	BpI_D(1,2)_BLD(4,5,6)	HC+Z2 1.3 2.3
S1G03	upper	8	0x15000000	1-12	35	BpI_D(1,2)_BLD(7,8,9)	HC+Z2 1.2 2.2
S1G03	upper	8	0x15000000	13-24	35	BpI_D(1,2)_BLD(10,11,12)	HC+Z2 1.1 2.1
S1G03	upper	11	0x17000000	13-24	37	BmI_D(1,2)_BLD(1,2,3)	HC-Z1 1.1 2.1
S1G03	upper	11	0x17000000	1-12	37	BmI_D(1,2)_BLD(4,5,6)	HC-Z1 1.2 2.2
S1G03	upper	12	0x18000000	13-24	38	BmI_D(1,2)_BLD(7,8,9)	HC-Z1 1.3 2.3
S1G03	upper	12	0x18000000	1-12	38	BmI_D(1,2)_BLD(10,11,12)	HC-Z1 1.4 2.4
S1G03	upper	10	0x16000000	1-12	36	BmO_D(1,2)_BLD(1,2,3)	HC-Z2 1.4 2.4
S1G03	upper	10	0x16000000	13-24	36	BmO_D(1,2)_BLD(4,5,6)	HC-Z2 1.3 2.3
S1G03	upper	13	0x19000000	1-12	39	BmO_D(1,2)_BLD(7,8,9)	HC-Z2 1.2 2.2
S1G03	upper	13	0x19000000	13-24	39	BmO_D(1,2)_BLD(10,11,12)	HC-Z2 1.1 2.1

Table 3: FEC connections for FPIX

Rack	Crate	Slot	VME addr.	mFEC	Name(Official)	Name(Construction)
S1G01	middle	5	0x28000000	3	BpO_D(1,2)_BLD(1,2,3)	HC+Z1 1.1 2.1
S1G01	middle	5	0x28000000	4	BpO_D(1,2)_BLD(4,5,6)	HC+Z1 1.2 2.2
S1G01	middle	5	0x28000000	5	Bp0_D(1,2)_BLD(7,8,9)	HC+Z1 1.3 2.3
S1G01	middle	5	0x28000000	6	Bp0_D(1,2)_BLD(10,11,12)	HC+Z1 1.4 2.4
S1G01	middle	5	0x28000000	8	BpI_D(1,2)_BLD(1,2,3)	HC+Z2 1.4 2.4
S1G01	middle	5	0x28000000	7	BpI_D(1,2)_BLD(4,5,6)	HC+Z2 1.3 2.3
S1G01	middle	5	0x28000000	2	BpI_D(1,2)_BLD(7,8,9)	HC+Z2 1.2 2.2
S1G01	middle	5	0x28000000	1	BpI_D(1,2)_BLD(10,11,12)	HC+Z2 1.1 2.1
S1G01	middle	10	0x50000000	3	BmI_D(1,2)_BLD(1,2,3)	HC-Z1 1.1 2.1
S1G01	middle	10	0x50000000	4	BmI_D(1,2)_BLD(4,5,6)	HC-Z1 1.2 2.2
S1G01	middle	10	0x50000000	5	BmI_D(1,2)_BLD(7,8,9)	HC-Z1 1.3 2.3
S1G01	middle	10	0x50000000	6	BmI_D(1,2)_BLD(10,11,12)	HC-Z1 1.4 2.4
S1G01	middle	10	0x50000000	8	BmO_D(1,2)_BLD(1,2,3)	HC-Z2 1.4 2.4
S1G01	middle	10	0x50000000	7	BmO_D(1,2)_BLD(4,5,6)	HC-Z2 1.3 2.3
S1G01	middle	10	0x50000000	2	BmO_D(1,2)_BLD(7,8,9)	HC-Z2 1.2 2.2
S1G01	middle	10	0x50000000	1	BmO_D(1,2)_BLD(10,11,12)	HC-Z2 1.1 2.1

Table 4: CCU connections for FPIX

Rack	Crate	Slot	mFEC	Name(Official)	Name(Construction)
S1G01	middle	18	8	BpI	HC+Z2
S1G01	middle	18	7	BpO	HC+Z1
S1G01	middle	18	6	Bm0	HC-Z2
S1G01	middle	18	5	BmI	HC-Z1

- PixelConfigDBInterface
- PixelDCSInterface
- PixelFECInterface
- PixelFECSupervisor
- PixelFEDInterface
- PixelFEDSupervisor
- PixelFunctionManager
- PixelLTCSupervisor
- PixelSupervisor
- PixelTKFECSupervisor
- PixelTTCSupervisor[2]
- PixelUtilities

The package dependency tree is shown in Fig 7. The supervisor applications are at the top and depend on the packages below. We should make sure that the dependencies form a tree and not contain loops. (If it seems necessary to create a loop the solution is almost always to separate out some piece of code into a separate package.)

4.1 Pixel Function Manager

The pixel function manager (the Level 1 Function Manager) acts as an interface between run control (the level 0 function manager) and the pixel online software. The pixel function manager is a java application. It implements the state machine of CMS [4]. The function manager interacts with the PixelSupervisor to carry out the different tasks needed in state transitions of the run control.

4.1.1 FSM Implementation Status

The CMS run control finite state machine (FSM) is shown in Fig. 8. This model is implemented in the the pixel function manager, and also in the PixelSupervisor. The other Supervisors implement portions of this model as required. At the moment, the FSM depicted in the figure is completely implemented in the PixelSupervisor, with the exception of the "Reset" transition and its accompanying "Resetting" state. The PixelSupervisor does include transitions from any state into the "Error" state, and then allows a "Recover" transition that returns the FSM to the "Halted" state.

[2]In use through tag POS_3_1_2; deprecated starting in POS_3_2_0.

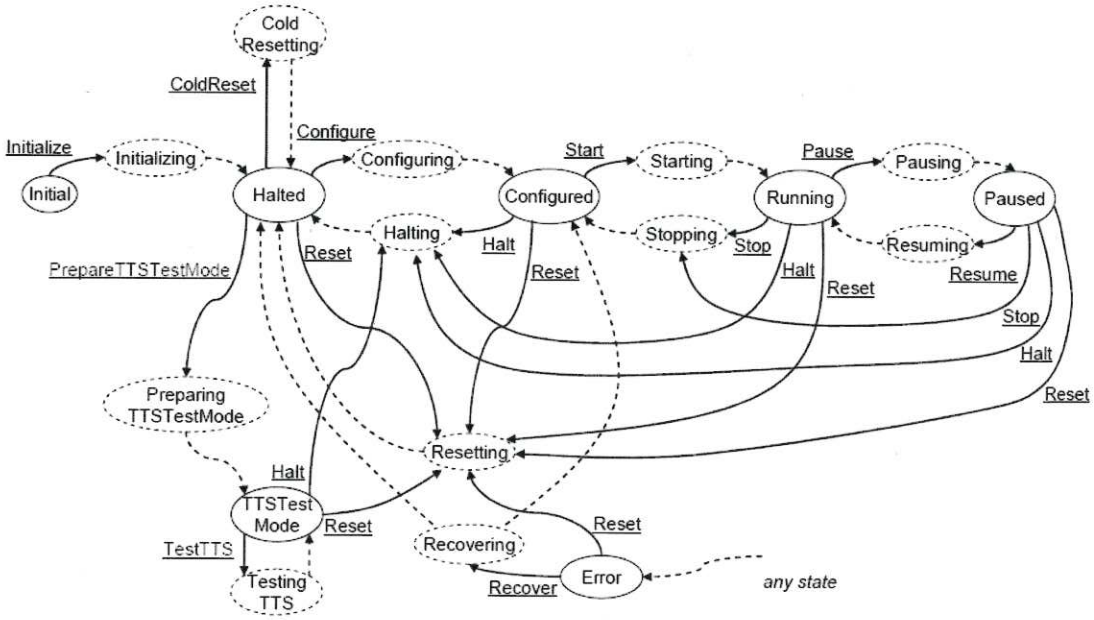


Figure 8: The CMS finite state machine definition. Figure taken from Ref. [4].

4.1.2 Control of the L1FM

During global running, the L1FM is driven by the L0FM, which is operated by central DAQ. The pixel user should not intervene via the L1FM GUI, except to check its status.

During local running, the L1FM must be created via its GUI. State transitions of the FSM can then be driven via the L1FM GUI. In general, we drive the "Initialize" state transition via the GUI, then drive subsequent state transitions directly from the PixelSupervisor. However, in principle all transitions can be initiated from the L1FM GUI.[3]

4.1.3 Outline of L1FM implementation

The L1FM is created using the Create button in RCMS. This calls the method of `PixelFunctionManager.ja` called `createAction()`.

After creation, transitions of the FM FSM are handled by methods in `PixelEventHandler.java`. Which method is triggered depends on the transition, as defined by a table near the top of this class. The names are fairly logical (for example, "Initialize" corresponds to the `initAction` method and "Configure" corresponds to the `configureAction` method). Inside each `fooAction` method are two blocks: one to handle objects of type `StateEnteredEvent` and one to handle objects of type `StateNotification`.

[3] In practice, this is only useful for simple tests of the L1FM to PixelSupervisor communication.

When an FSM transition "foo" is initiated by the L0FM or L1FM GUI, we enter the `StateEnteredEvent` block of `fooAction`. This block then contains the code to send a message to PixelSupervisor, telling it to proceed with transition "foo". The L1FM then does nothing while PixelSupervisor coordinates the necessary activities in POS. When the transition is completed by the PixelSupervisor, it passes a message back to the L1FM.[4] This message triggers entry into the `StateNotification` block of `fooAction`. If PixelSupervisor reports that the transition was successful, this block moves the L1FM FSM into the state "foo". If PixelSupervisor reports an error, this block moves the L1FM FSM into state "Error". In this way, the L0FM (central DAQ) learns whether the transition was successful.

4.2 PixelSupervisor

The PixelSupervisor is the top level xdaq application in the pixel online software. As described above it takes commands from the function manager. There is one pixel supervisor for the pixel online system.

4.2.1 Functions

The main function of the PixelSupervisor is to coordinate the activities of the other supervisors, particularly during configuration (see Sec. 10) and calibration (see Sec. 13). It is responsible for updating the configuration database with new settings obtained by calibrations.

The PixelSupervisor also communicates the state of the pixel xdaq software (POS) to the Level 1 Function Manager (Sec. 4.1).

4.2.2 Interface

The PixelSupervisor web GUI is an html page, which by default refreshes every few seconds. It displays information about the current configuration, or if it is not configured it allows the user to select a possible configuration from a list and configure the detector using that configuration.

The PixelSupervisor runs the JobControl Monitor, which is a utility that periodically sends "heartbeat" SOAP messages to the JobControl processes running on the various machines at P5. The PixelSupervisor GUI uses the replies from these SOAP messages to display whether any of the POS xdaq processes has crashed, or whether any of the JobControl processes themselves are unresponsive. (Note that we typically only run JobControl at P5, so this feature is not available elsewhere.)

[4]This is implemented in PixelSupervisor using the `stateChanged` method of the `rcmsStateNotifier_` object.

4.3 PixelFECSupervisor

The FEC Supervisor controls the pixel FECs. This means it is responsible for loading the configuration parameters for the ROCs from the configuration database and programming those parameters into the detector.

4.4 PixelFEDSupervisor

The FED Supervisor controls and monitors the pixel FEDs.

5 Coding practices

5.1 Makefile

The pixel online software is built with a Makefile located in each (sub)package. The Makefile builds on the xdaq tools. This file should ideally be as short as possible and use the functionallity in the xdaq package. You invoke the Makefile in each package by doing a 'make'. There is also a `clean` target.

In addition to the package level makefiles this is a toplevel makefile in the pixel directory. This makefile allow you to build all the xdaq packages using `make Set=pixel`. You can also clean all packages using `make Set=pixel clean`.

5.2 Include files

Include statements should include the file path starting from the project. For example you should do

```
#include "PixelCalibrations/include/PixelAOHBiasCalibration.h"
#include "CalibFormats/SiPixelObjects/interface/PixelCalibConfiguration.h"
```

5.3 CVS tags

We create 'official' tags of the form 'POS_X_Y_Z', e.g. 'POS_2_4_5'. For every tag created, starting with POS_2_5_0, an entry should be made in the file pixel/README that describes briefly the new features in the tag.

5.4 Building RPMs

The building of RPMs should be straightforward. The following steps are required.

- Prepare the code, update the README file, and VERSION file and commit and tag the code. The VERSION file contains the version of the RPM to build.
- Invoke 'make Set=pixel rpm' to build RPMs for all pixel online software packages.

- In the `PixelUtilities` directory, invoke 'buildExternalRPMs.sh' to build RPMs for DiagSystem, TTCSoftware, and FecSoftwareV3_0. At the end of building the external RPMs this script copies all RPMs to $BUILD_HOME/RPM_X.Y.Z-V, where X, Y, Z, and V are the major version, minor version, patch, and build version, respectively.

The set of RPMs built can be tested (in the online environment) for consistency by invoking the command 'rpm –test -Uvh *.rpm' in the directly where all the RPMs are located.

Old RPMs can be cleaned out of the build area using the command 'make Set=pixel cleanrpm'.

6 Configuration Data Management

The next two sections describe a C++ interface for configuration data management and the different classes that are used to store this information.

The idea behind this organization should be explained here. For now I just add a pointer to a presentation I gave at a pixel software meeting that contains the ideas:

`http://indico.cern.ch/conferenceDisplay.py?confId=8768` Link works, but better to write a few sentences

7 Configuration Database Interface

Since early summer 2006, we have used an interface for accessing configuration data in the online software framework. The access was originally file based, and has now (starting in 2009) been supplemented with infrastructure to access an Oracle database. The interface used is rather generic and the purpose of this document is to write down the interface so that we can have a clear separation between the database code and the applications that use data from the configuration database.

First in Sect. 7.1 we describe the C++ API. The interface is defined in the class `PixelConfigInterface`. This interface is fully implemented in the `PixelConfigFile` implementation based on files.

In Sect. 7.2 a simple command line tool is described. This tool is implemented through the C++ interface and should work both for the file based and the final data base implementation.

Examples below use a few of the configuration data classes that are used.

7.1 C++ Configuration Data Access API

The class `PixelConfigInterface` in the package `PixelConfigDBInterface` defines the interface for access of configuration data. This interface is intended to provide type safe access methods for retrieving and storing configuration data.

7.1.1 Retrieving data from the database

The primary method for retrieving the data is

```
template <class T>
static void get(T* &pixelObject,
std::string path,
pos::PixelConfigKey key)
```

The `PixelConfigKey` is just an integer that holds the top level configuration key to be used. This interface returns a pointer to the data. The caller is assumed to take ownership of the data and delete it. The path is a 'secondary key'. It would allow us to store more than one object of the same type in a given configuration. (In the file based implementation

this label is used to build the path to were the file is stored.) Below are a few example of using this interface.

```
PixelConfigKey theGlobalKey(5);

PixelNameTranslation *theNameTranslation=0;
PixelConfigInterface::get(theNameTranslation, "nametranslation/",
                    theGlobalKey);

PixelDetectorConfig *theDetectorConfiguration=0;
PixelConfigInterface::get(theDetectorConfiguration, "detconfig/",
                    theGlobalKey);

PixelFECConfig *theFECConfiguration=0;
PixelConfigInterface::get(theFECConfiguration, "fecconfig/",
                    theGlobalKey);
```

These examples show how you extract objects for which there is only one instance of for the whole detector configuration. You pass in as the first argument a pointer. The pointer will return 0 if the object was not successfully retrieved.[5] The second argument is the label for the object. This is essentially a key that is used to look up the data. The interface is type safe, i.e., if you specify a path to an object of the wrong type you will get back a null pointer. The third argument is the global configuration key. This basically specifies the versions of all objects used in a given configuration. This key is implemented as an integer.

Besides objects like the name translation and detector configuration listed above, there are objects such as trim bits, mask bits, and dac values that we need to access on a finer granularity than for the whole detector. To do this we use slightly modified arguments

```
PixelDACSettings *tempDACs=0;
PixelConfigInterface::get(tempDACs,''pixel/dac/FPix_BpI_D1_BLD1_PNL1'',
                    theGlobalKey);
```

where we have added to the path the module name for which we want to extract the dac settings.

As a given application, for example the PixelFECSupervisor, will need to access dac settings for many modules, and it is more efficient to extract the data 'in bulk' from the database, we have also added an interface that allows extraction of multiple objects at the time

[5]We have now tried to improve on this error-handling scheme by throwing a `std::exception` in case there is a failure to retrieve the configuration information. When we attempt to retrieve configuration data in the POS software, we both test for a null pointer and handle any exceptions thrown.

7.1.4 Alias manipulation

For configurations aliases you can retrieve the list of defined aliases using

```
static std::vector<std::pair<std::string, unsigned int> > getAliases()
```

The string is the name of the alias and the unsigned int is the corresponding key. *I think it should actually be a PixelConfigKey.*

A very similar method is

```
static std::map<std::string, unsigned int> getAliases_map()
```

That returns a map between the alias name and the corresponding key.

The method

```
static void addAlias(std::string alias, unsigned int key)
```

inserts a new alias. Note that if the alias already exists it will just be updated to point to the new key.

The method below is a lower level method. This method allows you to define an alias to a version that has key already created and link it to version aliases. This method should eventually be removed as it is probably to error prone...

```
static void addAlias(std::string alias,
                     unsigned int key,
     std::vector<std::pair<std::string, std::string> > versionaliases)
```

To create an alias for a configuration object use the method.

```
static void addVersionAlias(std::string path,
                            unsigned int version,
                            std::string alias)
```

To add an alias for a configuration object, as supposed to a configuration key, use

```
static void addVersionAlias(std::string path,
                           unsigned int version,
                            std::string alias)
```

To get the actual version that an alias points to use the method

```
static unsigned int getVersion(std::string path,
                               std::string alias)
```

7.1.5 Support of polymorphism

The interface should support polymorphism in the sense described below. For example, for trim bits it may be convenient to have ways of storing the trim bits in different ways. We will need to be able to store trim bits so that we can set them independently for each pixel. But there are other cases where we might want to set all trim bits the same on a whole ROC, or the same in each double column.

Consider that we have a base class, `PixelTrimBase` and that there are the concrete implementations `PixelTrimAll`, `PixelTrimROC`, and `PixelTrimDCol` which implements the per pixel, per roc, and per double column respectively. The interface should support operations like

```
PixelTrimBase *tempTrims=0;
PixelConfigInterface::get(tempTrims,"pixel/trim/FPix_BpI_D1_BLD1_PNL1",
                  theGlobalKey);
```

where after `tempTrims` points to the data type stored in the database for the configuration.

Similarly you should be able to store data

```
PixelTrimBase *tempTrims=new PixelTrimROC; //probably would not compile as
                                           //we don't have this constructor.
unsigned int ver=PixelConfigInterface::put(tempTrims,"pixel/trim/FPix_BpI_D1_BLD1_PNL1"
```

7.2 Command line interface to configuration data

Based on the interface described above in the C++ interface a simple command line tool has been written to allow manipulation of configuration data. The functionality of this interface is described below.

7.2.1 Inserting new data

Here we will start be discussing how you insert a `PixelDetectorConfig` object. There is only one such object in the configuration. Assuming that we have this in a file named `detconfig.dat`. We now want to insert this into the configuration database. This is done with

```
PixelConfigDBCmd.exe --insertData detconfig detectconfig.dat
```

This would install the content of the file `detconfig.dat` under the path `detconfig` by creating a new version. This new version is returned by the command so that it is known where it was installed. The implementation of this tool reads the file to create a `PixelDetectorConfig` object and then uses the C++ interface to store the data.

DAC vs. dac

As a variation of this you might want to install, e.g., a new set of DAC settings for the ROCs. As we insist that no data can be changed in the configuration database after it has been loaded this implies that the DAC settings for all ROCs needs to be loaded at once. To insert that prepare the files that contains the ROC dac settings. Then create the file daclist.txt that list all the file that you want to install. To install them into the database us

```
PixelConfigDBCmd.exe --insertDataSet dac daclist.txt
```

Again a new version has been created under the path `dac` and all the dac settings have been uploaded. The new version is then printed out.

Note that these interfaces are designed to not allow you to change any existing data. (We might want to consider allowing adding comments to existing versions of configuration data.)

7.2.2 Retrieving data

From the command line you can retrieve data using

```
PixelConfigDBCmd.exe --getVersion nametranslation/ 0
```

This will retrieve the data from version 0 of the nametranslation and write it out as a file.

7.2.3 Creating new configurations

So far we have discussed how to add new data to the configuration database. The next step is to combine versions of several data types into a configuration. This is done with a command like

```
PixelConfigDBCmd.exe --insertConfigAlias Physics dac 1 detconfig 0 nametranslation 0
```

where the versions of the different objects on the paths are listed. This will create a new configuration key. This key will be returned by the command.

Very often we have an existing configuration, `oldKey`, that we want to 'update'. Note that update actually means that we will create a new configuration. For example you can do

```
PixelConfigDBCommand --updateConfigAlias 5  tbm 6
```

not yet implemented. So in the example above a copy of configuration key number 5 would be made in which the TBM settings version 6 was added.

If one wants to remove an existing object from a configuration we could imagine doing something like

```
PixelConfigDBCommand --updateConfigAlias 5  calib -1
```

In both these cases the new configuration key that was created is returned. Again this interface guarantees that no existing data has been changed. Only new data has been added.

So now we have version 0 for each of these objects. Now I can create aliases for the different objects.

```
PixelConfigDBCommand --insertVersionAlias detconfig 0 Physics
PixelConfigDBCommand --insertVersionAlias tbm 0 Default
PixelConfigDBCommand --insertVersionAlias dac 0 Default
PixelConfigDBCommand --insertVersionAlias mask 0 Default

PixelConfigDBCommand --insertConfigAlias Physics dac Default detconfig Physics
tbm Default mask Default
```

This command will create the first configuration key, number 0. Now say that a new set of dac settings is created and loaded:

```
PixelConfigDBCommand --insertData dac daclist.txt
```

This will be version 1 of the dac settings. Next if you make this alias the 'Default':

```
PixelConfigDBCommand --insertVersionAlias dac 1 Default
```

The code will automatically update all toplevel aliases that are using this the 'Default' dac settings. In this example it means that the top level 'Physics' alias will point to key 1.

7.3 Managing Configurations

We need higher level tools to manage configurations. The way we currently have the configurations implemented there are about 15 different objects that are needed to build a configuration. Two examples of such configurations are:

```
key 0
detconfig   2
nametranslation   0
fedconfig   0
fecconfig   0
fedcard   0
dac   2
mask   2
trim   2
calib   0
tbm   0
portcard   0
portcardmap   0
ttcciconfig 0
tkfecconfig 0
```

```
key 1
detconfig   2
nametranslation   0
fedconfig   0
fecconfig   0
fedcard   0
dac   2
mask   2
trim   2
calib   8
tbm   0
portcard   0
portcardmap   0
ttcciconfig 0
tkfecconfig 0
```

For a moment I will not discuss the 'aliases', I will focus just on the data organization and the tools needed to manipulate the data. The different data types used in the configurations are discussed in Sect. 8 .

7.4 Configuration DB Implementation

to be added. Stay tuned.

8 Configuration Objects

This section describes the implementation of the configuration objects for the pixel online system. The interface for accessing this data was discussed in the previous section.

8.1 Introduction

Key goals for the design of the configuration data object include:

- The configuration has to be fast and reliable. The fewer components, read pieces of software, involved in the configuration step the more likely the system is to work reliably.
- The data volume should be small. I.e. the data has to be packed in an efficient way.
- Want to optimize the database access by accessing relatively few, but large, objects.

- Computers are good at manipulating data that is in memory, so the actual commands that are sent to the hardware can be built on the fly – assuming that all information to do this is accessible in memory.
- The data volume for the whole pixel system is $\mathcal{O}(100$ MB) so holding this in memory in a single computer should not be an issue. In fact this will be spread over more than one computer as the FECs are in more than one crate.

The following classes are used to configure the online pixel applications:

PixelTrimAllPixels: This class stores the trim bits for the ROCs on one module. The trims are stored for each pixel.

PixelMaskAllPixels: This class stores the mask bits for the ROCS on one module. The masks are stored for each pixel.

PixelDACSettings: This class stores the DAC settings for all ROCs on one module.

PixelTBMSettings: This class stores the TBM settings for one TBM.

PixelNameTranslation: This class translates from the pixel naming scheme documents names of ROCs to the hardware addresses used by both the FEC and the FED to identify a ROC.

PixelDetectorConfig: This class lists the modules used in a configuration. The utility of this class is that it allows one to only use a small subset of the detector without having to create a new name translation.

PixelROCStatus: This class keeps track of the status of ROCs. The default assumption is that a ROC is working and this object allows us to list ROCs that are not working, or that we want to turn off.

PixelFECConfig: This class lists the pixel FECs that are used.

PixelTKFECConfig: This class lists the tracker FECs that are used.

PixelFEDConfig: This class lists the pixel FEDs that are used in the configuration.

PixelFEDCard: This class stores the settings for one FED board.

PixelPortCard: This class stores the settings on a portcard, e.g. the delay25 settings and AOH settings.

```
 private:

    //Hold pointer to the mask override information.
    PixelMaskOverrideBase* maskOverride_;


};
```

The concrete implementation that implements mask bits for each channel looks like:

```
class PixelMaskAllPixels: public PixelMaskBase {

 public:

    PixelMaskAllPixels(std::string filename);

    void writeBinary(std::string filename) const;

    void writeASCII(std::string filename) const;

    const PixelROCMaskBits& getMaskBits(int ROCId) const;

 private:

    std::vector<PixelROCMaskBits> maskbits_;

};
```

The file format that we use looks like:

```
ROC:     FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC1
col00:   1111000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
col01:   0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
col02:   0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
.
.
.
col49:   0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
col50:   0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
col51:   0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
ROC:     FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC2
```

```
col00:   1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
col01:   2000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
col02:   3000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
.
.
.
```

Where the file contains the data for each of the ROCs in a module. Within each ROC the trim bits are listed for each column by the value if the trim bit as one hexadecimal character from 0 to F.

Similarly for the trim bits we have the base class:

```
class PixelTrimBase: public PixelConfigBase {

 public:

    PixelTrimBase(std::string description,
  std::string creator,
  std::string date);

    virtual ~PixelTrimBase();

    void setOverride(PixelTrimOverrideBase* trimOverride);

    //Build the commands needed to configure ROCs
    //on control link

    virtual void generateConfiguration(PixelFECConfigInterface* pixelFEC,
       PixelNameTranslation* trans,
       const PixelMaskBase& pixelMask) const =0;
    virtual void writeBinary(std::string filename) const =0;

    virtual void writeASCII(std::string filename) const =0;

    virtual PixelROCTrimBits getTrimBits(int ROCId) const =0;

    friend std::ostream& operator<<(std::ostream& s, const PixelTrimBase& mask);


 private:

    PixelTrimOverrideBase* trimOverride_;
```

```
};
```

And the concrete implementation looks like:

```
class PixelTrimAllPixels: public PixelTrimBase {

 public:

    PixelTrimAllPixels(std::string filename);

    //Build the commands needed to configure ROCs
    //on control link

    void generateConfiguration(PixelFECConfigInterface* pixelFEC,
       PixelNameTranslation* trans,
       const PixelMaskBase& pixelMask) const;

    void writeBinary(std::string filename) const;

    void writeASCII(std::string filename) const;

    PixelROCTrimBits getTrimBits(int ROCId) const;

 private:

    std::vector<std::string> rocname_;
    std::vector<PixelROCTrimBits> trimbits_;

};
```

We use basically the same format for the mask bits as we used for the trim bits:

```
ROC:    FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC1
col00:  1111111111111111111111111111111111111111111111111111111111111111111111111111111110000000
col01:  1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
col02:  1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
.
.
.
col49:  1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
col50:  1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
col51:  1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
ROC:    FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC2
```

```
col00:  1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
col01:  1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
col02:  1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
.
.
.
```

Here the mask bits are either 0 or 1 for each pixel.

8.3 ROC DACs

The DAC settings for each readout chip are stored in the class

```
class PixelDACSettings: public PixelConfigBase {

 public:

    PixelDACSettings(std::string filename);

    PixelROCDACSettings getDACSettings(int ROCId) const;

    //Generate the DAC settings
    void generateConfiguration(PixelFECConfigInterface* pixelFEC,
                       PixelNameTranslation* trans) const;

    void writeBinary(std::string filename) const;

    void writeASCII(std::string filename) const;

    friend std::ostream& operator<<(std::ostream& s, const PixelDACSettings& mask);

 private:

    std::vector<PixelROCDACSettings> dacsettings_;

};
```

The format for the DAC settings in the ASCII format is given by

```
ROC:            FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC1
Vdd:            6
Vana:           140
```

```
FPix_BmI_D1_BLD1_PNL1
FPix_BmI_D1_BLD1_PNL2
.
.
.
```

The format above is the 'old' format. After discussions with the database GUI developers we have decided to make this object specify the ROC and their status. All ROCs on a module have to be listed. Otherwise there is an internal inconsistency in the configuration. In addition to listing the ROC one can specify a status of the ROC. An example of the file is given below

```
Rocs:
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC0
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC1
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC2 off
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC3
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC4 noAnalogSignal
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC5
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC6 off noHits
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC7
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC8
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC9
```

The status words are explained in more detail in Sec. 8.5.

The class `PixelConfigurationVerifier` checks the internal consistency of the detector configuration. For instance, it checks that if any ROC on a FED channel is marked with `noAnalogSignal`, then the entire FED channel is similarly marked. Also, it ensures the consistency of the FED card with the detector configuration. If an entire FED channel is marked as `noAnalogSignal`, then the corresponding FED channel is automatically disabled. Similarly, if a FED channel is disabled in the FED card, but enabled in the detector configuration, then the FED card is dynamically modified to conform to the detector configuration. In this way the the detector configuration is the "master" flag for what parts of the detector are included in the configuration.

We should check if the detector configuration currently controls which portcard devices are initialized. Ideally if a portcard is not used, then it should not be initialized during the configuration.

8.5 PixelROCStatus

The `PixelROCStatus` class is used to store the status of ROCs. The default assumption is that a ROC is working and is on. However, we are likely to have problems with some ROCs given the number of components we have. This structure should allow us to add new failure modes as we discover new problems. An example of the data would look like

```
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC2 noHits
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC7 off
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC8 off noHits
```

The different status flags that we can set are:

- `noHits` indicates that we can not generate hits on the ROC. For example this means that the ROC can not be calibrated e.g. for address levels. However, it is not preventing us from doing the UB equalization. In principle this flag should be handled on a calibration-by-calibration basis, but at the moment it is ignored.
- `off` indicates that the ROC is disabled via a control bit on the ROC. It is not used in any calibration and will not generate hits. However, even it a ROC is `off` it will be configured. Presently the use of this flag is not implemented (it is ignored by the code).
- `noInit` indicates that the ROCs in a module should not be included in the configuration. This is implemented.
- `noAnalogSignal` indicates that the ROC can configured, but that something in the analog readout is broken. The ROC is included in the configuration but excluded from calibrations. This is implemented, and the corresponding FED channel is automatically disabled.

With the file-based configuration, we can set more than one of these flags at once. (Likely one would implement this as a bitmap.) However, the database configuration does not allow more than one flag to be set at a time.

8.6 PixelNameTranslation

This class generates the translation between the names used in the naming document and the hardware addresses. This includes both the FEC and the FED.

The data format used for the name translation is given by:

# name	TBMchannel	FEC	mfec	mfecchannel	hubaddress	portadd	rocid	FED	channel	roc#
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC0	A	1	8	1	31	0	0	1	12	0
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC1	A	1	8	1	31	0	1	1	12	1
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC2	A	1	8	1	31	0	2	1	12	2
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC3	A	1	8	1	31	0	3	1	12	3
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC4	A	1	8	1	31	0	4	1	12	4
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC5	A	1	8	1	31	0	5	1	12	5
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC6	A	1	8	1	31	0	6	1	12	6
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC7	A	1	8	1	31	0	7	1	12	7
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC8	A	1	8	1	31	0	8	1	12	8
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC9	A	1	8	1	31	0	9	1	12	9

The name translation allows us to map the ROC name to the hardware addresses used in the configuration.

8.7 PixelFECConfig

This class specifies the location of the pixel FECs. This basically gives the VME base address to use. An arbitrary FEC number is used here. *is this specified in the naming convention document?* Also we refer to the crate number here. There will be one PixelFECSupervisor per crate. How is this number identified? The PixelFECSupervisor is initialized with a crate number and will control the FEC cards that are in the crate.

The file format that we have to store this information looks like

```
#FEC number     crate     vme base address
1               1         0x80000000
```

Each FEC is identified by a number. The FEC is identified by the crate and base address.

8.8 PixelTKFECConfig

This class specifies the location of the tracker FECs in the system. This specifies the VME slot and crate used for each TKFEC board. An arbitrary TKFEC ID string is used here. *is this specified in the naming convention document?* There will be one PixelTKFECSupervisor per crate. The PixelTKFECSupervisor is initialized with a crate number and will control the TKFEC cards that are in the crate.

The file format that we have to store this information looks like

```
#TKFEC ID     crate     VME/PCI     slot/address
tkfec1          1                       0x1c
```

Each TKFEC is identified by an ID string. (Currently this string is arbitrary but we should use an agreed upon convention.)

Optionally, to use a PCI TKFEC, the string "PCI" may be added between the crate number and slot number. "VME" may also be specified. If this parameter is not specified, it defaults to VME.

8.9 PixelFEDConfig

This specifies how the FEDs are configured. This includes the FED number and the VME base address. The FED number is the same as the FED id in the Slink data. Each PixelFEDSupervisor is initialized with a crate number corresponding to the crate it controls.

The file format that we have to store this information looks like

```
#FED number     crate     vme base address
1               1         0x1c000000
```

Each FED is identified by a number, this is the same number as the FED id in the raw data. The FED is identified by the crate and base address. For now the crate is identified by an arbitrary number. *Should this be changed to the actual name of the crate?*

8.10 PixelCalibConfiguration

This class was formerly known as PixelCalib, but was renamed after it was moved to the CMSSW repository in order to make it more consistent with offline conventions. This class incorporates information about how a calibration is executed. In particular it handles calibrations where groups of pixels have charge injects. It specifies how we loop over pixels and pulse the detector in a calibration. The class is also used in the offline in order to analyze the calibration data, so that we know what event had what charge injected and what pixels were expected to be hit.

Below is an example of this file:

```
Mode: ThresholdCalDelay
Rows:
10 | 20
Cols:
10 | 20
VcalHigh
Scan: VcThr 0 255 8
Scan: CalDel 0 255 8
Set: Vcal 50
Repeat: 10
Rocs:
FPix_BmI_D1_BLD1_PNL1_PLQ1_ROC0
FPix_BmI_D1_BLD1_PNL1_PLQ1_ROC1
FPix_BmI_D1_BLD1_PNL1_PLQ2_ROC0
FPix_BmI_D1_BLD1_PNL1_PLQ2_ROC1
.
.
.
```

The `Scan` statement allows you to specify that you want to scan the settings of a dac parameter in a range, above starting at 0 and incrementing in steps of 8 until it exceeds 255. In addition to this you can specify a non-uniform set of scan points using the following format

```
ScanValues: Vcal 10 20 30 35 40 42 44 46 48 50
                 52 54 56 58 60 65 70 80 90 100 -1
```

The `-1`

Arbitrary parameters may be specified just before the "Rows:" line. For example,

```
Mode:  AOHBias
Parameters:
TargetBMin      412
TargetBMax      612
```

```
printFEDRawData              no
printFEDOffsetAdjustments    no
printAOHBiasAdjustments      no
Rows:
.
.
.
```

These parameters are accessible in the calibration code. Each calibration may look for particular parameters to control its operation. Parameters not defined for a particular calibration are ignored.

In particular the parameter "ScanMode" can be defined. It can take the three values of "maskAllPixel", "useAllPixel", and "default". In the default mode the trim and mask bits specified in the configuration is used during the scan. Charge is injected according to the pattern and the pixels that are enabled in the configuration is used during the calibration. In the maskAllPixels mode all pixels are disabled before the first event. Then the pixels are enabled corresponding to the pixel mask and the pattern that has charge injected. I.e., only pixels that have charge injection and that are not disabled in the configuration will be enabled. The pixels use the trim bits from the configuration. In the useAllPixels mode all the pixels on the current pattern is enabled independently of what the mask bit is in the configuration if ScanMode is not defined the mode useallPixels will be used.

DACs to scan are selected with lines of the form

```
Scan: [DAC name] [min scan value] [max scan value] [scan step size] [mix]
```

The last parameter is optional. If nothing is given here, then all ROCs will be set to the same DAC value at the same time. If the word `mix` is placed here (at the end of the line), then the ROCs on a particular channel will have not have the same DAC value at the same time. Instead, the DAC values on different ROCs will be spread out to cover the entire range. This is useful when scanning `Vsf` or any other setting that affects the power drawn by the chip, as it prevents the ROCs from all drawing high power at the same time.

The list of ROCs may be specified completely, or it may be auto-generated. Auto-generation requires knowing which modules are in the configuration, and which ROCs are on those modules. Currently, this information is not available in offline (i.e. CMSSW) code, so offline code must have the ROC list specified completely. To do this, the format is:

```
Rocs:
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC0
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC1
.
.
.
```

In online running, where this information is available, it is preferable to use auto-generation. To auto-generate the list, the first line should be "`ToCalibrate:`" instead of "`Rocs:`". (The

```
ToCalibrate:
+ all
- FPix_BpI_D1_BLD1_PNL1
+ FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC0
```

This adds all ROCs on all modules, except for `FPix_BpI_D1_BLD1_PNL1`, on which only `FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC0` is added.

Note that the order matters. The following:

```
ToCalibrate:
+ all
+ FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC0
- FPix_BpI_D1_BLD1_PNL1
```

would not include `FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC0` in the ROC list, because it is later removed by the removal of `FPix_BpI_D1_BLD1_PNL1`.

(For completeness, even though it's completely useless, you can use "`- all`" to clear the ROC list.)

The auto-generated ROC list only adds ROCs on modules given in the `PixelDetectorConfig`.

8.11 PixelFedCard

The `PixelFEDCard` class contains the settings for a FED board. The file is fairly long and I do not include an example here. You can find a sample file in PixelFEDInterface/test/params_fed.dat *There is some information on the PixelFEDCard that is redundant with the PixelFEDConfig like the VME base address. We should only specify this in one location.*

8.12 PixelTBMSettings

This class holds the settings used by the TBM. The file contains the module name and gain settings as well as if the TBM should be configured in 'SingleMode' or 'DualMode'.

The format used for this information is given by

```
FPix_BpI_D1_BLD1_PNL1_PLQ2_ROC1
AnalogInputBias: 160
AnalogOutputBias: 110
AnalogOutputGain: 207
Mode: SingleMode
```

Note that the name here should be a module name and not contain the plaquette and ROC number. We now have a 'PixelModuleName' class that we should use here.

```
GridSize:
8
Tests:
10
StableRange:
6
StableShape:
2
```

8.16 PixelGlobalDelay25

The PixelGlobalDelay25 is a class that contains one global delay setting that delays the signal in delay25 chip for the clock and data. In the FED it adds the same delay such that the digitization works independently of the delay setting in the global delay.

The format of this file is very simple, it is just one single number in hex

```
0x10
```

The delay specified here is in units of 0.499 ns. This corresponds to the steps of the delay in the delay 25 chip.

9 Usage of configuration data in xdaq applications

This section contains a brief description of how the configuration data objects are used.

First we look at the PixelSupervisor. RCMS, via the PixelFunctionManager, will pass a string for a configuration alias to the PixelSupervisor during the configure transition. The PixelSupervisor gets the configuration alias and looks up the corresponding configuration key:

```
std::string alias=parametersReceived[0].value_;
unsigned int globalKey=PixelConfigDB::getAliases_map().find(alias)->second;
theGlobalKey_=new PixelConfigKey(globalKey);
```

This code should be fixed so that it catches if the alias does not exist. This can be an assert, as the choices for the alias are listed from the same map and not being able to find it is an internal error.

Having obtained the configuration key this is what is used to extract configuration data. For example the PixelSupervisor extracts some objects:

```
PixelConfigDB::get(theCalibObject_, "pixel/calib/", *theGlobalKey_);
PixelConfigDB::get(theDetectorConfiguration_, "pixel/detconfig/", *theGlobalKey_);
```

The get method returns a pointer that the PixelSupervisor is responsible for deleting.

When the PixelSupervisor asks the other supervisors to configure it does this by passing the configuration key, not the alias, to them. This guarantees that the configuration used by the different supervisors is consistent. We only need to retain the configuration key as a record of how the detector was configured.

Inside the PixelFECSupervisor in the configuration method we have code like

Note that this code snippet is a bit out of date, although the idea remains the same.

```
PixelConfigDB::get(theNameTranslation_, "pixel/nametranslation/", *theGlobalKey_);
assert(theNameTranslation_!=0);

PixelConfigDB::get(theDetectorConfiguration_, "pixel/detconfig/", *theGlobalKey_);
assert(theDetectorConfiguration_!=0);

PixelConfigDB::get(theFECConfiguration_, "pixel/fecconfig/", *theGlobalKey_);
assert(theFECConfiguration_!=0);
assert(theFECConfiguration_->getNFECBoards()==1); //FIXME

PixelConfigDB::get(theCalibObject_, "pixel/calib/", *theGlobalKey_);
calibStateCounter_=0;

// Loop over all modules in the Detector Configuration and instantiate FECInterfaces re
// Download TBM, DAC, Masks and Trim settings into hardware.
std::vector <PixelModuleName>::iterator module_name = theDetectorConfiguration_->getMod
for (;module_name!=theDetectorConfiguration_->getModuleList().end();++module_name)
{
 diagService_->reportError("Congiguring module=" + module_name->modulename(),DIAGDEBUG)
 const PixelHdwAddress* module_hdwaddress=theNameTranslation_->getHdwAddress(*module_na

unsigned int fecnumber=module_hdwaddress->fecnumber();
unsigned int feccrate=theFECConfiguration_->crateFromFECNumber(fecnumber);
unsigned int fecVMEBaseAddress=theFECConfiguration_->VMEBaseAddressFromFECNumber(fecnum

if (feccrate==crate_){
  PixelMaskBase *tempMask=0;
  PixelTrimBase *tempTrims=0;
  PixelDACSettings *tempDACs=0;
  PixelTBMSettings *tempTBMs=0;
  std::string modulePath=(module_name->modulename());

  PixelFECInterface* tempFECInterface=new PixelFECInterface(fecVMEBaseAddress, aBHandle
  assert(tempFECInterface!=0);
  tempFECInterface->setssid(4);
```

```
    PixelConfigDB::get(tempMask, "pixel/mask/"+modulePath, *theGlobalKey_);
    assert(tempMask!=0);
    theMasks_.insert(make_pair(*module_name, tempMask));

    PixelConfigDB::get(tempTrims, "pixel/trim/"+modulePath, *theGlobalKey_);
    assert(tempTrims!=0);
    theTrims_.insert(make_pair(*module_name, tempTrims));

    PixelConfigDB::get(tempDACs, "pixel/dac/"+modulePath, *theGlobalKey_);
    assert(tempDACs!=0);
    theDACs_.insert(make_pair(*module_name, tempDACs));

    PixelConfigDB::get(tempTBMs, "pixel/tbm/"+modulePath, *theGlobalKey_);
    assert(tempTBMs!=0);
    theTBMs_.insert(make_pair(*module_name, tempTBMs));

    tempDACs->generateConfiguration(tempFECInterface, theNameTranslation_);
    tempTBMs->generateConfiguration(tempFECInterface, theNameTranslation_);
    tempTrims->generateConfiguration(tempFECInterface, theNameTranslation_, *tempMask);

    FECInterface[fecVMEBaseAddress]=tempFECInterface;
  }
 }
```

Similar extractions of the configuration data is used by other supervisors. The supervisors cache the data received. In general, configuration data is cleared in the halt transition. However, we now hold on to most of the configuration data in the FECSupervisors, and only clear it before the next configuration if we see that the value of the global key has changed. In this way, we can avoid reloading identical data from the database on the next configuration.

9.1 Global delay 25 usage

The global delay 25 does not need to be included in a configuration. Then it is simply ignored if the PixelConfigInterface::get call returns a null pointer. However, if it exists in the configuration it has the following effects in the different applications:

In the PixelTKFECSupervisor it is checked that we are in a physics run and have the global delay 25 settings. If this is the case then SCL and TRG are delayed by the delay setting in the global delay 25. *It is checked in the code if the calculated delay 25 setting is larger than 127. If this is the case then an error message is printed and the delay 25 without the global delay is applied. This is a little bit dangerous as it is easy to miss such a message.*

Should think about a safer way of handling this. For the SDA, if the calculated delay25 setting is to large, it allows the SDA to wrap around.

In the PixelFEDSupervisor the a similar logic is applied; if you have the global delay 25 and you are taking a physics run then the TTC RX chip adds a delay. This delay is calculated using the method `PixelGlobalDelay25::getTTCrxDelay`.

Comments by Anders: I think that we should modify this code such that the global delay25 delay is applied when your run all calibrations except for the Delay25 scans. This will allow simpler tests to make sure that the delays are properly applied, e.g., by running the address level calibration with different global delay 25 settings.

`std::map<instance, state>` maintained by the PixelSupervisor. These objects track the state of the underlying supervisors[6], and are called:

`statePixelFECSupervisors_`
`statePixelFEDSupervisors_`
`statePixelTKFECSupervisors_`
`statePixelDCSFSMInterface_`

These are initialized in `PixelSupervisor::Initialize` by actively asking each supervisor with a SOAP command "FSMStateRequest".

Note that the first step in the configuration after loading the global key is to send it to the FECSupervisors. Allow the FECSupervisors cannot begin programming the hardware until later in the configuration sequence, they can then begin to immediately fetch configuration data from the database. This step, called preConfiguration, is done outside of the finite state machine structure (the PixelFECSupervisors remain in the Halted state even as they do the preConfiguration). This step saves considerable time in the configuration process (roughly 25 seconds when configuring from the database).

In the `PixelSupervisor::stateConfiguring` method, the PixelSupervisor loops over all the PixelTKFECSupervisors, checks the FSM state of each as maintained in `statePixelTKFECSupervisors_` and if it finds "Halted", it tries to "Configure" it and updates the map to read FSM state "Configuring". If any of these PixelTKFECSupervisors are not in the "Configured" state, the ::stateConfiguring function exits. The PixelSupervisor then does nothing until it receives a "FSMStateNotification" SOAP message from PixelTKFECSupervisor with the message "Configured". This FSMStateNotification signals the PixelSupervisor to make a transition from "Configuring" to "Configuring", thus triggering ::stateConfiguring to run again. In this way the PixelTKFECSupervisors configure in parallel, and can take as long as they like to finish configuration.

Once the TKFECSupervisors are done, we can configure the pixel FECs and FEDs in parallel. The procedure is similar: if their FSM state is "Halted", we give them the SOAP message to Configure and set their local map state to "Configuring". And now we can configure the FEDs too in a similar, parallel, manner. (Note that we do not return out of the function directly, but rather if we detect one of the supervisors to not be in "Configured", we set a bool called "proceed" to false, and this bypasses a large chunk of code that configures the LTC and TTC, and transitions the FSM of PixelSupervisor to "Configured".) Only when all FECs and FEDs are configured, then we proceed to configure the TTC & LTC, and then the PixelSupervisor is pushed into the "Configured" state.

10.1.1 Possible modifications to the configuration Sequence

In principle there is nothing to prevent the TTC and LTC from being configured before the other Supervisors. Arguably, this would be more logical, since the trigger source should be configured before the other devices are programmed.

[6]Note that these objects are useful for more than just configuration, and are always kept updated with the state of each Supervisor

10.2 Configuration steps of the underlying supervisors

In this section we give an outline of the steps taken by each supervisor during configuration.

10.2.1 PixelDCSFSMInterface and PixelDCStoTrkFECDpInterface

The PixelDCSFSMInterface loads the detector configuration. If it finds that an entire ROG is in `noInit` or `noAnalogSignal`, then it ignored that ROG when summarizing the power state of that section of the detector. Note that initially this was true only for "noInit", since a ROG marked `noAnalogSignal` will still be configured and thus needs LV to be on. This logic was changed to accomodate turning on only a small fraction of the HV during the first beam operations.

Note that presently the relationship between detector modules (as listed in the detconfig) and DCS ROGs is hard-coded in this class. See Sec. 15.9 for more information.

10.2.2 PixelTKFECSupervisor

Need to look through the code to verify this info! ← ?

- Create a FecAccess object and issue a VME bus reset
- Issue a resetPlxFec to reset the CCU and portcard devices
- Load the portcard configuration data from the database and program the portcard devices (AOH, DOH, Delay25, etc)

Note that in this last step the data is programmed to the hardware in the same order that it is provided by the class that loads the data from the configuration files or database. That class passes file-based data directly, in the same order as it appears in the files. For the database, it applies a crude sorting algorithm to put PLL settings first, Delay25 settings second, and AOH settings last. Any other settings come between the Delay25 and AOH settings.

10.2.3 PixelFEDSupervisor

10.2.4 PixelFECSupervisor

} Text missing !

10.3 Quick reconfiguration for the fine delay scan

A special "Reconfiguration" option is available for changing the settings that are directly relevant to the global delay of the detector with respect to the trigger and the clock.

10.3.1 Relevant settings

The following settings are modified during the reconfiguration process:

- PixelTKFECSupervisor: the SCL, TRG, and SDA registers of the Delay25 chip

Table 5: Pass condition and summary information stored in the trees. If the condition for failure is satisfied, then the '"Pass" state is stored as 0 in the summary tree. Otherwise the "Pass" state is stored as 1.

Calibration	Condition for Failure	Summary Information
Address Levels	Recommended Level $0^{\dagger}$ is less than UltraBlack high threshold or Number of ROC peaks found is not equal to 6	number of peaks, maximum peak RMS minimum peak separation, RMS of the black levels
ROC UB	The measured UB does not cross the target level	new VIBias value and change of VIBias
Vs	no hits found or no good Vsf found during scan	new Vsf value and the change of Vsf
VHldDel		new VHldDel and its change
LinearityVsVsf	no hits found or no good Vsf found during scan	new Vsf value and its change
PHRange	no hits found or no settings give PHInRange or only one Vcal produces hits	new VIbias_PH, VOffsetOp and their changes new VIon and VOffsetRo and their changes
VcThrCalDel	there is not valid settings	new VcThr, CalDel and their changes
VcThr	there is not valid VcThr	new VcThr and its change
CalDel	there is not valid CalDel slope	new CalDel and its change
Iana	new Vana $\geq$ 249 or maximum Iana $<$ 25	new Vana, change in Vana, new Iana, maximum Iana (fit Iana at Vana = 250), χ^2 of the fit

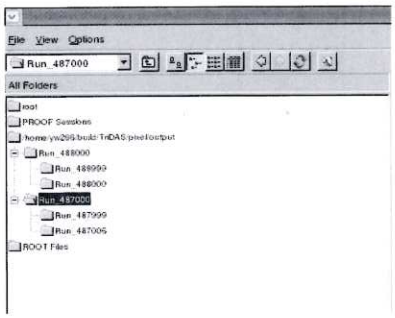


Figure 9: Output directory structure veiwing by ROOT.

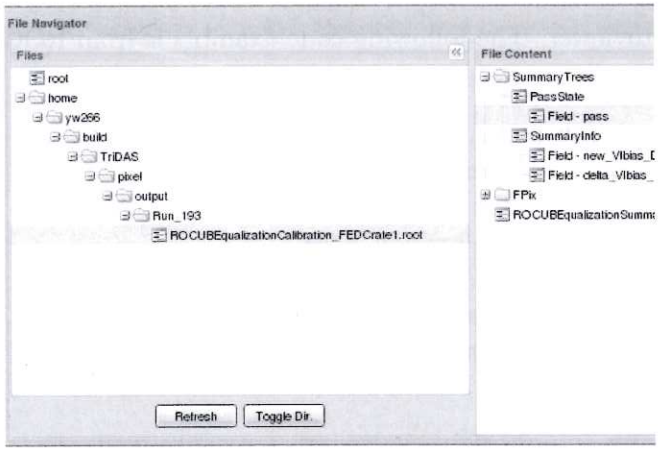


Figure 10: Output root directory structure veiwed by ROOT.

13 Calibration Algorithms

The two main purposes of the online software are the configuration for data taking and the online calibrations that are performed between runs for the calibration of the detector. There are a large number of different calibration tasks that need to be performed. This section describes the algorithms implemented to carry out these calibrations.

A brief summary of the most fundamental calibrations is given here, roughly in order of use. In the next section, Sec. 14 we document our experience calibrating the detector using these calibrations.

The Delay25 calibration is used to ensure correct FEC communication; it is very fundamental but once the settings are found they do not need to be readjusted often. We usually start by running the FEDBaselineCalibration, because it runs quickly and provides an informative plot of the data buffer for each FED channel. It is also needs to be run relatively frequently to adjust the black level. If the BaselineCalibration fails to converge, one can run the AOHBias calibration to more coarsely adjust the black levels, then subsequently try the BaselineCalibration again. The ClockPhase calibration is used to adjust the timing of the signal digitization; a bad clock phase can also cause problems for most other calibrations. The ultrablack levels are adjusted for the TBM using the TBMUB calibration, and then the ROC UB levels are adjusted with the ROCUB calibration. (Note that it is never harmful to repeat the BaselineCalibration as one progresses through these steps.) The AddressLevels calibration is then run in order to be able to decode pixel hits properly. If this calibration has difficulty seeing hits, it may be necessary to adjust the threshold and calibration injection delay using the VcThrCalDel scan (or a related calibration). Once the AddressLevels are found one can test the detector using the PixelAlive or SCurve scans.

More details are given below, along with descriptions of additional calibrations not included in the above list.

13.1 AOH and FED channel mapping test

The AOH and FED channel mapping test is not really a calibration, but rather a test to see whether the connections between AOH channels and FED channels in the configuration are correct. The idea is to change the AOH bias for a single channel, and look to see whether the black level on the corresponding FED channel changes. (FED automatic baseline correction must be turned off.) This process is repeated for each channel. Those for which the black level changes are correctly connected; those for which there is no change are not connected correctly or have a mistake in the configuration files.

13.1.1 Mapping test steps

The steps of this calibration are listed in order below.

1. Set the FED channels to the 2V peak-to-peak range in order to provide maximum dynamic range for the AOH bias scan.

Table 6: Optional parameters for AOH and FED channel mapping test.

Parameter	Default	Description
ScanMin	0	Low end of AOH bias scan range
ScanMax	50	High end of AOH bias scan range
ScanStepSize	5	Step size for AOH bias scan
printFEDRawData	no	Whether to print decoded transparent buffer
printScan	no	Whether to print the AOH bias scan to the screen

Only two "standard" parameters are used. The "`Repeat:`" parameter determines the number of triggers at each AOH bias scan point. The channel list is used to determine which channels are calibrated. Note that the rows, columns, and DAC scan settings in `calib.dat` are completely ignored.

Some optional parameters may also be set. All have default values which will be used if the parameter is not set. These parameters, their defaults, and their functionality are given in Table 6.

13.2 FED phase and delay scan

The FED has a delay (0 to 15) that goes in steps of 25/16 ns. In addition to the delay there is a phase that controls when the data is latched. Certain combinations of the delay and phase are invalid and result in garbage ADC values.

In this calibration the 32 values of the phase and delay are scanned. For each setting of the phase and delay a fixed number of events (typically around 10) is read out in transparent mode.

A detailed description of this algorithm, is presented in Appendix A.

13.2.1 Output

The FED phase and delay calibration produces new fed_params.dat files that are updated with the new FED settings for the phase and delay. The files should otherwise be identical to the input files specified in the configuration. In addition to the fed_params.dat files there are several canvases generated in the output ROOT file. For each channel you get four output canvases:

```
PhaseAndDelayRaw_37_1
PhaseAndDelayPurged_37_1
PhaseAndDelayOrdered_37_1
PhaseAndDelayFinal_37_1
```

Where the numbers in the file name represent the fed id and the fed channel. Appendix A gives examples of these plots.

13.2.2 Example configuration

An example of a configuration file to run the phase and delay scan is given below.

```
Mode: ClockPhaseCalibration
Rows:
Cols:
Vcal:
100 100 5
Repeat:
10
ToCalibrate:
all
```

As described in Appendix A one can change the algorithm used to find the best phase and delay using

```
Parameters:
oldMode Yes
```

13.3 Delay25 settings for send data and return data

This calibration scans the delay for the send data and return data. For each set of values commands are sent to the TBM and the return status in the (pixel) FEC is checked. This calibration uses the PixelFECInterface::testDelay25 method that sends 4 different types of commands. This algorithm combines the results from the 4 different types of commands and only if all 4 succeeded will you get 100% efficiency. Note that one of the commands to be tested is read from the file `infeccmd.dat` in the `PixelRun` directory. If necessary, one can customize this file.

Note that this algorithm does not check that the TBM or ROC actually received the command, it just checks the return status.

13.3.1 Output

The main output from this calibration is new delay settings that stored in the portcard files. The only changes made to the port card settings are for the send data and return data delays.

In addition to generating the new portcard files with delay settings the scans are plotted in a ROOT file. An example plot is shown in Fig. 13.

13.4 Delay25 trigger setting

Not implemented. Should also scan the trigger delay to make sure that the triggers are received correctly. Can we just scan the delay setting and look at the corresponding FED

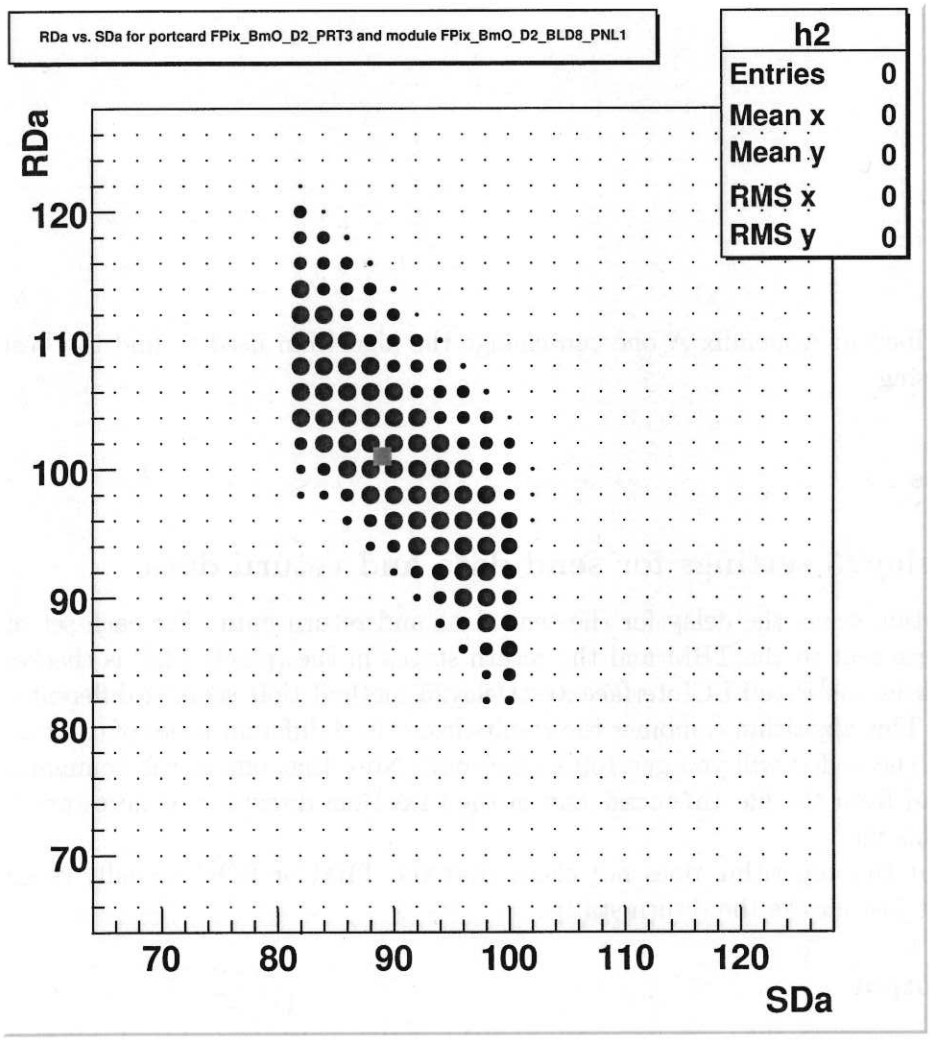


Figure 13: This plot shows efficiency as a function of RDa and SDa. The blue dots indicates areas with 100% transmission efficiency. The black dots indicated partial efficiency, larger dots have higher efficiency. The red square indicates the point chosen by the algorithm.

channel to make sure that the trigger arrived? I.e. that we saw data in the FED? You would need to do this before you run the phase and delay scan of the FED?

13.5 FED baseline calibration

This calibration adjusts the input offset and channel offsets of the optical receivers in the FED such that the black level is adjusted to be near a given target value, normally 450, which is near the midpoint of the dynamic range of the ADC. During this calibration the baseline correction in the FED is turned off.

Besides determining the input offset and the channel offsets the algorithm determines address levels for the black and ultra-black levels.

13.6 AOH bias settings

13.6.1 Introduction and discussion

The AOH bias is a setting on the port card which controls the amount of light sent to the FED. There is one AOH bias setting per FED channel. As AOH bias increases, more light is sent, and the ADC values on the FED increase. At low values of AOH bias, both black and ultrablack do not change with AOH bias, and there is no separation between black and ultrablack levels. At some threshold, the black level begins to increase approximately linearly. At a higher threshold, the ultrablack level also starts to increase linearly with approximately the same slope. This behavior is illustrated in Fig. 14.

Note that the maximum black-ultrablack separation depends on how the TBM DACs are set. At low DAC settings, the TBM outputs a signal with relatively low separation; as these settings increase, the separation also increases. In the AOH bias scan, the black level is independent of the TBM settings. However, the linear rise of the ultrablack level begins at a later point for higher TBM settings, and hence the black-ultrablack difference saturates at a higher AOH bias value when the TBM DAC settings are higher.

The goal of the AOH bias calibration is to determine an AOH bias setting for each channel that is just high enough to saturate the black-ultrablack difference. The calibration measures this difference, using black and ultrablack levels from the TBM header and trailer, as a function of AOH bias. It is important, though, that during the scan the TBM DACs are set at least as high as they will be set in later calibrations and physics runs. Otherwise, the AOH bias value determined from the saturation point will be too low. TBM settings above those used in this scan will not increase the B-UB separation because the AOH cannot provide more separation.

Temperature variations alter the response of the AOH, essentially shifting the curves in Figure 14 to the left or right. In order to provide a margin of error for temperature changes, the AOH bias should be set higher than the saturation value. A temperature increase of 5 degrees Celcius will shift the curves by about 4 AOH bias counts. Therefore, by default the chosen AOH bias setting will be 4 counts higher than the saturation value. This offset is a configurable parameter.

It is also important that the AOH bias not be too high; otherwise the FED offsets could not bring the signal into the dynamic range of the FED.

The last part of the AOH bias calibration is to do a coarse baseline adjustment. The FED channel offsets are set to the center of the range (127), and then the FED optical receiver offsets and AOH bias settings are adjusted to bring all FED baselines into a wide target range. AOH bias is not decreased below the saturation value unless it is absolutely necessary. The end result is a configuration of AOH bias and FED offset values that puts all FED baselines near the center of the dynamic range, with AOH bias values that allow for a large B-UB separation. After the AOH bias calibration, the FED baseline calibration should be run to perform fine adjustments of the baseline (using the freedom to move each channel offset).

13.6.2 AOH bias calibration steps

This calibration involves many distinct steps. They are listed in order below. Each step is performed on each channel being calibrated.

1. Set the FED channels to the 2V peak-to-peak range in order to provide maximum dynamic range for the AOH bias scan.

2. Turn off the FED automatic baseline correction.

3. Issue `ClrCal` to all ROCs, and disable hits with the control register on all ROCs, to ensure that no hits are output.

4. Set all TBM DACs to high values. These values may be specified as parameters in `calib.dat`.

5. Set all FED optical receiver input offsets to the highest useful value (the setting that minimizes the FED ADC values, without impairing the B-UB separation). This value is configurable, and it defaults to 8 (on a scale from 0 to 15). Also, all channel offsets are set their maximum value, 255.

6. Issue `LRES` and `CLRES` commands to all FEDs to clear the transparent data. This ensures that no stale data is sitting in the buffer.

7. Loop over AOH bias values. Attempt to decode the transparent buffer on each trigger, assuming the correct number of ROCs and no hits. That is, find the TBM header, and verify that the TBM trailer is in the right place. When decoding is successful, record the start slot of the TBM header and trailer. On each channel, this slot should be the same for all triggers. The reason for this step is to find the time slots for later use in reading out TBM B and UB levels, even at low AOH bias settings where full decoding

would fail. If no reliable time slots are found[7] on a channel, this channel is considered failed, and it is ignored in the rest of the routine.

8. Now scan over AOH bias again, this time to record the TBM B and UB levels at each setting, using the time slots recorded in the previous step.[8] During the scan, if the signal goes out of range high or low, the FED channel offset is adjusted to bring it back into range, if possible. Since only the B-UB difference is of interest, coherent shifts in B and UB do not matter. The FED optical receiver offset is not adjusted during the scan – it remains at the high setting described in step 5.

9. Output plots of the TBM black and ultrablack, and the difference, as a function of AOH bias, to a ROOT file. Figure 14 is an example of the plots produced. Find the AOH bias value at which the B-UB difference saturates (defined as a reduction in the slope to less than 20% of its maximum value). Set each AOH to its saturation value plus the offset defined by the parameter `SaturationPointOffset` (which defaults to 4).

10. Set FED channels back to the 1V peak-to-peak range, for those channels which were originally set at 1V in the FED configuration file. This is done so that the coarse baseline adjustment will be done with the range that will be used for future data-taking.

11. Set FED optical receiver input offsets to 0 (lowest value, corresponding to highest FED ADC values), and all channel offsets to 127 (middle of the range). The channel offset will be left at 127 for the rest of the routine.

12. Measure black levels on all channels. On each FED optical receiver, if any channel has a black level above the target range, increase that receiver's offset by one. If not, do nothing. Repeat this until all channels have black levels in or below the target range, or have a receiver offset equal to the maximum value described in step 5 (defaults to 8). The idea here is to ensure that no AOH bias value will have to be decreased to place the black level in the target range (unless this is absolutely necessary because the receiver offset cannot be increased further).

13. Again measure the black levels on all channels. If a channel's black level is within the target range, that channel is done. If it is above or below the target range, decrease or increase AOH bias. Repeat this until all channels are within range. (Or, if two adjacent AOH bias values produce black levels that straddle the target range, chose

[7] A time slot is considered found if the most common start slot occurs on at least 95% of the triggers. Alternatively, if the two most frequent time slots differ by 1 (due to a jumping clock) and together account for at least 95% of triggers, then the time slots are considered found, and TBM B and UB are sampled only in those time slots which are known to be B or UB for either of the two start slots.

[8] If good time slots were not found on a given channel, only the black level will be recorded and plotted. It is taken from the first 10 slots of the transparent buffer, rather than from the TBM header and trailer. This plot is for diagnostic purposes only; it is not used in generating new configuration settings.

Table 10: Optional parameters for ROC UB equalization calibration.

Parameter	Default	Description
printFEDRawData	no	Whether to print decoded transparent buffer
printScan	no	Whether to print TBM UB levels for each DAC setting

13.10 Address level determination

The address level determination determines the values used by the FED to encode the transparent data. This include address levels for the pixel addresses, TBM header and trailer levels, and the black and ultrablack levels.

Pixels are scanned to make sure that we probe combinations of address levels that could potentially cause problems, such as transitions from high to low levels and vice versa.

Due to limitations in the size of the FIFO1 transparent data size of 512 clocks, we can not even take one hit per ROC unless the timing is adjusted such that the start of the pulse train comes relatively early, i.e. in the first 20 or so clock cycles. Recall, each ROC, with one hit takes 9 clock cycles to read out. Then with 24 ROCs in a module we need 216 clock cycles just to read out the hits. Plus about 16 clock cycles to read out the TBM headers

13.10.1 Data volume and time estimate

The worst case scenario for an address level calibration is one BPIX crate. We have 16 FED cards in one crate, and we can assume that all of them are fully populated. This means that we have $16 \times 36 = 576$ links. As we read out this data in transparent mode we will read 1024 words per event (even though the transparent data is only 512 bytes long). This means a total of 235926 bytes. If we want to go thought all pixel we have to read out 4160 times this data volume for a total of 9,4 GB. At a speed of 10 MB/s this will take 940 s.

Assuming that we change the transparent data FIFO to be 1024 bytes. Then we can easily fit 4 hits into each ROC. This then cuts the readout time by a factor of 4 as we will need exactly 1/4 the number of triggers. This then becomes 235 seconds.

We can also use different patters that used fewer pixel hits. But one has to take care to have all relevant transitions.

13.11 Pixel alive, Scurve, and Gain calibration

The pixel alive calibration loops over pixel, inject charge, for a fixed VCal setting. The data is analyzed to produce an efficiency map that displays the efficiency for each pixel on a plaquette.

The Scurve and gain calibrations are variations of the pixel alive test were we in addition loop over the VCal setting for each pixel. For the Scurve the data is analyzed to produce an efficiency as a function of the VCal setting while for the gain calibration we look at the charge (ADC value) as function of the VCal.

To analyze the data, go to PixelAnalysisTools/test and do one of the following:

```
./bin/linux/x86/PixelAnalysis.exe PixelAlive <runnum>
./bin/linux/x86/PixelAnalysis.exe SCurve <runnum>
./bin/linux/x86/PixelAnalysis.exe Gain <runnum>
```

Writing e.g. PixelAlive will pick up the default configuration xml file, configurations/PixelAliveAnalysis.xml. You can also specify the exact configuration file you would like to use, e.g. configuration/PixelAliveAnalysis_FPix.xml.

You can scan over multiple WBCs when taking this data and analyze the data from all the WBCs summed together or only one of them using the ChooseWBC option in the configuration xml file. To measure absolute thresholds, use the in-time WBC and the following one. To measure the in-time threshold, use the in-time WBC only.

13.11.1 Practical issues

If you get a lot of "Time Out Error" messages it could be because you don't have the right channels enabled on the FED. Or more specifically that you have enabled channels that are not connected.

13.12 Iana vs. Vana Calibration

A calibration called "Iana" is implemented to scan Vana and measure the analog current, Iana. As the power distribution for the low voltage is done per portcard for the forward pixels (*need to add how this works for BPix.*) this calibration works in parallel on a ROC at the time on each of the portcards. The current changes we are measuring is of the order 50 mA as you go from Vana=0 to Vana=255. The A4603 power modules has a resolution of about 7 mA. We perform this measurement when all other ROCs are configured to their default values. Hence, we measure the current changes on top of a current of several amperes. (I performed trials were I lowered Vana on all other ROCs to reduce the current, but it does not really improve the measurements.) Hence, we have to take the data and fit it in order to interpolate between the points. (It takes a long time to do these measurments as one has to wait up to 6 seconds before the current measurement is stable. This should also be brought up with CAEN.)

To analyze the data we fit it to a functional form. After some experimentation I came up with the following. I divide the range of Vana values into 3 ranges, compare Fig. 16. In the high range Iana is taken to be a constant. In the low range Iana is taken to be a constant plus an exponential. In the middle range Iana depends linearly on Vana, and the function is required to be continuous. In addition the function is required to have a continuous derivative as it goes between the low and middle region. This function has 5 free parameters that are determined in the fit.

First the raw data is fit to this form. After doing this fit the offset at Vana=0 is subtracted and the data is refit and plotted. These are the fits shown in Fig. 16.

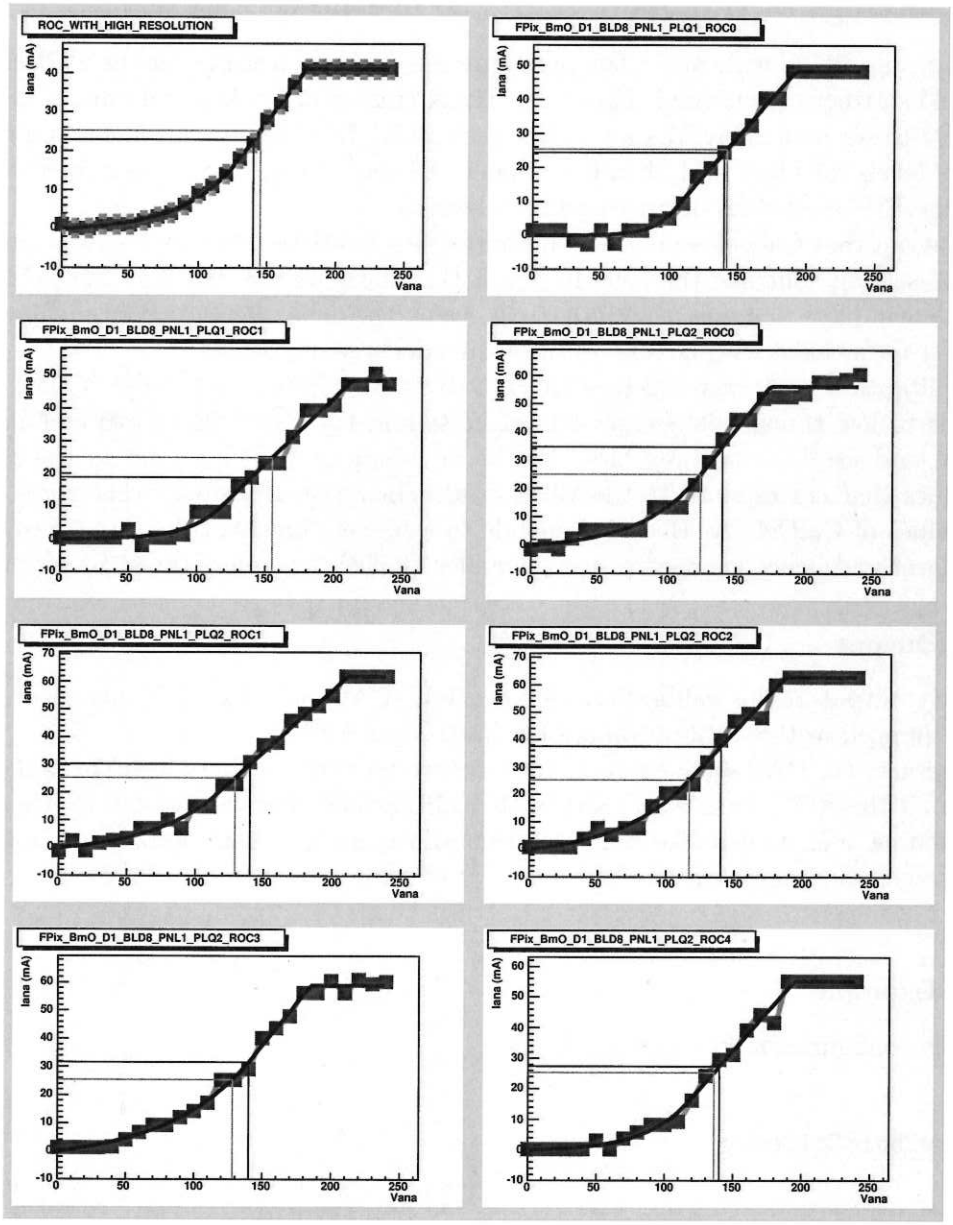


Figure 16: This figure shows the measured analog current, Iana, vs the Vana setting for a few ROCs. The top left plot shows the Iana measured for one ROC with the A1715 power supply that has a better resolution. The black lines indicate the default setting (Vana=140) used in the configuration when this data was taken. The red line indicates the setting that gives a analog current of 25 mA.

A black line is used to indicate where the current used for the default Vana setting (Vana=140). In red is shown where you should set Vana to get an analog current of 25 mA.

13.13 Settings of CalDelay and VcThreshold

These settings are ROC wide and a few pixels are selected and a scan over the CalDelay and VcThreshold setting is performed. For each setting, triggers are taken and data is looked at using FIFO3 to see how many hits we had on each ROC. In order for this algorithm to work the address levels for black and ultra-black has to be set. (There is also an implementation that uses the FIFO1, it does not need address levels.)

We point out that CalDel is *only* relevant for calibration data taken with charge injection. The CalDel setting controlls the time in which the charge is injected into the pixels. For real data we will have to adjust the timing, e.g. using the delay 25 settings from the trigger. However, for data taken with charge injection this delay is important.

This calibration performs a 2-D scan of CalDel vs VcThr. For large VcThr, which corresponds to low thresholds, we get lots of noise and the ROC digital readout basically shuts down and we don't see any hits. For lower values of VcThr we get in the range of CalDel values that corresponds to the WBC used. For larger thresholds this curve 'bends' to lower values of CalDel. As this curresponds to a higher threshold the signal reaches the threshold later and hence we need to use a smaller CalDel, i.e. inject the signal earlier.

13.13.1 Output

The primary output of this calibration are new ROC DAC settings. The only settings that should be changed in this calibration are for CalDel and VcThr.

In addition to the DAC settings the calibration produces the output file `VcThrCalDelaySummary.txt` which lists all the ROCs that were used in the calibrations. For each of the ROCs there is a corresponding file, named like `ThrCalDelScan_FPix_Bm0_D1_BLD8_PNL2_PLQ1_ROC0.dat`. Also, plots and a summary tree are put into a ROOT file. An example of a plot is shown in Fig. 17.

13.13.2 Example

An example configuration file is shown below.

```
Mode: ThresholdCalDelay
Rows:
10 | 20
Cols:
10 | 20
VcalHigh
Scan: VcThr 0 255 8
```

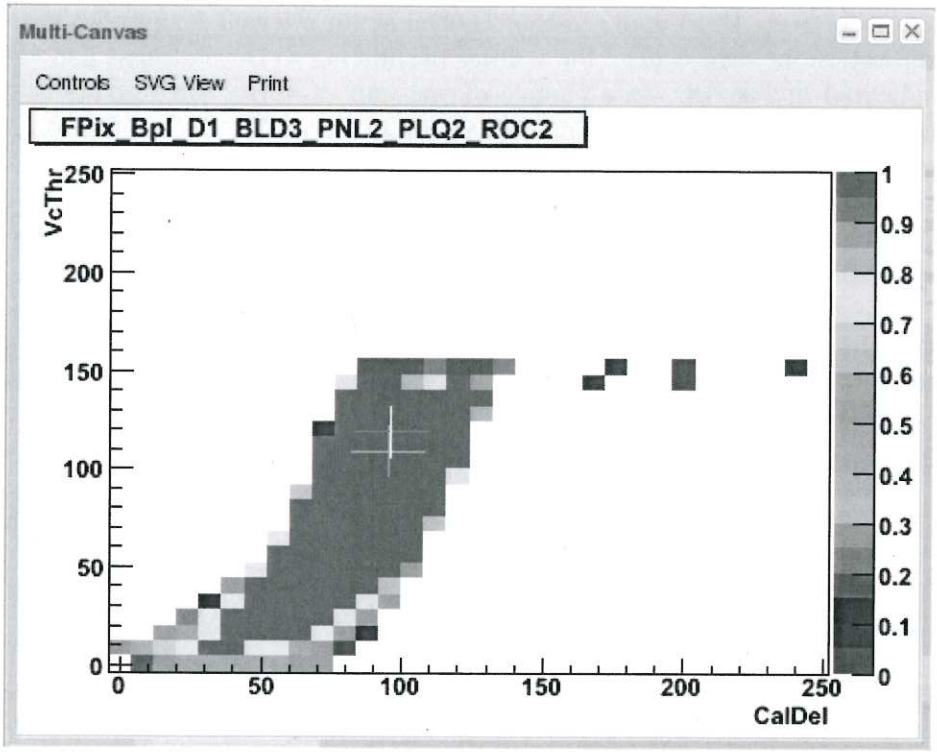


Figure 17: The efficiency for detecting a hit is shown as a function of VcThr vs. CalDel. To large values of VcThr, corresponding to a low threshold, generates much noise that saturates the digital circuit and no hits are seen. The optimal point is indicated in black and the blue point indicates the old point from the configuration.

```
Scan: CalDel 0 255 8
Set: Vcal 50
Repeat: 10
ToCalibrate:
all
```

13.14 CalDel calibation

Again the note from Sect. 13.13 apply in that CalDel is not relevant for physics data.

In this calibration we scan CalDel for a series of different WBC settings. The creates a pattern as indicated in Fig. 18. As a change of one unit of WBC corresponds to 24.95ns (40.079 MHz) we can use the change in CalDel for different WBC settings to calibrate absolutely what a change of CalDel corresponds to in absolute time.

We run this calibration with an injection of a large signal (255 on the high Vcal range). We then pick a Vcal setting that is near the *early* start of the efficient range for the WBC setting. We pick an early time so that we retain efficiency for signals that are not as strong and takes time to get over threshold. How close to the edge the CalDel setting is taken can be specified by the parameter "Fraction". Setting this to 0 means that you pick the point right at the early edge, setting the parameter to 1 means that you select the late edge.

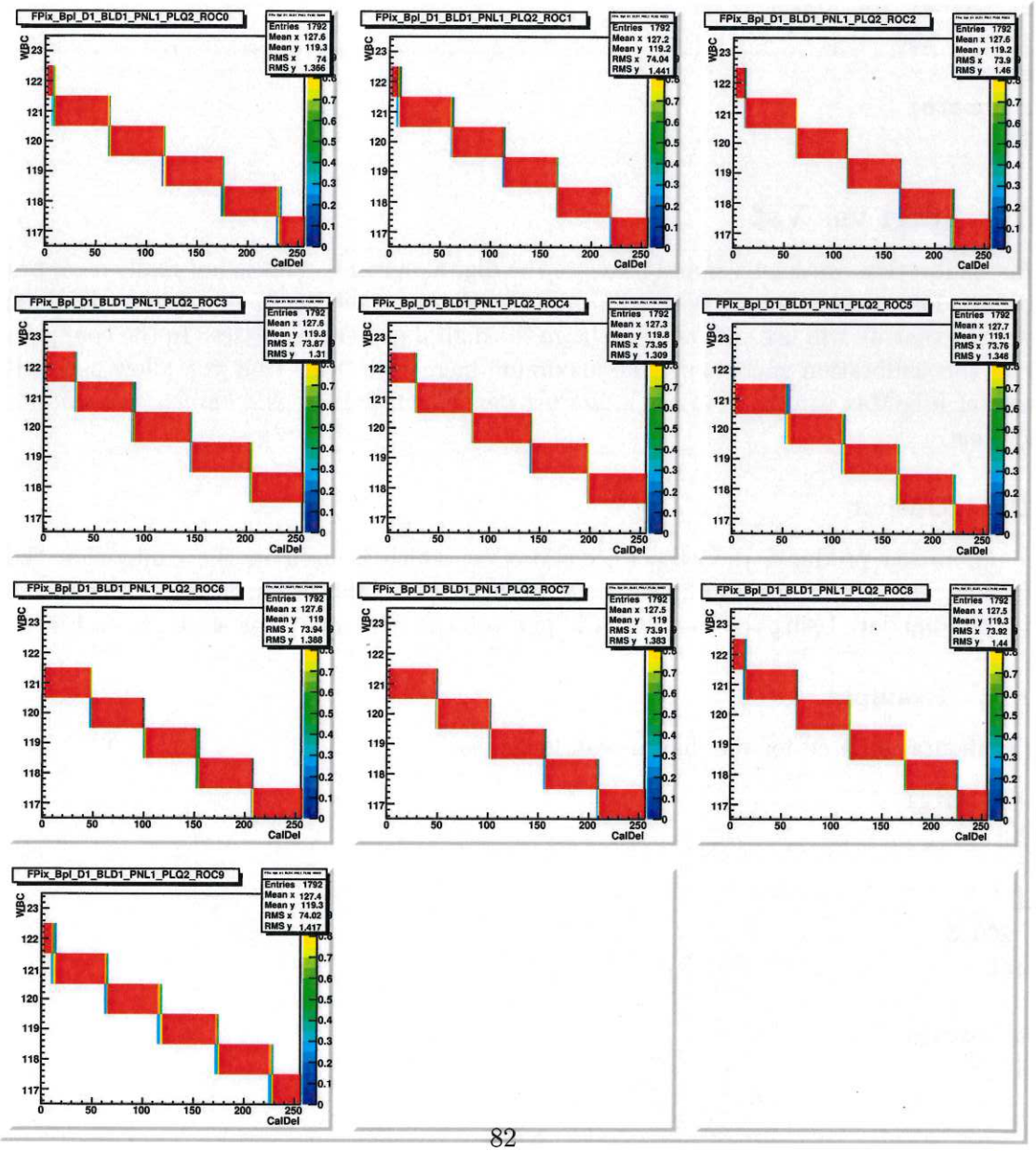
13.14.1 Output

The CalDel calibration produces new ROC DAC settings that update the CalDel settings, all other settings should be left unchanged. The calibration also generates output to display the region of good efficiency in the WBC vs CalDel. The output can be processed to generate plots using the root script `caldel.c`. An example plot is shown in Fig. 18.

13.14.2 Example

An example of a configuration file for the CalDel calibration is shown below:

```
Mode: CalDelCalibration
Rows:
10 | 20
Cols:
10 | 20
VcalHigh
Scan: WBC 117 123 1
Scan: CalDel 0 255 1
Set: Vcal 250
Repeat: 2
ToCalibrate:
+ all
```



Caption is missing (make Fig. smaller?)

If you want to just scan over WBC to find the right WBC value you can use a configuration that sets CalDel to a fixed value:

```
Mode: CalDelCalibration
Rows:
10 | 20
Cols:
10 | 20
VcalHigh
Scan: WBC 117 123 1
Scan: CalDel 100 100 1
Set: Vcal 250
Repeat: 2
ToCalibrate:
+ all
```

13.15 Idigi vs. Vsf

In this 'calibration' we scan Vsf and measure the digital current. This is not really a calibration as we don't determine any settings from this. Rather what we do is to find a maximum Vsf setting that we will use. If Vsf is too large the digital current increases. In the configuration for this calibration you can set the maximum increase in Idigi that you allow using the parameter IdigiMax (units are in mA). *Not yet implemented, there is a hardcoded number of 7 mA now.*

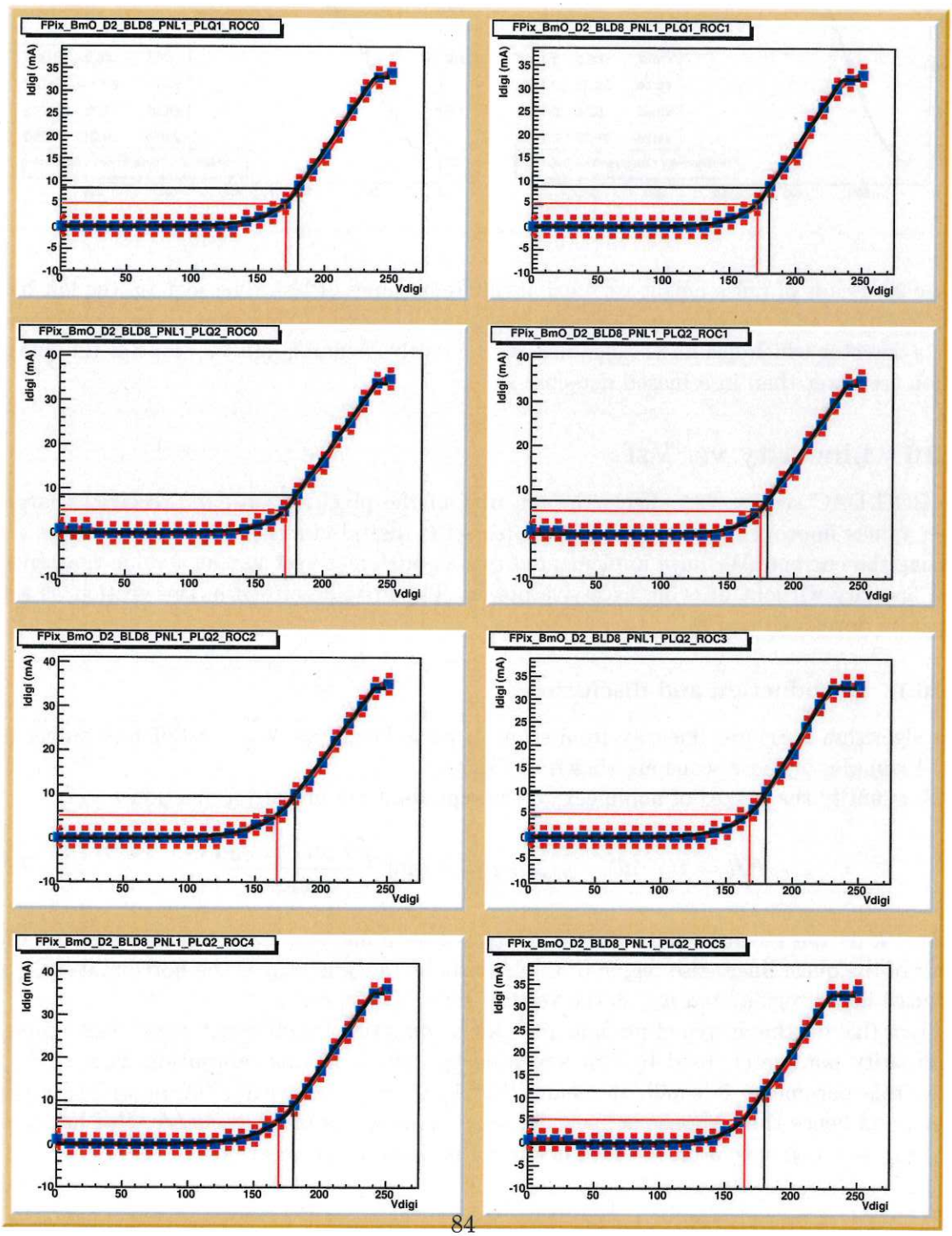
13.15.1 Output

This calibration produces the object PixelMaxVsf which is used in the calibration that determines Vsf to optimize the linearity. In addition this calibration produces output in a file called idigi.dat. Using the root script idigi.c you can generate plots as shown in Fig. 19.

13.15.2 Example

The configuration used for the digital scan looks like

```
Mode:  Idigi
Rows:
Cols:
Vcal:
200 200 5
Repeat:
1
ToCalibrate:
+ all
```



Caption missing. Make plot smaller to fit it.

```
VcalLow
Scan: VHldDel 0 255 5
Set:  Vcal 250
```

`Vsf` is not scanned, and a full scan is specified for `VHldDel`.

To determine both `Vsf` and `VHldDel`, use something like:

```
VcalLow
Scan: Vsf     0 255 3    mix
Scan: VHldDel 0 255 5
Set:  Vcal 250
```

Note that this scan is more time-consuming than running first for `Vsf` and then for `VHldDel`. It also does not allow for the use of interpolated `Vsf` values when determining `VHldDel`. Therefore, it is recommended to run two separate scans.

This calibration has no additional parameters aside from the standard ones.

13.18 Pulse height range calibration

13.18.1 Introduction and discussion

A number of ROC DAC settings affect the scaling of the pulse height signal that is sent to the FED. (This refers not to the actual charge collection, but to the translation of that charge into the signal sent out to the FED.) We want the range of this signal – the difference in recorded pulse height between small and large amounts of charge – to be large. However, the pulse height signal should not go low enough to be confused with the ultrablack level, nor high enough to exceed the FED's dynamic range.

The ROC DAC setting `VIbias_PH` is intended to be used for adjusting the pulse height, so this DAC should be scanned. Other DACs that affect the pulse height are `VOffsetRO`, `VIon`, and `VOffsetOp`. Previous studies found that adjusting `VIbias_PH` and `VOffsetOp`, while fixing `VOffsetRO` and `VIon`, worked well. [5]

13.18.2 Pulse height calibration procedure

The algorithm for this calibration is rather minimal. In the configuration, the user specifies any number of arbitrary ROC DACs to be scanned, and also "low" and "high" amounts of injected charge (i.e. low and high `Vcal` settings). The pulse height is recorded at each scan point for the low and high charges. (If multiple pixels are enabled, their pulse heights are averaged on each ROC.) A scan point is discarded if either the low or high reading falls outside a user-specified range. From the remaining scan points, the point chosen is the one with the largest pulse height difference between high and low charges. These settings are written out.

If either 1 or 2 DACs are scanned, plots are generated of the high PH, low PH, and PH range (difference between high and low) as a function of the DAC setting(s).

Table 12: Optional parameters for pulse height range calibration.

Parameter	Default	Description
minPH	75 = 300/4	Minimum allowed pulse height
maxPH	254	Maximum allowed pulse height

13.18.3 Parameters

The standard parameters in `calib.dat` may be used to control the calibration.

The "`Repeat:`" parameter determines the number of triggers in each step of the scan. The ROC list is used to determine which ROCs are calibrated.

The hits specified in `Rows:` and `Columns:` will be used. At most two hits per pixel pattern may be specified (or alternatively SingleROC mode must be used); otherwise the calibration will abort. The reason for this restriction is that the FED's spy FIFO3 will overflow if each ROC connected to it has more than two hits.

The scan ranges are used to determine which DACs will be adjusted. For example, to scan over `VIbias_PH` and `VOffsetOp`, use something like:

```
VcalHigh
Scan: VIbias_PH 0 255 5
Scan: VOffsetOp 0 255 5
Scan: Vcal 25 255 230
```

Note that a `Vcal` range should also be given. The pulse height range at a given scan point will be calculated from the highest and lowest `Vcal` for which hits were found. If the final scan contains more than two `Vcal` values with hits found, the points in the middle are ignored in determining the pulse height range. Therefore, usual practice is to specify just two `Vcal` scan points. Note that if the lower `Vcal` is below threshold, it will generate no hits; therefore the user should select a minimum that is above threshold. If just one `Vcal` value produces hits on a ROC, the calibration fails on that ROC.

Two optional parameters may also be set – `minPH` and `maxPH`. These specify the minimum and maximum allowable pulse height values; DAC settings that give pulse height outside of this range are rejected. These parameters are specified on the scale of pulse height readings, which runs from 0 to 255. This is in contrast to FED ADC counts, which run from 0 to 1023. (Multiply the pulse height by 4 to get FED ADC counts. For example, a PH of 75 corresponds to 300 FED ADC counts.) These parameters default to 75 and 254; they may be changed in `calib.dat`. The parameters are summarized in Table 12.

13.19 Vana and time walk

This calibration is not yet fully implemented. The description below represents what we intend to do.

The idea is to run a scan of Vana vs CalDel. For a large charge we will assume that the response of different sensors is similar. As for a large charge (the maximum that can be

injected) the raise time is small it is probably a reasonable assumption that we can assume that we have a small variation between ROCs. However, for a smaller charge, like 250 on the low Vcal range, this is not obviously true. This calibration finds the time (absolute) that the Vcal=250 signal is over threshold and also finds the slope dt/dIana, or dt/dVana. Using this information the ROCs can be unified to one time at which a pulse of this strength passes over threshold.

13.20 Trim bit determination

This section outlines an idea for triming. The actual algorithms used are slightly different. The next section describes the analysis tools that are available. This section should be updated.

The algorithm described below is iterative. It uses a linerized model to determine the parameters. Successive iterations of this algorithm should provide better values of the trim parameters.

The threshold (for a pixel) is a function of Vcthr, Vtrim, and Trim.

$$\text{Threshold} = \text{Threshold}(\text{pix}_i, \text{Vcthr}, \text{Vtrim}, \text{Trim}_i) \tag{4}$$

We can expand this as:

$$\begin{aligned}
\text{Threshold}(\text{pix}_i, \text{Vcthr}, \text{Vtrim}, \text{Trim}_i) &= \text{Threshold}(\text{pix}_i, \text{Vcthr}_0, \text{Vtrim}_0, \text{Trim}_{0i}) \\
&+ \text{deltaVcthr}\frac{d\text{Threshold}}{d\text{Vcthr}} \\
&+ \text{deltaVtrim}\frac{d\text{Threshold}}{d\text{Vtrim}} \\
&+ \text{deltaTrim}_i\frac{d\text{Threshold}}{d\text{Trim}}
\end{aligned}$$

Simplifying the notation we have

$$\text{Threshold}(\text{pix}_i, \text{Vcthr}, \text{Vtrim}, \text{Trim}_i) = A_i + B_i * \text{deltaVcthr} + C_i * \text{deltaVtrim} + D_i * \text{deltaTrim}_i \tag{5}$$

Now we want to set all of these equal to some value, ThrTrim.

$$\text{ThrTrim} = A_i + B_i * \text{deltaVcthr} + C_i * \text{deltaVtrim} + D_i * \text{deltaTrim}_i \tag{6}$$

So we have N equations and $N + 2$ unknowns. Generally there are many solutions. But recall that the trim bits only takes on values from 0 to 15. So we need more constraints. We use the following:

- We need to decide how many pixels we include in the triming. I.e. setting Vtrim large enough so that the trim bits can adjust the threshold. Based on the 'initial' parameters

find the rms of the threshold distribution. Find $n-\sigma$ and using an average C_i calculate deltaVtrim needed to include pixels out to $n-\sigma$. So this gives the first parameter. (Use C_i for the smallest Vtrim.)

- Require that the average trim is 7.5, that is

$$\sum_i \text{deltaTrim}_i = \sum_i (7.5 - \text{Trim}_\text{i})$$

This yields a simple constaint:

$$\frac{\text{ThrTrim}}{D_i} = \frac{A_i}{D_i} + \frac{B_i}{D_i}\text{deltaVcthr} + \frac{C_i}{D_i}\text{deltaVtrim} + \text{deltaTrim}_i$$

and now summing over pixels give:

$$\sum_i \frac{\text{ThrTrim}}{\text{D}_\text{i}} = \sum_i \frac{A_i}{D_i} + \sum_i \frac{B_i}{D_i}\text{deltaVcthr} + \sum_i \frac{C_i}{D_i}\text{deltaVtrim} + \sum_i (7.5 - \text{Trim}_\text{i})$$

as we already have determined deltaVtrim we now also get deltaVcthr.

Last all we have to do is to loop over all pixels and calculate deltaTrim_i. This algorithm can be iterated to determined a better approximation of the trim bits.

The derivatives used above has to be determined numerically by running several Scurve runs. Need to describe how to do this; but this is really the core of the implementation.

13.21 Trim bit analysis

This section describes the tools that we have available today for trim bit determination.

13.21.1 Running the threshold analysis

The inputs to the threshold determination is Scurve runs with different parameters. After taking one of these Scurve runs it has to be analyzed to produce a file that contains the thresholds for each pixel.

After taking an SCurve (trim) run (TrimDefaultShort, TrimVcThrShort, TrimVtrimShort, TrimOnShort, TrimOffShort, TrimDefault, TrimVcThr, TrimVtrim, TrimOn, TrimOff), analyze the run using

./bin/linux/x86/PixelAnalysis.exe SCurve ¡runnum¿

The configuration/SCurveAnalysis.xml file needs to contain:

```
<OutputTrimFile      Write="Yes"/>
```

in order to generat the output needed in the trim analysis.

Currently this produces a very long output name. Rename this file:

```
mv $POS_OUTPUT_DIRS/Run_100000/Run_100141/TrimOutputFile_Fed_32-33-34\
35-36-37-38-39.dat $POS_OUTPUT_DIRS/Run_100000/Run_100141/TrimDefault.dat
```

For a 'TrimDefault' run. For the other types of runs listed above they should be named:

```
TrimVcThr.dat
TrimVtrim.dat
TrimOn.dat
TrimOff.dat
```

13.21.2 Quality of trimming

There is a command line tools that looks at the quality of the trimming. It can be run using:

```
[pixelpro@vmepcS2B18-17 test]$ ./bin/linux/x86/PixelTrimAnalysis.exe 100141 | more
Usage: PixelTrimAnalysis.exe runTrimDefault
trimDefault:/nfshome0/pixelpro/TriDAS_build7/pixel/PixelRun/Runs/Run_100000/\
Run_100141/TrimDefault.dat
All ROCs
FPix_Bm0_D1_BLD1_PNL1_PLQ1_ROC0 81 64.1024 2.61066
FPix_Bm0_D1_BLD1_PNL1_PLQ1_ROC1 81 69.239 1.39128
FPix_Bm0_D1_BLD1_PNL1_PLQ2_ROC0 81 74.0942 1.6798
FPix_Bm0_D1_BLD1_PNL1_PLQ2_ROC1 81 65.1567 1.36941
FPix_Bm0_D1_BLD1_PNL1_PLQ2_ROC2 81 65.5652 1.4101
FPix_Bm0_D1_BLD1_PNL1_PLQ2_ROC3 81 77.1227 2.36945
FPix_Bm0_D1_BLD1_PNL1_PLQ2_ROC4 78 111.641 18.149
FPix_Bm0_D1_BLD1_PNL1_PLQ2_ROC5 81 70.3416 1.4199
FPix_Bm0_D1_BLD1_PNL1_PLQ3_ROC0 81 65.2256 1.4215
.
.
.
FPix_BpI_D2_BLD12_PNL2_PLQ3_ROC6 81 68.3048 1.82446
FPix_BpI_D2_BLD12_PNL2_PLQ3_ROC7 77 75.3988 1.54956
FPix_BpI_D2_BLD12_PNL2_PLQ3_ROC8 81 70.205 1.43413
FPix_BpI_D2_BLD12_PNL2_PLQ3_ROC9 81 59.5848 4.26486
Problem ROCs
FPix_Bm0_D2_BLD1_PNL2_PLQ1_ROC3 81 49.3475 3.26693
FPix_Bp0_D1_BLD9_PNL1_PLQ3_ROC2 11 65.2924 43.158
```

Where the criteria for a problem ROC was set in the example above as a ROC with a threshold below 50 or with less than 50 hit pixels (out of 81).

13.21.3 Determining new VcThr settings

If you just want to adjust VcThr to a fixed threshold you can run the following. First, take one 'TrimDefault(Short)' and one 'TrimVcThr(Short)'. In this example these are runs 65624 and 65626 respectively. The 'TrimVcThr' run adjusts VcThr (down) by a fixed amount for each ROC, e.g. 5 DAC settings. *Currently you have to manually make sure when you run the analysis below that this change of VcThr is conistent in the analysis program and in the configuration file. A relatively simple update in the future would be to automatically determine this change from the calib.dat files.*

To run the analysis code

```
./bin/linux/x86/PixelTrimVcThr.exe 10630 65624 65626 > log_TrimVcThr_65624
```

10630 is the configuration key for run 65624. *Again this has to be manually supplied.* The output is a new set of dac settings in the current directory with new VcThr values. The target Vcal is set in `src/common/PixelTrimVcThr.cc` on the line

```
trimROC.setThrTrim(60.0);
```

This is the target threshold in Vcal units.

The produced logfile is huge. One useful thing to do is:

```
grep deltaVcthr log_TrimVcThr_65624
```

to see what the changes in VcThr are for the different ROCs.

13.21.4 Make simple plots of thresholds

To make simple plots of the thresholds do:

```
[pixelpro@vmepcS2B18-17 test]$ root -l
root [0] .L read.c
root [1] .x plot.c
```

plot.c specifies the input file.

13.21.5 Full trim bit determination

To do full trimming, i.e., adjusting the trim bits, Vtrim, and VcThr to trim the ROCs to a preset threshold the following steps are to be followed.

First you need to take five runs:

```
TrimDefaultShort
TrimVcThrShort
TrimVtrimShort
TrimOnShort
TrimOffShort
```

Normally, we use the 'short' configuration files that only use about 50 to 100 pixels on each ROC. This is enough pixels to find the values need for VcThr and Vtrim on a given ROC. In a later step we adjust just the trim bits for the remaining pixels. The TrimDefaultShort run is a run with the 'default' or current settings. The procedure is iterative and the parameters produced in this iteration will be the default parameters for the next iteration. The TrimVcThrShort and TrimVtrimShort runs adjusts the VcThr and Vtrim parameter by a default amount (typically -5 and +10 for VcThr and Vtrim respectively). The last two runs, TrimOnShort and TrimOffShort, turns the trim bits on and off respectively for each of the pixels.

After taking these runs, they are analyzed as described in Sect. 13.21.1. Having done this the new dac and trim bits are determined using

```
./bin/linux/x86/PixelTrim.exe <key> <runTrimDefault> <runTrimVcThr> \
<runTrimVtrim> <runTrimOn> <runTrimOff>
```

Running this command produces new dac settings and trimbits. In addition it produces a file called 'rocder.dat'. The use of this file is described in the next section.

13.21.6 Determining all trimbits

Having carried out the procedure described in the previous section a file called 'rocder.dat' should have been produced. This file contains three numbers for each ROC determined from the analysis described above. First is the derivative of the threshold with respect to the VcThr setting. The second is the derivative of the threshold with respect to Vtrim *This needs to be defined more precisely.* The last is the change in the threshold with respect to changing trim bits, averaged over the pixels on the ROC. The change in threshold with respect to the trimbits is used in the adjustment of the trim bits.

A run is taken using the 'TrimDefault' configuration. This run is analyzed as described in Sect. 13.21.1.

```
./bin/linux/x86/PixelTrimBits.exe <key> <runTrimDefault>
```

Where `<key>` is the global key used and `<runTrimDefault>` is the run number for the trim default run. This produces new trim bit settings. The quality of the trimming can be checked as described in Sect. 13.21.2.

13.22 Emulated physics

The FED allows us to program emulated analog data as inputs to the FPGAs in order for us to test the veracity of the Finite State Machines within them that decode the analog signals. In order to do this, the FED must identify the baseline Black level and all other address levels from the emulated data. This does not involve the FEC at all, as no real data from the ROCs are received. Therefore, this requires independent but similar calibration procedures.

13.23 Baseline with emulated data

13.24 Address levels with emulated data

14 Calibration Procedure

In this section, we describe a way to do a full calibration the detector "from scratch". Detailed descriptions of each algorithm are in the previous section. The procedure must be followed *in order*, because of the various dependencies between settings.

This section is based on our experience calibrating the FPix in 2008-2009 before beam and then again at a colder temperature in 2012 (7.4 C changed to 0 C). Large portions of the procedure were also used for the BPix.

We will try to include tips for common problems.

14.1 Useful Tools, Background Knowledge

This section assumes some background knowledge:

1. **POS:** You will obviously need to know how to run the Pixel Online Software. The results of the calibration are usually summarized in a log file (which one depends on the calibration, unfortunately). Most calibrations produce a root file in the Run directory which you can view in root or with the very useful HistoViewer tool.

2. **Configuration Base:** It is important that you are familiar with the structure of the configuration base and the ways to make new configurations. You should know how to change the aliases using the PixelConfigDBCmd.exe tool.

3. **Scripts to change DACs:** changeSome.pl can be used to shift the value of a particular DAC on a subset of ROCs by a specified value. The script needs a list of all DAC files and a list of ROCs to change. Run it by doing "perl changeSome.pl". There is a similar script, called changeAll.pl, to change all ROCs. Both scripts are in TriDAS/pixel/PixelAnalysisTools/test/additionalTools.

4. **Analyze SCurve/PixelAlive/Gain:** Know how to analyse these calibrations as described in Sec. 13.11.

5. **Loop scripts:** In TriDAS/pixel/PixelAnalysisTools/test/additionalTools, there are root macros that look at the output of the analysis mentioned in the previous point. These are run by doing e.g. "root -l SCurveLoop_FPix.C".

14.2 Starting Point

The calibrations we run only affect a subset of the detector's many configurable settings. Many settings are fixed to initial recommendations found in specifications or early studies

```
GridSize:
1
Tests:
10
Commands:
0
```

Tip: If your efficient band is very thin, it might be because the bias settings of the DOH (DOH_Ch0Bias_CLK and DOH_Ch1Bias_Data), which read back the result of the programming test, are too low. The plot can also look bad if the fibers are dirty.

14.5 Basic Signal Properties

Before you can optimize the performance of the ROC, you have to adjust the very basic properties of the signal so that the FED can interpret the data. There is a little bit of a bootstrapping problem here because even these calibrations need to start with a somewhat recognizable signal. Sometimes, you will have to do a preliminary calibration of setting "A" so that you can calibrate "B" before you do the final calibration of "A". You may even have to manually tune a setting at the beginning, depending on your starting point.

14.5.1 FEDBaseline

We run this calibration all the time. It has to be run any time there is a significant change in the temperature of the AOH. At P5 in 2010-2011, we determined that the automatic baseline correction could cope with changes of 1-2 degrees C. We monitored the size of the automatic baseline correction with the PixelMonitor tool and reran the FEDBaseline calibration whenever the size of the correction surpassed 100 units.

The calibration is also useful as a debugging tool. The calibration is fast and can often provide the feedback you need to assess the status of the detector.

We target a black level of 450 ADC units.

Here is the `calib.dat` file we used at P5:

```
Mode: FEDBaselineWithPixels
SingleROC
Rows:
0 | 1 | 2 | 3 | 4 | 5 |
6 | 7 | 8 | 9 | 10 |
11 | 12 | 13 | 14 | 15 |
16 | 17 | 18 | 19 | 20 |
21 | 22 | 23 | 24 | 25 |
26 | 27 | 28 | 29 | 30 |
31 | 32 | 33 | 34 | 35 |
36 | 37 | 38 | 39 | 40 |
```

```
29 | 30 | 31 | 32 |
33 | 34 | 35 | 36 |
37 | 38 | 39 | 40 |
41 | 42 | 43 | 44 |
45 | 46 | 47 | 48 |
49 | 50 | 51
VcalHigh:
100 100 5
Repeat:
10
ToCalibrate:
all
```

14.6 Getting Hits

The rest of the calibrations require the use of the charge injection feature of the ROC. For the injected charge to be readout as a hit, it has to cross threshold (which involves amount of charge and the threshold) and be validated be the trigger (which involves the timing of the injection). You can get hits by doing the VcThrCalDelFIFO3 calibration, which chooses compatible values for VcThr and CalDel. This calibration will not choose the optimal settings; rather it produces initial settings that will allow you to collect the data you need to find the final ones.

Here is the `calib.dat` file we use at P5:

```
Mode: ThresholdCalDelay
Rows:
10 | 20
Cols:
10 | 20
VcalHigh
Scan: VcThr 0 255 8
Scan: CalDel 0 255 8
Set: Vcal 40
Repeat: 10
ToCalibrate:
all
```

14.7 Address Levels

Now that you have hits, you can take an address level calibration. This will allow the FEDs to decode the hit related data. We run on a subset of pixels ~~of pixels~~ to save time. The subset is chosen to exercise the various address level transitions.

`all`

You should now have SCurve.data for VcThr-20, 0, +2, +4, +6, +8, +10, +12, +14, and +16. Proceed by analyzing the data using the usual PixelAnalysisTools to get the absolute thresholds (so analyze both WBCs). Analyze the FPix and the BPix separately to produce one root file for each. The root files are used in the following steps.

Now, go to the TriDAS/pixel/PixelAnalysisTools/test/additionalTools/ directory. You will make use of several ROOT macros and perl scripts in this directory.

First, look at the VcThr-20 run. This run was taken at a high threshold where there should really be no problems due to thresholds being too low, so you can use it to determine how many pixels on each ROC out of the 81 you injected with charge are working. To do this, run SCurveLoop_BPix.C and SCurveLoop_FPix.C on the VcThr-20 ROOT files with some special options set: `PRINT=0`, `print=1`, `plot=0`, `hold=0`, `useBase=0`. Also, make sure to change `fFile` to the correct root file and `out_name` to something that indicates which run it is (e.g. "minus20"). Run by doing `root -l -b -q SCurveLoop\_FPix.C`.

You should now have two "..._print.dat" files (one for FPix and one for BPix) that contain the number pixels to expect from each ROC. Change `printedFile` in SCurveLoop_BPix(FPix).C to open the BPix(FPix) file and set the following options: `PRINT=0`, `print=0`, `plot=0`, `hold=0`, `useBase=1`. Now when the macros are run, the number pixels expected from a ROC will be taken from the VcThr-20 run.

Now run SCurveLoop_BPix(FPix).C on the remaining SCurve runs (VcThr+0,2,4,6,8,10,12,14,16). Remember to change `out_name` and `fFile` for each run. Combine the FPix and BPix "..._fail.dat" files for each run by doing e.g. `cat plus6_BPIX_fail.dat plus6_FPIX_fail.dat > plus6_fa`

Create a copy of the Default dacs. Use the changeSome.pl script to shift the VcThr of every ROC included in the configuration (so, not noInit or noAnalogSignal ROCs) by +14 (two units less than the farthest you went, as a safety margin). iterate.pl will tell you how much the thresholds of each ROC must be raised away from these settings in order for them to be above their failure point. First change the input files in iterate.pl to the "..._fail.dat" files you just created. Then run it by doing "perl iterate.pl" (sorry the code is so ugly – this was Ben's first perl script). This will create lists of ROCs grouped by the required VcThr shift. The lists are put into text files with names the indicate the required VcThr shift. For example, `sub6.dat` contains the list of ROCs that must have 6 subtracted from their VcThr value. Use changeSome.pl with all of these "subX.dat" files to apply all of the necessary shifts.

You now have a the first guess at the minimized thresholds. You should now rerun an SCurve with the new VcThr values (and no shift). Some ROCs will fail even though according to the runs you took earlier they should pass (I guess this is because the ROCs are not fully independent from one another... or they are unstable near the failing point). Analyze the SCurve using SCurveLoop_BPix(FPix).C and correct any ROCs that fail (i.e. that are listed in the "..._fail.dat" files). To correct them, I recommend shifting their VcThr value by -4. Repeat this until everything passes.

Next, you can try running on a different subset of pixels (it's best to shift at least the double columns used) to make sure all the ROCs still pass. You'll have to run at VcThr-20

```
27 |
40 |
...
51
VcalLow
Set: Vcal 150
Repeat:
10
ToCalibrate:
+
 all
```

Note: The "..." are just to save paper – you should use every pixel. You should use the Default masks with this one (to have whatever is in Physics in and have noisy pixels masked)! VCal 150 was carefully chosen to be above the threshold of all pixels without being so large that it adds unnecessary cross-talk in the charge injection.

14.12 Pulse Height Optimizations

14.12.1 Vsf

Here is the `calib.dat` file we use at P5:

```
Mode:  VsfAndVHldDel
Rows:
3 12
Cols:
14
VcalLow
Scan: Vsf     0 255 3    mix
Scan: VHldDel 0 255 255
Set: Vcal 150
Repeat: 10
ToCalibrate:
all
```

Tip: If a ROC fails, it may be because it's threshold is too low. Decrease VcThr by 5-10 units and try again.

14.12.2 VHldDel

Here is the `calib.dat` file we use at P5:

```
Mode:  VsfAndVHldDel
```

```
Rows:
12
Cols:
14 25
VcalLow
Scan: VHldDel 0 255 5
Set: Vcal 250
Repeat: 40
ToCalibrate:
all
```

Note that the value the calibration chooses depends somewhat on the VCal used. THe value of 250 (low scale) has been used for a long time and seems to work well enough.

14.12.3 PHRange

Here is the `calib.dat` file we use at P5:

```
Mode:  PHRange
Parameters:
minPH 70
maxPH 235
Rows:
12
Cols:
14 25
VcalHigh
Scan: VIbias_PH 0 200 10
Scan: VOffsetOp 0 150 5
Scan: Vcal 17 167 150
Repeat: 4
ToCalibrate:
all
```

Tip: If a ROC fails this calibration, it may be because the range of the scan is not wide enough. We try to keep the scan narrow and with only a few points because this calibration uses a lot of memory for some reason.

14.12.4 Gain Calibration

The last step is to take a gain calibration. We usually take a short one on a subset of pixels to verify that everything is okay before doing the final one on all pixels that takes more than six hours.

is used for making connections to DCS to learn the status of the power supplies (the topic of this section). The Tracker PSX server is used to send DCU data to DCS. Note that the Tracker PSX server is also used by the strip tracker, and for performance reasons it has proven critical to use the independent Pixel PSX server for making the connections to the power supply status.

15.2 PixelDCSFSMInterface

The PixelDCSFSMInterface provides an interface between the PSX server and our other pixel xdaq applications. Changes in the states of each power supply modules are sent in real time from the PSX server to the PixelDCSFSMInterface application. The PixelDCSFSMInterface summarizes this information into one state per half cylinder/shell (thus there are 8 summarized states, 4 in the FPix and 4 in the BPix). The logic for making the summary is as follows:

- If all A4603 power supplies in the summary group are ON (HV and LV is ON), then the summarized state reflects this
  - A4603 power supplies in state `HVMIXED` (one HV channel is on and the other is either off or ramping), are treated as if they were ON.[9]
- When forming the summary, if an entire power supply has been marked as noInit or noAnalogSignal in the detconfig, then that power supply is ignored when forming the summary[10].
- If one or more power supply in the summary group has the HV in the off state, then the summarized state treats the whole group as though it is in the state with HV off and LV on.
- If one or more power supply in the summary group has the LV off, then the summarized state treats the whole group as off.

When the summarized state of one of the half cylinders/shells changes, the PixelDCSFSMInterface sends a SOAP message to the relevant PixelTKFECSupervisor/PixelFECSupervisor.

The nodes to be readout from DCS are defined in `PixelDCSInterface/xml/interface.xml`.

[9]This configuration was deployed while planning for the first beam data, because of a desire to have the high-voltage treated as ON, even if the FPix inner radius HV was OFF.

[10]Initially the code would only ignore the power supply if the status was marked as noInit (because a noAnalogSignal power group still required LV power). However, for the partial configurations prepared for the first beam running, we wanted to be able to ignore a power supply even if it was in state noAnalogSignal.

15.3 Use of DCS information by the supervisors

SOAP messages are sent from the PixelDCSFSMInterface to the PixelTKFECSupervisors and PixelFECSupervisors in real time whenever the summarized state of the power supplies changes. The Supervisors receive these messages asynchronously (in other words, they handle them immediately in a separate thread). When a supervisor receives an update about the power status, the only action taken is the update member data that stores the summarized state. No other action is taken at that moment.

The PixelFECSupervisors receive and track the states of the low and high voltage of the A4603 power supplies. The low voltage has three possible states: LV_OFF, LV_ON_REDUCED, and LV_ON. The high voltage has two possible states: HV_OFF and HV_ON.

The PixelTKFECSupervisors receive and track the states of the low voltage provided by the A4602 power supplies. There are two possbile states: LV_OFF and LV_ON.

15.4 Configuration

The progress through the DAQ's FSM states and transitions for initialization and configuration is envisioned as follows:

- The FEC Supervisor notices that the state of the HV has changed since the configuration. Therefore the VcThr and chip control register DACs are set to their nominal values. Because of this settings change, the user will see a large change in the amount of digital current drawn by the FPix.
- The calibration proceeds as normal.

15.9 Notes on hardcoding of information

Because the xdaq software (POS) and DCS use different naming schemes and different granularity, several translations must be made in order for POS to communicate with DCS. Some of these translations are encoded in the `interface.xml` file, while others are hard-coded in the logic of the PixelDCSFSMInterface.

Specifically, the xml file defines which ROGs[11] exist, and which PVSS datapoint name belongs to which ROG[12]. The xml file also defines the POS applications that need to be notified in case the state of a ROG changes.

The hardcoded logic in PixelDCSFSMInterface defines which ROG correponds to which pieces of the detector (which ROCs are supplied by which power supply).

In principle, this information could be instead contained in a database table with roughly 5 fields:

Field in database	Example value
POS name	FPix_BmI_D2_BLD1
DCS name	CMS_TRACKER:PixelEndCap:PixelEndCap_BmI:PixelEndCap_BmI_D2:PixelEndCap_BmI_D2_ROG1
datapoint name	cms_trk_dcs_10:tkPg_CAEN \CMS_TRACKER_SY1527_5 \branchController05 \easyCrate3 ...
SOAP connection (TKFEC)	PixelTKFECSupervisor instance 2
SOAP connection (FEC)	PixelFECSupervisor instance 2

With a table such as this, we could eliminate both the xml file and the hardcoding.

16 Procedures to follow when hardware components are replaced

This section documents the procedures that are to be followed when off detector hardware components are replaced that affects the online software and calibrations. *This section is not complete!*

16.1 FED board

As the FED decodes the analog levels it will be required to redo several calibrations to check the optical connections after replacing a FED. The FED firmware version has to be validated. The following calibrations should be run

[11] readout groups

[12] These datapoints are used to indicate when POS has configured the ROCs on a ROG.

- ResetTBM: sent via the TTC. Availabe as a button in the PixelSupervisor GUI. Implemented and verified to work.
- ResetCCU: causes the TKFECSupervisors to issue the command `resetPlxFec()`. Availabe as a button in the PixelSupervisor GUI. Implemented. Sends resetPlxFec via the FecSoftware.
- PIA resets: a menu of resets available from the PixekTKFECSupervisor GUI, implemented by passing a value to the FecSoftware command `testPIAResetfunctions`
  - roc: value = 0x1
  - aoh: value = 0x2
  - doh: value = 0x4
  - res1: value = 0x8
  - res2: value = 0x10
  - fpixroc: value = 0x2A (bits 1, 3, and 5); this resets the TBM and ROCs
  - fpixdevice: value = 0x15 (bits 0, 2, and 4); this resets the portcard devices (Delay25, DOH, AOH, DCU)

  The PIA resets are hard resets using hardware lines. In the FPix they go from the CCU parallel output lines to reset lines on the portcard. One is fanned out to all the portcard devices. The other is routed to the TBM and ROCs through the adapter board.[13]

[13] https://hypernews.cern.ch/HyperNews/CMS/get/pixelOnlineSW/1155/1/1.html

A FED phase and delay calibration

This section will describe some aspects of determining the so called 'phase' and 'delay' of the FED. The sample plots shown in this section are taken from runs on the FPIX pilot run detector. this was done using a version 4 FED with the Aug. 22, 2007 firmware version.

In order to get the transparent data to 'look' OK I had to send an LRES before each trigger. Will Johns and I had long email exchanges trying to understand this. I think that the final conclusion was that the FED state machine gets confused when the input data is non standard. In the scans over phases and delays there are certain settings that are invalid, i.e., that you try to read out the ADC before the data is available. At the end of this section I will discuss two issues that I noticed in looking at the transparent data.

For the above reason we also turn off the automatic baseline correction to avoid it getting confused if you seend non valid data or if you don't have correct address levels.

In these tests we take 10 triggers for each delay and phase setting. I'll show plots, as in Fig. 23, where for each of the two possible phases the profile histogram shows the adc value as a function of clock+delay/16. The third plots just shows the two plots on top of each other. We see that there are certain values of the delays that produce garbage as you try to read out the ADC before the data is available. To identify the invalid combinations of phase and delay I calculate the rms for the different phase and delay settings in the first 20 clock cycles, i.e. before the TBM header arrives. Based on the rms distribution it is seen that for phase=0 that we got invalid data with delays of 1, 2, and 3, while for phase=1 the delays of 10, 11, and 12 generate invalid data. From now on I will reject these combinations. (In the plots I set the adc value to 0.)

After rejecting the invalid phase and delay combinations we now have a signal that looks as what is shown in Fig. 24. The remaining signal still has artifacts due to when the adc latches on to the signal such that it looks like the signal is not read out in order. This is corrected by 'reordering' the data within a clock. I shifted the phase=0 events backward by 3 delay steps and the phase 1 events forward by 3 delay steps. This makes the signal continuous as shown in Fig. 25.

The last step is to align the data so that the phase=0 and phase=1 data overlaps. I do this by shifting the data for the events with phase=1 by 10 delay steps. The final result is shown in Fig. 26. For the bins in the plot where there is data from both phase=0 and phase=1 we see that there is excellent agreement. We also not that the signal undershoots in a transition from black to ultrablack and that it overshoots on the reverse transition.

To determine a best delay I look at the combined data - where I average the bins that has both phase=0 and phase=1 contributions - and calculate a difference for bin i by taking the difference between bin $i-1$ and $i+1$. I calculate this difference for the 25 MHz clock cycles from clock 27 to 100. I add the magnitude of these differences for all bins that corresponds to the same delay. I pick the smallest such sum of differences for the different delay as the optimal choice. If more than one phase is allowed I pick the choice that is farthest from the invalid delay for that phase.

A new alogorithm has been implmented that determines the phase to be right at the

edge of the transition from black to ultrablack in the TBM trailer. You can still run the old algorithm by setting the parameter 'oldMode' to 'Yes'.

Looking at the 8 channels that were connected to the pilot run detector I find that the results are very consistent. The phase is either 0 or 15 in the (aligned delay) for the raw delay setting this corresponds to 2 or 3 for phase=1.

Open questions: Is the same set of phases and delays illegal on all FEDs and channels? Also, are the shifts that are needed to time-order the data the same on all FEDs and channels? Looking at more data will resolve this. It is easy to test that the data in the two phases are compatible by forming a χ^2.

The code was written as a root macro. I will 'transplant' this code into xdaq so we can run it automatically to produce the phase and delay settings needed.

However, looking at this I observed an odd feature. This is illustrated in Fig. 27. In Fig. 28 is a close up of the region with the noise. Here you can see that there is 'wiggle' in the data that is large compared to the rms, as indicated by the error. This wiggle seems to persist for many clock cycles after the TBM trailer. However, it was not present in the black data before the data train. It seems to first start in the TBM trailer when the UB signal is generated. With the pilot run detector we are using 8 channels (1, 2, 3, 4, 7, 8, 9, and 10). I see the same feature on almost all channels, though they are not as strong on channel 1, 7, and 9.

B Calibration Organization

The calibrations all share a similar format.

- The PixelSupervisor controls a loop over 'events' and coordinates the activity of the FED, FEC, TKFEC, and TTC supervisors.
- Each calibration needs to do some initialization before processing the event data.
- During event processing data is acquired and processed.
- After all event data is taken it is analyzed and the results are saved.

This section will describe a *proposal* for how to formalize this process. A first step in this direction was taken when the calibration classes were broken out from the the PixelSupervisor and the PixelFEDSupervisor. This first step was almost trivial as it just copied the code out to separate classes. There are a few more changes we should make that will allow us to run these jobs in workloops. Among other things, this will allow us to look at the progress of the calibrations from the web browser.

B.1 PixelSupervisor calibration code

We change the `PixelCalibrationBase` class to have the interface below.

Where does this belong to? Not mentioned any where in the text.

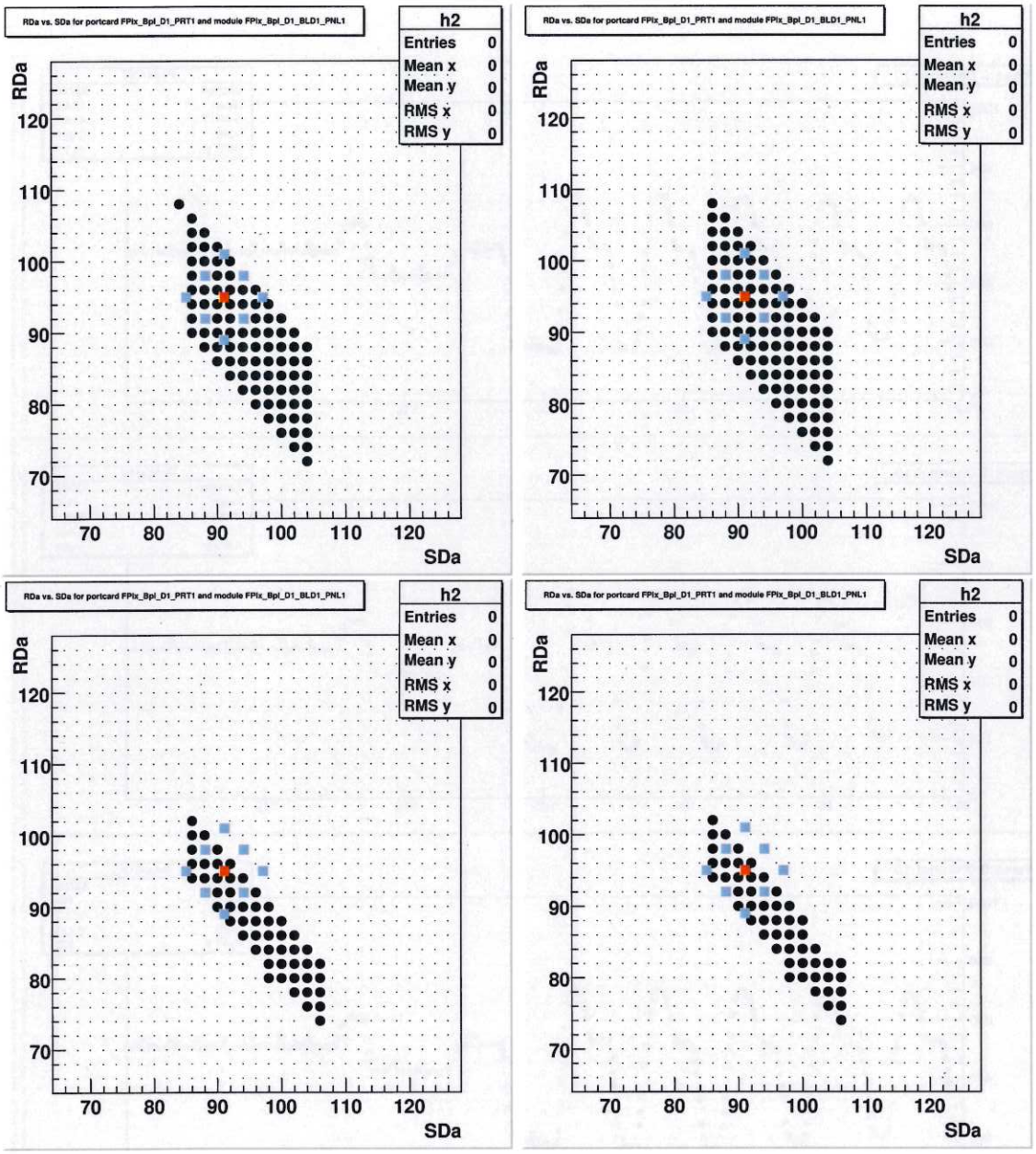


Figure 29: This figure shows the results of scans for delay25 settings on the Cornell test stand. The upper left plots shows (large black dots) the region for which the return data was valid when sending a ROC command (CalPix). The upper right plot shows the valid region when sending a TBM command (tbm speed). The lower left shows the region of success when sending a roc init and the lower right shows the region of success with a roc trim load command. As is seen, the working region is smaller for the long commands. The red and blue points indicates the algorithm used to select the operating point. Need to check with Jennifer what this is doing.

```
class PixelCalibrationBase : public PixelSupervisorConfiguration,
    public SOAPCommander {
 public:

  PixelCalibrationBase( const PixelSupervisorConfiguration &,
   const SOAPCommander& soapCommander  );

  virtual ~PixelCalibrationBase(){};

  virtual void beginCalibration();

  virtual bool calibrationEvent();

  virtual void endCalibration();

 protected:

  private:

  // PixelCalibrationBase Constructor
  PixelCalibrationBase(const SOAPCommander& soapCommander);

};
```

The method `begin()` should do any initialization that is needed before processing any triggers. The method `event()` is called repeatedly until it returns true, this indicates that the calibration has come to an end. Then the `end()` method is invoked to perform any analysis of the acquired data.

More details on what these methods needs to do will be discussed below.

B.2 PixelFEDSupervisor calibration code

Similarly, I suggest a more general structure for the calibration code that runs in the PixelFEDSupervisor.

```
class PixelFEDCalibrationBase : public PixelFEDSupervisorConfiguration,
    public SOAPCommander {
 public:

  PixelFEDCalibrationBase( const PixelFEDSupervisorConfiguration &,
   const SOAPCommander& soapCommander );

  virtual ~PixelFEDCalibrationBase(){};
```

In addition it would be nice to take out the code from the PixelSupervisor and the PixelFEDSupervisor that select the calibration type. One way of doing this is to create a 'factory' for calibrations in the PixelCalibrations package. This factory would accept a string from the calibration mode and create the calibration object and return it to the PixelSupervisor and the PixelFEDSupervisor. Besides simplifying the code in these supervisors (they will not need to #include all the calibrations header files) it will also separate out dependencies.

C Outstanding tasks and improvements

Remove ?

I will divide these task into three different categories. The first is algorithm/calibration development. The second is 'operations', the third is code improvements.

C.1 Calibration development

- Implement algorithm for time walk. Some of the tools are available. Need to understand how to analyze the data and set parameters.
- Adjust AOH gain setting. Need to discuss a strategy for this.
- Need to fix the delay25 calibration; it wraps around.
- Linearity adjustment. Steve started to look at Vsf and Vhlddel.
- Adjust the ROC analog signal offset and gain. Want to keep the lowest level well above UB, and the maximum near 255 (or 1024).
- In the address level determination if the highest level hit 1023 it will not be detected. Should allow this.

C.2 Operation improvements

- Migrate calibration algorithms to use root objects, like histograms. This will allow us to write out root files with the calibration results as well as monitoring the progress of calibrations.
- Tools to view the calibration results should be migrated to look at root files.
- Deploy database.
- Use filebased interface and aliases to nor write over old files. This is basically done. But should modify the writing of files so that all files in the configuration is written out again. Otherwise it will be to hard to used. This is not very complicated, but not sure what is the best strategy.

not

be

- Deploy the error logger to make sure that we send relevant and not to verbose messages.
- Have to run with the FEDSpySupervisor.
- Run with the power-on sequence. This is work in progress now and a first try will take place during the week of Jan. 28, 2008.
- We should properly initialize the CCU.
- We should try out the schemes for reconfiguring a CCU ring to drop a CCU.
- Is it possible that we can program ROCs on one panel, or blade, such that we can burn the fuse on the adapter board?
- We have to try out the 'popcon' feature for calibrations.
- Embed the 'private' words in the FED data. Have code example now from Will. Need to modify the rawToDigi code to unpack this information.

C.3 Code improvements

- Move DCU readout workloop to configure transition.
- Calibration algorithms should run on separate threads. This is mostly implemented now. Some calibrations still need to migrate to executing the calibration in 'steps', and not just as on call.
- should use dynamic cast instead of static cast.
- Further cleanup in calibrations. A number of the calibrations contains a large amount of duplication.
- Some code like the FEDInterface and FEDCard is too verbose. Need to review printouts.
- There are some exceptions we need to catch. For example if we try to talk to a nonexisting FED base address we should catch the exception and not just crash.
- Should the supervisor applications be self updating like the trigger supervisor applications? This would then allow to automatically update progress status during a calibration and handle when a calibration is complete.
- The ROC status should be respected. This would, e.g., allow us to not generate a failed calibraton when one of the ROCs can't generate hits.
- More consistency checking needed; in particular for the AOH initialization. Should only initialize portcards that are used.

References

[1] The pixel online wiki is available at https://twiki.cern.ch/twiki/bin/view/CMS/PixelOnlineSoftw

[2] *PSI46 Pixel Chip*, version 2. Available at http://cms.web.psi.ch/roc/psi46v2.pdf.

[3] Token Bit Manager 05a Chip Documentation, version 1.13. Available at http://www.physics.rutgers.edu/ bartz/cms/tbm2/0.25u/index.html.

[4] RCMS Documentation (Level 1 Function Managers, updated 17 Feb 2009). Link avaiable at http://cmsdoc.cern.ch/cms/TRIDAS/RCMS/Docs/Manuals/manualIndex.html.

[5] Katarina Gromova, ETH Zurich, Semester Thesis, *Adjustment of the Readout Range of the CMS Pixel Detector* (2008). Available at http://www.phys.ethz.ch/ ursl/home/v0/projects/p06.pdf.