

pXar User Manual

Urs Langenegger

*Paul Scherrer Institute
CH-5232 Villigen-PSI, Switzerland*

January 15, 2021

Abstract

The document provides an overview of the pXar software, the testing, calibration, and data acquisition framework used for the CMS pixel detector phase-1 upgrade project. It describes the installation (with a complete tutorial for a UBUNTU installation) and running of the software and provides a user manual for operation of PSI46 and proc600 Chips and Modules with the Digital Testboard (DTB).

Furthermore, many of the implemented chip and module tests are discussed. A tutorial is provided how users may integrate their own test procedures and algorithms into the pxar framework by writing their own user tests and scripts.

Contents

1	Introduction	4
2	Installing pxar	6
2.1	Dependencies	6
2.1.1	CMake	6
2.1.2	C++ compiler	6
2.1.3	PERL	6
2.1.4	ROOT	6
2.1.5	libusb 1.0	6
2.1.6	FDT2XX / FTDI library	7
2.1.7	Python, Cython, Numpy	7
2.2	Downloading the source code	7
2.3	Configuring via CMake	8
2.4	Compilation	9
2.5	DTB firmware	9
2.6	Running the program	10
3	Hardware overview	11
3.1	DTB	11
3.2	Readout chips	11
3.2.1	PSI46digV2	11
3.2.2	PSI46digV2.1	12
3.2.3	proc600v3	12
3.2.4	proc600v4	13
3.3	Basic aspects of ROC behavior	13
3.3.1	Thresholds and implications	13
3.3.2	Thresholds and trimming	13
3.3.3	Readout length	13
3.3.4	Hits and pulse height	14
3.4	TBM: Token bit manager	14
4	Software design overview	15
4.1	API and hardware abstraction layer	15
4.2	Software Information	15
4.2.1	pXarcore data structures	15
4.2.2	pXarcore flags	15
4.2.3	pXarcore test and utility functions	15
4.2.4	pXar test and utility functions	15
4.3	User interface and tests	16
5	Tutorial	18
5.1	Installation	18
5.2	First steps	20

6 Tests	21
6.1 Pretest	21
6.1.1 Pretest:programroc	22
6.1.2 Pretest:setvana	23
6.1.3 Pretest:findtiming	23
6.1.4 Pretest:findworkingpixel	23
6.1.5 Pretest:setvthrcalcdel	24
6.2 Alive	25
6.2.1 Alive:alivetest	26
6.2.2 Alive:masktest	26
6.2.3 Alive:addressdecodingtest	27
6.3 Bump bonding tests	27
6.3.1 BB	28
6.3.2 BB2	29
6.3.3 Discussion	30
6.4 Trim	30
6.4.1 Trim:trim	32
6.4.2 Trim:trimbits	34
6.5 Scurves	36
6.6 Ph (Pulse height optimization)	37
6.7 GainPedestal calibration	41
6.8 Threshold/efficiency scans and maps (DacScan and DacDacScan)	43
6.9 Readback test and calibration	44
6.10 Xray (VCal calibration)	46
6.11 HighRate	48
6.12 IV (leakage current test)	49
6.13 FullTest	51
6.14 Additional tests	52
References	53

1 Introduction

The **pXar** framework provides all required software tools and user interfaces to study, test, and readout various CMOS pixel detector chips and modules developed and built at PSI. It provides

- A low-level interface [1] to the digital test board (DTB), illustrated in Fig. 1, connecting the hardware [the device under test (DUT)] to a computer.
- A graphical user interface (GUI) for the hardware configuration, the execution of tests, the display of the test results, and the monitoring of various hardware parameters (*e.g.* current levels). The GUI is illustrated in Fig. 2.
- Configuration of the DUT, accessible in the ‘h/w tab’ (the green tab in Fig. 2).
- Tests for the study and qualification of the pixel detector modules. Each test, together with its subtests, is accessible in its own ‘tab’ in the GUI.
- Test sequences chaining together various tests for the production qualification.
- A command-line interface allowing low-level controlling of the attached hardware.

pXar runs on UNIX-like computers (macOS and Linux) and Windows. The low-level interface is quite self-contained, the high-level part relies heavily on ROOT, both for the GUI and the processing and analysis of the test results.



Figure 1: The digital test board used as data-acquisition board for CMS pixel chips and modules (for the phase-1 upgrade and also for legacy hardware). [2]

It should be noted that **pXar** is not multi-threaded. This implies that significant wait cycles occur when large DTB readouts are active. From time to time, the GUI checks whether a

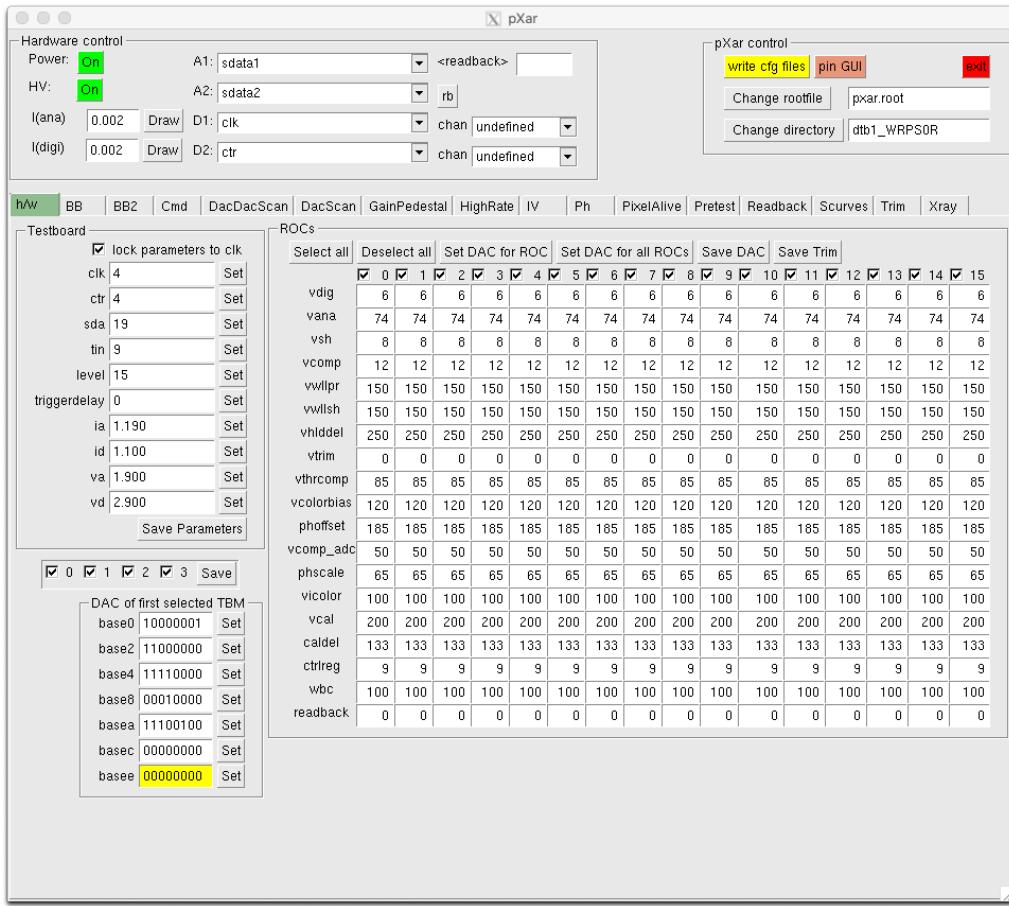


Figure 2: The graphical user interface to pXar. The ‘hardware’ tab (h/w) is open and shows the various hardware configuration possibilities. The other tabs to its right provide access to specific tests and subtests. The section ‘Hardware control’ in the top left provides a monitoring of the analog and digital currents and the mapping of specific signal to the DTB LEMO outputs. The section ‘pXar control’ allows to exit the program and apply some changes.

test has been interrupted (using the Stop button on the test tabs). However, most of the wait cycles are due to activity at the lower levels (DTB, USB transfer speed limitations, data unpacking). It is not foreseen to convert pXar to a multi-threaded setup.

2 Installing pxar

2.1 Dependencies

`pxar` has relatively few dependencies on other software, but some features do rely on other packages. To configure the `pxar` build process, the CMake cross-platform, open-source build system is used. The libusb and FTDI libraries are needed to communicate over USB with a DTB.

Note that in section 5 a complete and detailed description is provided how to install all dependencies and program parts.

2.1.1 CMake

In order to automatically generate configuration files for the build process of `pxar` both compiler and platform independent, the CMake build system is used.

CMake is available for all major operating systems from <http://www.cmake.org/cmake/resources/software.html>. On most Linux distributions it can usually be installed via the built-in package manager (aptitude/apt-get/yum etc.) and on OSX using packages provided by e.g. the MacPorts or Fink projects.

Make sure to have CMake version 2.8.12 or above.

2.1.2 C++ compiler

The compilation of the `pxar` source code requires a C++ compiler and has been tested with GCC, Clang, and MSVC (Visual Studio 2012 and later) on Linux, OS X and Windows.

2.1.3 PERL

Any standard UNIX computer should have PERL installed.

2.1.4 ROOT

The graphical user interface of `pxar`, as well as a few command-line utilities, use the Root package for histogramming. It can be downloaded from <http://root.cern.ch> or installed via your favorite package manager. Make sure Root's `bin` subdirectory is in your path, so that the `root-config` utility can be run. This can be done by sourcing the `thisroot.sh` (or `thisroot.csh` for csh-like shells) script in the `bin` directory of the Root installation:

```
source /path/to/root/bin/thisroot.sh
```

ROOT 6 is supported. It is strongly recommended to build ROOT from the sources with the *very same* compiler which will be used to compile `pxar`. Most likely your compilation will not work if this requirement is not fulfilled.

2.1.5 libusb 1.0

In order to communicate with the DTB, the libusb library is needed. Please make sure that libusb is properly installed.

On Mac OS X, this can be installed using Fink or MacPorts. If using MacPorts you may also need to install the `libusb-compat` package. On Linux it may already be installed, otherwise you should use the built-in package manager to install it. Make sure to get the development version, which may be named `libusb-1.0-dev` or `libusb-1.0-devel` instead of simply `libusb-1.0`.

2.1.6 FDT2XX / FTDI library

The DTB features a FTDI USB chip which needs the a special driver to communicate over USB. There are two options available. The FTDI library is a open source driver for the FTDI chips and is shipped with most Linux distributions and can usually installed via the package manager. The package is usually called `libftdi-dev` or similar. However, this driver has shown performance problems in the past and should only be used as fallback in case the proprietary driver does not work for some reason.

The proprietary driver provided by the FTDI company is called FTD2XX. Its binaries and header files can be fetched from <http://www.ftdichip.com/Drivers/D2XX.htm>. Make sure to pick the correct version for your operating system and system architecture. The driver library and header files should be placed in the usual install locations of your system, e.g. for a Linux distribution usually `/usr/local/lib` and `/usr/local/include` are a good choice since those directories are not supervised by the package manager but the files are discovered by CMake. The files from the package are required:

```
ftd2xx.h  
WinTypes.h  
libftd2xx.so
```

It might be necessary to create a symlink pointing to `libftd2xx.so` (without the additional version numbers).

2.1.7 Python, Cython, Numpy

In case you want to use the Python interface to the pXar core library to use the Pythin Command Line Interface or write short and simple scripts to program the detector and perform tests, you need Python 2.7, the python libraries, as well as Cython (version 0.19 or later) and the Python-Numpy package. All should be available for Linux (via the package manager), OS X and Windows.

2.2 Downloading the source code

The pXar source code is hosted on github. The recommended way to obtain the software is with git, since this will allow you to easily update to newer versions. The latest version can be checked out with the following command:

```
git clone https://github.com/psi46/pxar.git pxar
```

This will create the directory `pxar`, and download the latest development version into it. An alternative to the https-based download is using ssh

```
git clone git@github.com:psi46/pxar.git pxar
```

For this to work you need to set up ssh-keys. There are descriptions how this is achieved on github.

If you already have a copy installed, and want to update it to the latest version, you do not need to clone the repository again, just change to the `pxar` directory use the command:

```
git pull
```

to update your local copy with all changes committed to the central repository.

Alternatively you can also download a zip file from <https://github.com/psi46/pxar/archive/master.zip>.

For production environments (e.g. test stands and probe stations) we strongly recommend to use the latest release version. Use the command `git tag` in the repository to find the newest version and type e.g.

```
git checkout tags/v2.8.0
```

to change to version v2.8.0. Note that this setup is for production environments, not for code development. Code development takes place should branch off into feature branches, starting from the HEAD of the master branch.

2.3 Configuring via CMake

CMake supports out-of-source configurations and builds – just create and enter the `'./build'` directory and run CMake, i.e.

```
mkdir build && cd build  
cmake ..
```

CMake automatically searches for all required packages and verifies that all dependencies are met using the `CMakeLists.txt` script in the main folder. By default, only the central shared library, the main executables including the graphical user interface (GUI) are configured for compilation. The corresponding settings are cached, so that they will be again used next time CMake is run.

Some of the optional packages or configuration options include:

`pxarui`: The main executable and GUI. This requires the ROOT libraries and header files to be installed.

`tools`: Additional auxiliary code for `pXar` is built. This source code resides in the `./tools` folder of the repository. Use this at your own risk.

`HV` (for directly controlling a HV source meter from `pXar`):

```
cmake -DBUILD_HVSUPPLY=Keithley2410 ..  
cmake -DBUILD_HVSUPPLY=Keithley237 ..  
cmake -DBUILD_HVSUPPLY=Iseg ..
```

At the option for a `Keithley2410` works and is maintained (cf. section 6.12).

`dtbemulator` (useful for code development if no DTB is attached)

```
cmake -DBUILD_dtbemulator=ON ..  
pxarui (in case you can live without the user interface)
```

```
cmake -DBUILD_pxarui={ON,OFF} ..
```

To install the binaries and the library outside the source tree, you need to set the `INSTALL_PREFIX` option, e.g.

```
cmake -D INSTALL_PREFIX=/usr/local ..
```

to install the executables into the `bin` and the library into `lib` sub-directories of `/usr/local`.

If you ever need to, you can safely remove all files from the build folder as it only contains automatically generated files. Just run

```
cd build  
rm -rf *
```

to start from scratch.

2.4 Compilation

Assuming that you start from the `pxar` directory, do

```
cd build  
cmake ..  
make [-j4] install [VERBOSE=1]  
cd ..
```

This will install by default the library into `pxar/lib` and the executables into `pxar/bin` (both directories are sub-folders of your local `pxar` folder).

2.5 DTB firmware

To properly operate the DTB, you also need to have the matching firmware loaded onto that device. The firmware can be obtained from <https://github.com/psi46/pixel-dtb-firmware/tree/master/FLASH> and then loaded onto the DTB (its FPGA) with the following statements (assuming that you are in directory `pxar`):

```
git clone git@github.com:psi46/pixel-dtb-firmware.git  
./bin/pXar -f ../../pixel-dtb-firmware/FLASH/FLASHFILE
```

where `FLASHFILE` should be replaced with the name of the most recent version. The download will take a while. Follow the instructions at the end: Wait until all 4 LEDs turn off and *power-cycle* the DTB.

2.6 Running the program

To run the GUI with a proc600 module (as for the layer-1 replacement production run of 2020/2021), we recommend doing (see below for more details)

```
mkdir data
cd data
./main/mkConfig -d testModule -t TBM10D -r proc600v4 -m
./bin/pXar -d testModule -g [-v WARNING|QUIET|...]
```

For other hardware configurations see below for more detailed instructions.

Without the GUI, using a 'standard' ROOT C++ macro

```
./bin/pXar -d testModule \
-c '../scripts/singleTest.C("DacScan", "DacScan.root")'
```

is also possible.

3 Hardware overview

This section provides a short overview on the hardware. It is neither complete, exhaustive nor authoritative. The intention is to help an outsider with no prior knowledge of the connected hardware.

3.1 DTB

See ref. [2] for a detailed description of the DTB. Note that not all DTBs are equivalent. Some have internal drivers replaced, which improves the readout stability. There is unfortunately no way to identify these DTBs from the outside.

3.2 Readout chips

The original PSI46v2 ROC, with analog readout, was designed to cope with a hit rate of 80 MHz expected for the initial LHC design instantaneous luminosity of $\mathcal{L} = 10^{34} \text{ cm}^{-2}\text{s}^{-1}$. After the phase-1 upgrade of the LHC, the instantaneous luminosity reaches $\mathcal{L} = 2 \times 10^{34} \text{ cm}^{-2}\text{s}^{-1}$ and requires a new ROC to cope with the substantially increased readout bandwidth.

3.2.1 PSI46digV2

The first version of the digital version of the ROC, with full digital signal readout, is based on a *column drain* architecture and can digest a hit rate of 230 MHz. It aimed at the following improvements compared to the previous analog PSI46v2 ROC [3]:

- A higher rate capability because of larger L1 latency buffers (24 time-stamp and 80 data buffers) and an additional ROC readout buffer stage (63×23 bit FIFO) to lower readout-related data losses through readout throttling
- The readout was changed to a digital scheme to substantially increase the readout bandwidth
- A lower operational threshold to increase the detector lifetime
- Lower current consumption and miscellaneous operational improvements (an additional metalization layer to increase the uniformity for lower thresholds and less cross-talk; an optimized faster comparator for a lower in-time threshold and less cross-talk)

The main issues with this ROC version included the power-up reset signal (this made programming the ROCs impossible), a stacked triggers problem (triggers could not be stacked), and double-column freezing when operating the ROC without reset for long periods of time. Other less critical issues were an inverted pixel address, occasional errors in the ROC header, the VTHRCOMP DAC range was too wide, and setup issues for the analog readout chain (especially at higher irradiation doses).

3.2.2 PSI46digV2.1

The second version of the digital version of the ROC fixed the main issues of the first version and provided in addition the following improvements [4, 5]:

- **FIXME**

The ROC psi46digV2.1respin is basically the same chip with a single mask change addressing a minor issue with the analog pulse height (this depended on VDIG and the pixel position in the column drain). A simplified sketch of the PSI46digV2.1 is provided in Fig. 3.

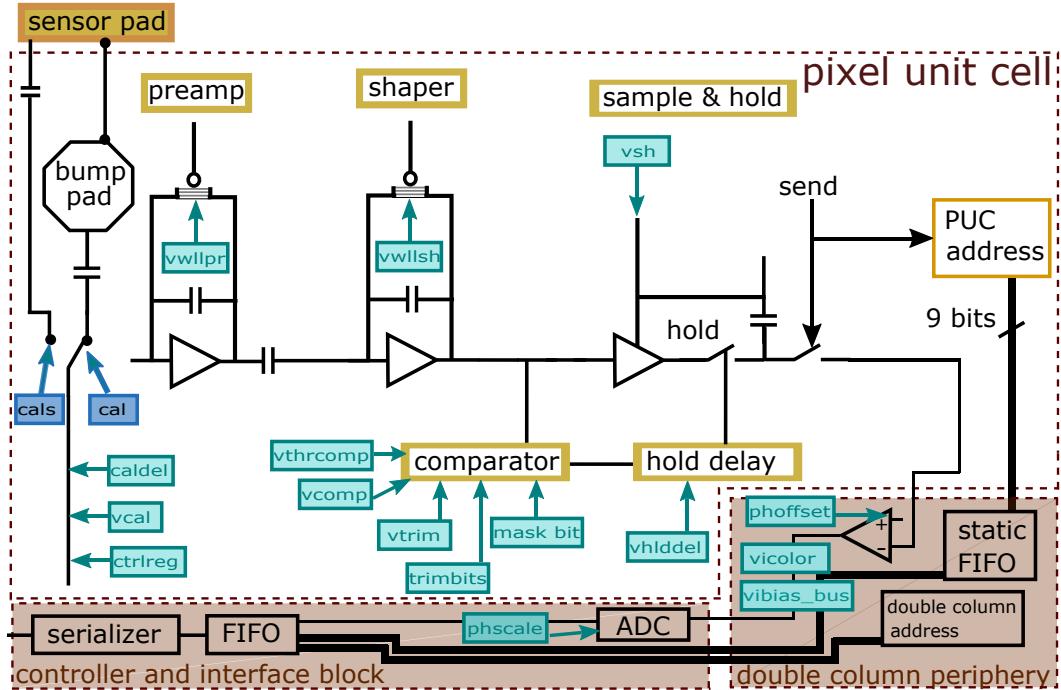


Figure 3: Simplified sketch of the PSI46digV2.1 ROC with the DAC programming indicated.

3.2.3 proc600v3

To address the increased rate of 600 MHz at the innermost detector layer, a readout chip design was required as the PSI46dig architecture could not accommodate the increased pixel hit rate. The ‘pixel ROC’ (proc) was designed with a completely new core and a new concept of *dynamic cluster column drain*, where 2×2 pixels are transferred to the periphery instead of single pixels. The commissioning of the proc600v3 ROC proceeded on an extremely tight schedule. The proc600v3 addressed several critical issues of its predecessor:

- larger cross-talk than expected (implying higher thresholds, but fixable in software)
- hit inefficiencies at low ($\approx 10^{33} \text{ cm}^{-2}\text{s}^{-1}$) and high ($> 10^{34} \text{ cm}^{-2}\text{s}^{-1}$) instantaneous luminosity due to loss of synchronization between the time-stamp and databuffers (logic glitch)
- a large non-uniformity of the pixel pedestals.

The ROC proc600v3 worked well in layer 1 of the CMS pixel detector during LHC run 2.

3.2.4 proc600v4

The replacement layer 1, built for LHC run 3, the ROC proc600v4 is used. The improvements over the previous version include [6]

- Fine tuning of comparator timing to optimize the efficiency plateau width
- Shift of the VTHRCOMP DAC range to ensure a low startup power consumption
- Lower cross talk and lower stable threshold with a modified injection capacitor layout, improved shielding of various sensitive nodes, and layout changes of the analog busses in the double columns
- Less pedestal spread with improved column readout amplifier

This ROC has no serious issues affecting its performance.

3.3 Basic aspects of ROC behavior

In the following we summarize a few basic aspects of the ROC the user should be aware of.

3.3.1 Thresholds and implications

The per-ROC threshold set by VTHRCOMP is inverted in the sense that a large numerical value for VTHRCOMP implies a small (closer to zero or the noise level) threshold.

If the ROC is operated with small thresholds (closer to the noise level), the noise fluctuations will fill the readout buffers in the periphery and as a consequence the real hits (from calibration pulses or external particles) most likely will get lost. This results in a low efficiency.

3.3.2 Thresholds and trimming

There is one DAC VTHRCOMP per ROC that controls the threshold level across the entire ROC. To allow a fine-tuning of this, each pixel has four trimbits which allow the lowering of the global VTHRCOMP setting for the pixel. The scale of the modifications is given by the DAC value of VTRIM. Symbolically, the per-pixel threshold Thr is then determined by

$$\text{Thr} = \text{VTHRCOMP} - (16 - \text{TB}) \times \text{VTRIM} \quad (1)$$

where TB stands for the value encoded in the four trimbits (0–15). The trimbits have inverted logic, i.e. a large value of TB corresponds to a small effective threshold subtraction. Enabling the trimbits one by one, from the least significant trimbit to the most significant, implies setting TB = 14, 13, 11, 7.

3.3.3 Readout length

FIXME

3.3.4 Hits and pulse height

FIXME

3.4 TBM: Token bit manager

In the following, we summarize the different TBM versions used in the modules for the phase-1 upgrade of the pixel detector:

- TBM10C for the original layer 1.
- TBM10D for the replacement of layer 1 for LHC run 3
- TBM09C for the (original) layer 2
- TBM08C for the (original) layer 3
- TBM08C for the (original) layer 4

4 Software design overview

4.1 API and hardware abstraction layer

4.2 Software Information

This section summarizes information useful for the understanding of the `pXar` source code and for the writing of user (test) code (see section 4.3).

4.2.1 pXarcore data structures

write about `pixel`

4.2.2 pXarcore flags

4.2.3 pXarcore test and utility functions

`pXarcore` test functions implement a loop over all ‘active’ pixels with specific actions. They normally return a vector of pairs containing a DAC value and the corresponding `pixel` structs.

A few short remarks on the most commonly used functions:

- `getEfficiencyVsDAC`
- `getEfficiencyMap` Returns a vector of `pixel` containing the numbers of hits (not the efficiency).

4.2.4 pXar test and utility functions

- `getIdxFromId`
- `scurveMaps` is a general utility function that fills and analyzes hitmaps and PH maps and returns a configurable amount of resulting 1D and 2D histograms. It is defined and implemented in `PixTest`:

```
std::vector<TH1*> scurveMaps(std::string dac, std::string name, int ntrig = 10,
                                int daclo = 0, int dachi = 255,
                                int dacspersstep = -1, int ntrigperstep = 1,
                                int result = 15,
                                int ihit = 1,
                                int flag = FLAG_FORCE_MASKED);
```

The meaning of its parameters is as follows.

- `dac` defines which DAC is scanned to obtain the s-curve.
- `name` provides a name which will be a defining element of the returned histograms (both in their name and title).
- `daclo` and `dachi` allow restricting the range of the DAC to be scanned to speed up the test.
- `dacspersstep` allows splitting the DAC scan into different steps. This parameter was required for intermediate `pXarcore` releases and should be set to the default `dacspersstep = -1`.

- `ntrigperstep` determines the statistics for the measurement of hitmaps and PH maps.
- `result` encodes bitwise how much information is returned.

<code>result &</code>	<code>returns</code>
0x1	threshold maps
0x2	significance (width) maps
0x4	noise maps
0x8	also dump distributions for those maps enabled with 1,2, or 4
0x10	dump ‘problematic’ threshold histogram fits
0x20	dump all threshold histogram fits

s-curves provide threshold maps with associated information (width or noise) by fitting, for each pixel, an error function to the scanned DAC histogram illustrated in Fig. 4. The function fitted to the DAC histogram is defined as

$$f = p_3 \times \left(\text{TMath::Erf}\left(\frac{x - p_0}{p_1}\right) + p_2 \right), \quad (2)$$

where $\text{TMath::Erf}(x) = (2/\sqrt{\pi}) \int_0^x \exp^{-t^2} dt$.

The resulting threshold is given by the value of p_0 , the DAC position where the 50% threshold is crossed. The resulting width of the threshold rise is determined as $1/(\sqrt{2} \times p_1)$. The noise threshold, by default, is determined as the last DAC value where the bin content is larger than 50% of the maximum bin content. The noise threshold is also used to flag pathological s-curve histograms: If the fitted threshold is below the first DAC value, the noise is set to -2, and if the fitted threshold is above the last DAC value, the noise is set to the last DAC value.

It should be noted that no fit is attempted in two cases:

- * if an exact step function is found in which case the threshold is set to the corresponding DAC value and $p_1 \equiv 100$ (i.e. $\sigma \approx 0.01$). The noise threshold is determined as above in the default case.
- * if no plateau is found, the parameters are set such that p_0 is one bin width below the lower boundary of the histogram and $p_1 \equiv 100$.
- `ihit` determines whether a hitmap (`ihit = 1`) or a PH map (`ihit = 2`) is measured.

- `efficiencyMap`
- `trimHotPixels`
- `maskHotPixels`

4.3 User interface and tests

It is straightforward to add your own tests to `pxar`: you need to write the source code and provide the configuration for `testparameters.dat`.

- The source code for all user tests are located in `pxar/usertests`. It is recommended that the name of your class starts with `PixTest`. Provide the class definition in a header

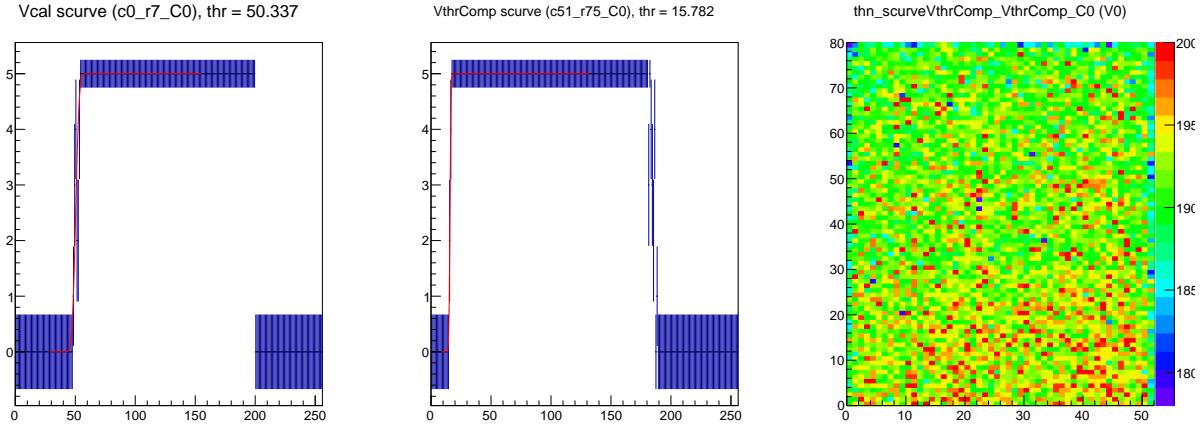


Figure 4: Illustration of (right) a VCAL threshold s -curve, obtained with `ntrig = 5` and the fitted error-function overlayed. The rising edge at VCAL = 50 corresponds to the VCAL threshold (this ROC has been trimmed to VCAL = 50). The falling edge at VCAL = 200 is an artifact of the scan going only until VCAL = 200. (middle) VTHRCOMP threshold s -curve. Here the falling edge around VTHRCOMP = 180 corresponds to the noise level. (right) Noise level map for a VTHRCOMP scan.

file , whose name (without the extension `.hh`) corresponds to the class name. Provide the implementation in a source file with extension `.cc`. Following these recommendations allows that the ‘glob’ in `usertests/CMakeLists.txt` file picks up all user test classes in `pxar/usertests`, and there is no need to manually insert your filenames into `usertests/CMakeLists.txt`.

- To instantiate your test class in the GUI, and to provide the initial configuration of the test parameters, provide a section for `testparameters.dat`.

After inserting the header and implementation files into `pxar/usertests`, go to your build directory, re-run the `cmake` command, and compile. The `cmake` step will (re)build the `pxar/usertests/PixUserTestFactory.cc` file, which will make your test class visible, for instance in the GUI.

5 Tutorial

If you encounter any problems that you cannot solve, please check in <https://twiki.cern.ch/twiki/bin/viewauth/CMS/Pxar#FAQ> for a similar problem and solution, post a question to [pixel-psi46-testboard hypernews](#), or open an issue at <https://github.com/psi46/pxar/issues/new>.

5.1 Installation

The following packages/programs are required:

- C++ compiler
- ROOT 6. It is very strongly recommended to have ROOT compiled locally with the same C++ compiler used for the pXar installation.
- git
- cmake (at least version 2.8.4)
- USB (header files)
- FTDI drivers

The following is the complete history and a tutorial where to obtain and how to install all required software on an UBUNTU virtual machine.

```
# -- essentials
sudo apt-get install git
sudo apt-get install cmake
# -- compilers and required devel packages
sudo apt-get install dpkg-dev make g++ gcc binutils libx11-dev libxpm-dev \
                    libxft-dev libxext-dev
sudo apt-get install gfortran libssl-dev libpcre3-dev \
                    xlibmesa-glu-dev libglew1.5-dev libftgl-dev \
                    libmysqlclient-dev libfftw3-dev \
                    graphviz-dev libavahi-compat-libdnssd-dev \
                    libldap2-dev python-dev libxml2-dev libkrb5-dev \
                    libgs10-dev libqt4-dev
# -- build ROOT
cd /opt
sudo git clone http://github.com/root-project/root.git source-root
cd source-root
sudo git checkout -b v6-16-00 v6-16-00
cd ../
sudo mkdir root-061600
cd root-061600/
sudo cmake ../source-root
sudo ./configure
```

```

sudo make -j4 install
source /opt/root/bin/thisroot.csh

# -- USB setup:
sudo apt-get install libusb-1.0-0-dev

(with sudo) create /etc/udev/rules.d/11-testboard.rules with the following contents:
# for access to DTB:
SUBSYSTEM=="usb", ATTR{idVendor}=="0403", ATTR{idProduct}=="6014", \
ATTR{manufacturer}=="PSI", GROUP="plugdev", MODE="0664"

(reboot)

# -- FTDI
wget http://www.ftdichip.com/Drivers/D2XX/Linux/libftd2xx1.1.12.tar.gz
mkdir ftd2xx1.1.12
mv libftd2xx1.1.12.tar.gz ftd2xx1.1.12/
cd ftd2xx1.1.12/
tar zxvf libftd2xx1.1.12.tar.gz
cd release/
cd build/x86_64/
sudo cp lib* /usr/local/lib
sudo chmod 0755 /usr/local/lib/libftd2xx.so.1.1.12
sudo ln -sf /usr/local/lib/libftd2xx.so.1.1.12 /usr/local/lib/libftd2xx.so
cd ../..
sudo cp ftd2xx.h WinTypes.h /usr/local/include/

# -- and finally PXAR
(setup ssh-keys for github: see https://help.github.com/articles/generating-ssh-keys/)

git clone git@github.com:psi46/pxar
git clone git@github.com:psi46/pixel-dtb-firmware
cd pxar
mkdir build
cd build
cmake ..
make -j4 install
cd ../main
./mkConfig -d testModule -t TBM10D -r proc600v3 -m
../bin/pXar -d testModule -g -v DEBUG

:-)

```

Note: It is important to remember the `install` part in the command `make -j4 install`
If on a virtual machine you run into problems with USB, e.g.

CRITICAL: Could not find any connected DTB. pxar caught an internal exception:

Could not find any connected DTB.
or
Detached kernel driver from selected testboard.
then the first proposal is to 'play' around with the 'Devices -> USB':

- select the DTB entry
- maybe add it to the USB Device Filters list
- if that is not sufficient, look at its details (in my case I had to switch to 'remote')

After a power-down of the VM, I had to re-select the DTB entry in the USB devices list (the first step above, the rest persisted).

5.2 First steps

The first steps after connecting a module to the DTB and turning on the HV (to anything between -80 to -150 V) are:

- create all pXar configuration files for the specific hardware. For the most recent modules this can be done with

```
cd .../pxar && mkdir data && cd data
./main/mkConfig -m -t tbm10d -r proc600v4 -d v4
```

- invoke the pXar GUI with

```
./bin/pXar -d v4 -g
```
- change into the Pretest tab (cf. Fig. 5), click on
 - **setvana** to properly configure the power consumption of the module
 - (possibly **findtiming**) to configure the readout timing. Most likely, this step is not necessary.
 - **setvthrcmpcaldel** to properly configure the injection of test pulses.
- change into the Alive tab (cf. Fig. 9), click on **alivetest** to get a hitmap of all ROCs on the module.
- if the hitmap is filled with high efficiency for all ROCs, go back to the Pretest tab and hit **savedacs** and **savetbms**. If not, start debugging your hardware.

6 Tests

In the `pXar` code base, the source code of ‘normal’ tests is located in the `tests` subdirectory. The class name for each test is composed of the test name with the prefix `PixTest`, e.g. `PixTestPretest`, and the source code is split into `.hh` files for the declaration and `.cc` files for the implementation. The configuration file `testParameters.dat` file (in the directory created by `main/mkConfig`) controls, which test appears in the GUI and allows the configuration of the default parameters. Tests not appearing in `testParameters.dat` do not show up in the GUI. `testParameters.dat` can be edited manually, both adding and removing tests. However, another invocation of `main/mkConfig` will overwrite the manual edits.

In addition, user-defined test classes are available in the `usertests` subdirectory. It is sufficient to put the `.hh` and `.cc` files of a user-test into the `usertests` subdirectory to get the code included into the library. It is the responsibility of the user to update the configuration file `moreTestParameters.dat` (or to add the corresponding lines into `main/mkConfig`). This is described in more details in section 4.3.

Tests and calibration methods can affect the setting of the DUT. Not all methods, however, persist the DUT settings to the configuration files. Depending on the context, this is the desired behavior: For instance, after trimming the DUT to a uniform threshold setting, most users would expect that the DUT remains in that state and that the configuration files are persisted. On the other hand, running a `DacScan` should not result in the DAC value remaining at the maximum value used in the `DacScan`. In the descriptions below we try to clarify which tests alter the behavior of the DUT and which leave the DUT in the same as before.

The illustrations of the tests start with its GUI tab and then the most important results of the subtests are shown with smaller plots (obtained by hitting `print` in the test tab).

6.1 Pretest

The goal of the `Pretest` is to quickly check the very basic functionality of the DUT and determine its proper timing setup. Rapid subtests allow a proper configuration of the most basic DACs (VANA, VTHRCOMP, CALDEL) required before further tests can be performed, checks that the DUT can be programmed, and checks that its readout is functioning. The `Pretest` should be the first test performed when a new DUT is connected. After the `Pretest` has been successfully run, non-faulty pixels with respond to test pulses from charge injection and their readout can be decoded.

The following subtests are available in the `Pretest`:

- `programroc`: checks whether the DUT ROCs can be programmed
- `setvana`: sets VANA for each ROC such that the desired analog current consumption is reached.
- `settimings`: configures the readout timing for ‘old’ TBM (TBM08A and TBM08B). This entry is just for reference, normally this subtest is not part of the `Pretest` tab (it has been removed from `main/mkConfig`).
- `findtiming`: configures the readout timing for all other TBMs.
- `findworkingpixel`: determines a pixel that is responsive for all ROCs on the DUT.

- `setvthrcmpcaldel`: determines a working point for VTHRCOMP and CALDEL within the high-efficiency region.

To assess whether the DUT is properly connected and works (in the sense that it can be programmed and read out), it is sufficient to run `setvana` and `setvthrcmpcaldel` and to make sure that these tests were successful. The `doTest` subtest runs the `programroc`, `setvana`, `findtiming`, `findworkingpixel`, and `setvthrcmpcaldel` subtests. After successfully completing this sequence, it saves all DACs for the ROCs and the TBM. If a problem occurred in any of the subtests, the `doTest` subtest is aborted, except if `ignoreproblems` is activated.

The Pretest GUI tab is shown in Fig. 5. All subtests of the Pretest are now described in sequence.

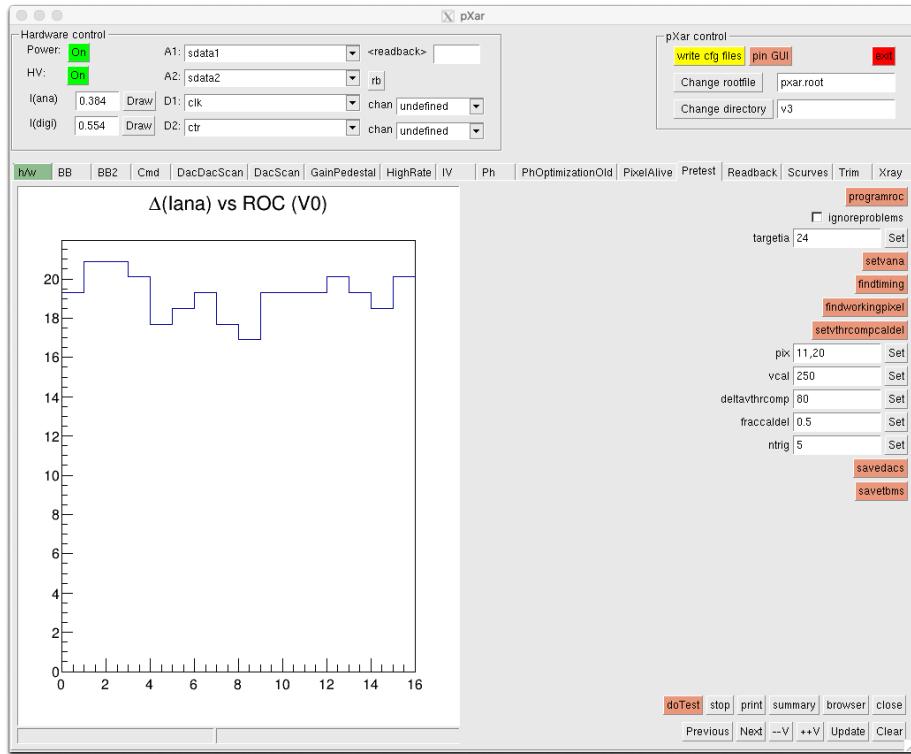


Figure 5: The Pretest GUI tab. The display in the embedded canvas corresponds to the result of a `programroc` subtest for a module. The results of the subtests are illustrated in Figs. 6–8.

6.1.1 Pretest:programroc

The purpose of the `programroc` subtest is to check whether the programming of the **DACs** works for all ROCs of the DUT. This is achieved by setting all VANA to 0. After a delay of 2 sec, the analog current is measured to obtain a reference. For each ROC, the VANA is set to its initial value, waiting for 1 sec, and measuring the analog current. The difference between the two analog currents is calculated and histogrammed. A difference smaller than 5 mA is considered problematic and results in `fProblem = true` (which may lead to the termination

of a `FullTest`, if this subtest is invoked from within a `FullTest`). After this calculation, `VANA` is again set to 0, and the next ROC is investigated.

6.1.2 Pretest:setvana

The important `setvana` subtest configures `VANA` for each ROC such that the analog current consumption reaches the specified target `targetia` for each ROC, nominally `targetia = 24mA`. To obtain a current measurement per ROC, correcting for offsets, the following procedure is applied.

- The initial `VANA` values are cached, and then `VANA = 0` is programmed for each ROC.
- In this state, the analog current is measured after a delay of 0.1 sec. A baseline analog current offset, corresponding to the other ROCs, is determined by scaling the obtained value with 15/16.
- In a loop over all ROCs, the following optimization is performed.
 - the initial (cached) `VANA` value is re-programmed and the analog current is measured (with a delay of 0.1 sec)
 - the difference Δ between `targetia + 0.1mA` (where 0.1mA is an extra margin) and the analog current attributed to the ROC (obtained from the difference of the analog current minus the baseline offset) is determined.
 - iteratively `VANA` is modified such until the difference Δ is either smaller than 0.25mA or more than 10 iterations have been attempted or `VANA` reaches the physical boundaries (0 or 255). The modification uses a ‘slope’ of 6, obtained from 255DACs/40mA, to speed up the convergence.
- As final step, a check is performed that the analog current drops more than 15mA if a single ROC is set to `VANA = 0`. The resulting drops are printed and used to potentially set `fProblem = true`.

6.1.3 Pretest:findtiming

This is magic code and you are advised to consult the source code for more information. In Fig. 7 the phase scans and the chosen operating points are illustrated.

6.1.4 Pretest:findworkingpixel

This subtest is used to set `fPIX` with a column/row value for a working pixel in all ROCs. This is achieved by determining the efficiency vs. `VTHRCOMP` and `CALDEL` for a hard-coded list of possible pixel choices: (12,22), (5,5), (15,26), (20,32), (25,36), (30,42), (35,50), (40,60), (45,70), (50,75). The pixels on this list are a purely random choice. A ROC is qualified as working if its efficiency is more than 50% with a `CALDEL` plateau with a width of more than 30DAC and a `VTHRCOMP` plateau of more than 50DAC.

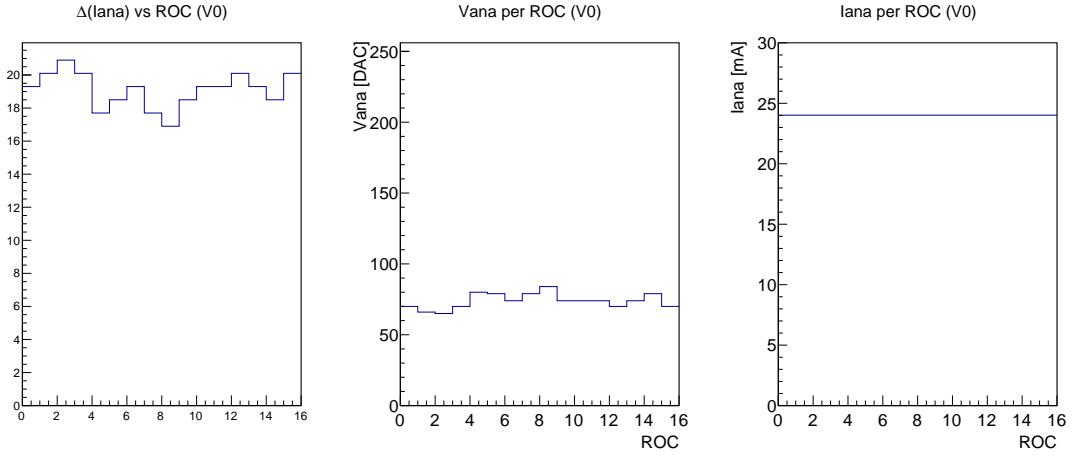


Figure 6: Illustration of the `programroc` and `setvana` subtests. (left) Analog current consumption drop when switching off a ROC. This is not at 24 mA because the offset is not corrected for (this is simply a check that the ROCs can be programmed). (middle) VANA settings per ROC and (right) analog current consumption determined by switching off a ROC with the proper VANA settings and correcting for the offset.

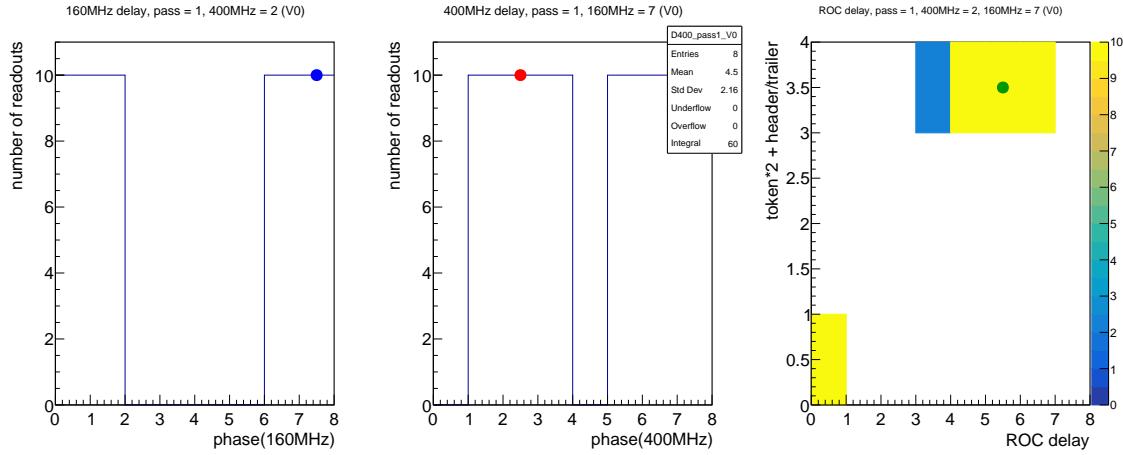


Figure 7: Visualization of the `findtiming` subtest. The plots show the second pass results for (left) the 160 MHz phase scan, (middle) the 400 MHz phase scan, and (right) the 2D scan for the ROC delay and the token/header/trailer setting. In all cases the chosen values are indicated with colored filled circles. (The first pass is also available and is labeled pass = 0.)

6.1.5 Pretest:setvthrcalcdel

This important subtest configures VTHRCOMP and CALDEL for normal calibration operations. In the low-range regime, VCAL is set to 250 and the single pixel specified in `fPIX` (possibly set in `findworkingpixel1`) is used in each ROC to obtain the efficiency vs. VTHRCOMP and CALDEL covering the full DAC range for both DACs. The configurable parameter `parntrig` allows to change the number of triggers (the default is `parntrig` = 5). In the first step of the determination of the optimal operating point, the VTHRCOMP value is calculated from the bottom of

the VTHRCOMP projection, where the efficiency raises above 50%, plus `deltavthrcmp`. If the parameter `deltavthrcmp` is negative, it is subtracted from the top of the distribution. In a second step, the `CALDEL` setting is determined from the minimum value, where the efficiency is above 50%, plus `fraccaldel` times the `CALDEL` plateau width (defined as the separation between the first and last `CALDEL` value with an efficiency above 50%, for the `VTHRCOMP` value determined in the first step).

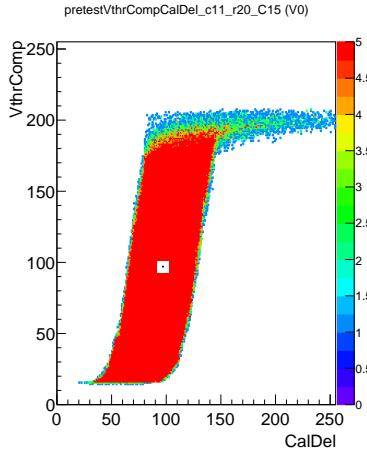


Figure 8: One example, for ROC 15, of the result of the `setvthrcmpcaldel` subtest. The black dot in the middle of the white square shows the operating point. It is located at `fracdel` ($0.5 = 50\%$) of the `CALDEL` width at the `CALDEL` value situated `deltavthrcmp` DAC units above the lower `VTHRCOMP` edge (cf. Fig. 5).

6.2 Alive

The Alive test, to be run after a successful Pretest has resulted in a proper configuration of the most basic DACs, determines the faulty pixels in the DUT by measuring the pixel hit efficiency, and checking that all pixels can be masked and that their addresses are properly encoded. These three defect types (inefficient, not maskable, address errors) are the most basic possibilities for a pixel to be faulty.

Note that the naming convention for this test is inconsistent¹ with the rest of `pXar`: In the GUI and `testParameters.dat` the test is called `PixelAlive`, while the C++ class is named `(PixTest)Alive`.

All subtests of the `Alive` test are implemented with calls to `PixTests::efficiencyMaps` (calling `pXarcore::getEfficiencyMap`), but differ in the pixel unmasking/masking and specific flags for the readout.

The `doTest` subtest runs the `alivetest`, `masktest`, and `addressdecodingtest` subtests. In contrast to the `Pretest:doTest`, there is no check for any problem and the complete sequence is performed without interruption.

¹Remember the motto of `pXar`: Life is too short for perfection.

6.2.1 Alive:alivetest

The `alivetest` sets VCAL to `vcal` (configurable, by default `VCAL= 200`) and `ntrig= 10` (configurable). Note that there is no specific setting of either the high-range or the low-range for the VCAL injection. This is a feature and the intended behavior since the user can specify the high-/low-range behavior with `CTRLREG` in the hardware tab. All pixels are tested and enabled/unmasked. The analysis of the resulting hit map (despite the name) is straightforward. Pixels with less than `ntrig` (1) hits are considered ‘problematic’ (‘dead’) pixels and counted separately per ROC. No check is made at this point against pixels with more than `ntrig` hits. The test parameter `maskdeadpixels` allows for masking ‘problematic’ pixels for subsequent tests. The test parameter `savemaskfile` allows to save this list to file. Fig. 10 (left) illustrates the pixel map for a single ROC (the color red indicates maximum hit counts and is a good sign), while Fig. 10 (middle) shows the summary plot for a module (where all 16 ROCs are concatenated in their physical order).

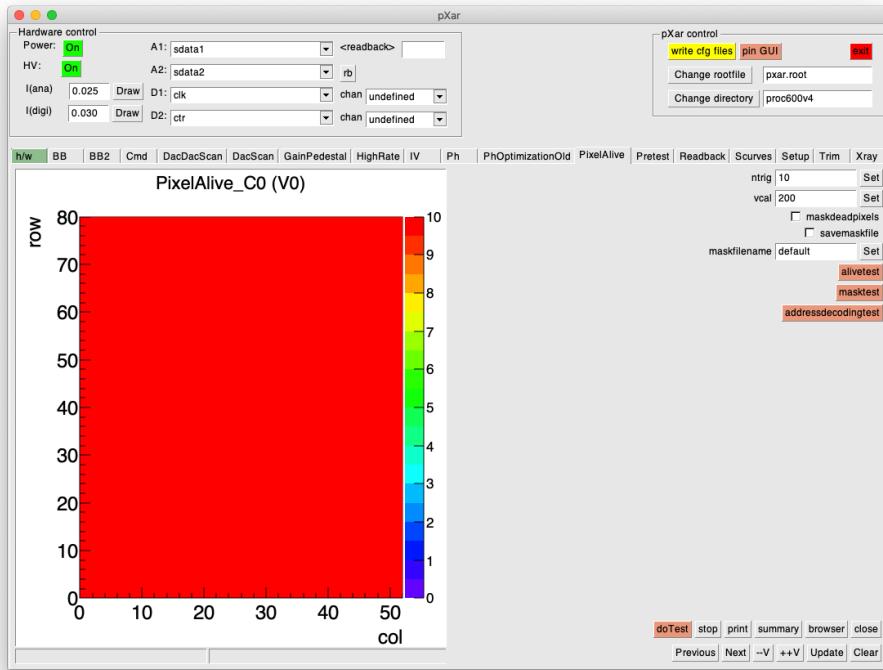


Figure 9: The Alive GUI tab. The `alivetest` subtest has been run.

6.2.2 Alive:masktest

The `masktest` is an important test because the masking of a noisy pixel is essential to avoid double columns being saturated with noise hits (resulting in a low hit efficiency). The implementation of this test is very similar to `Alive:alivetest` (`VCAL= 200` and `ntrig= 10`; cf. remarks above), except that all pixels are masked and therefore no hits are expected in the hitmap. For the visual representation of the test, a histogram is filled with +1 (-1) if

no (> 0) hits are detected for the pixel. To take into account dead pixels, the results of a previous `Alive:alivetest` is used to differentiate between masked and dead pixels. If no `Alive:alivetest` was run beforehand, it is run as part of this test. Figure 10 (right) illustrates the three-state results of this test (in the z -axis, the module had not mask defects).

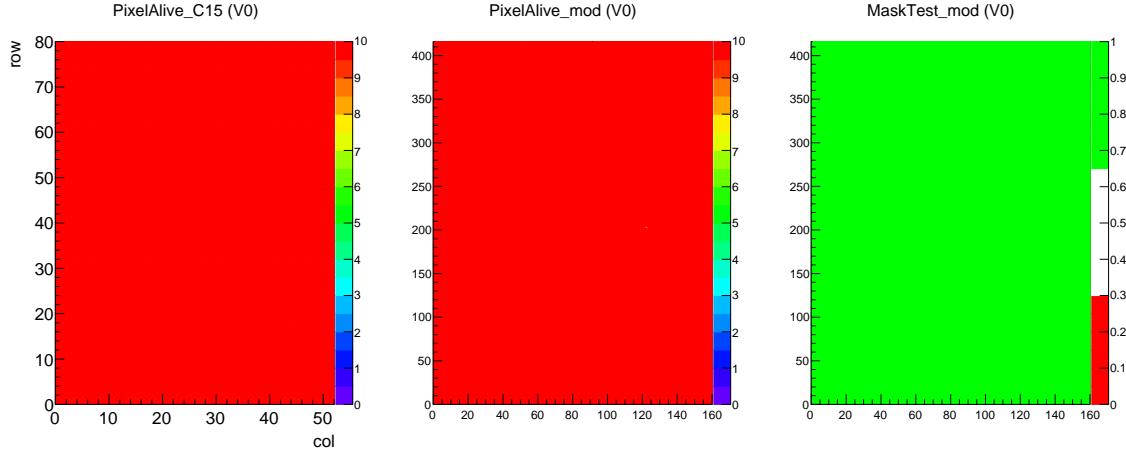


Figure 10: Illustration of the `alivetest` and `masktest` subtests. (left) Hitmap, with `ntrig` = 10, of ROC 15. All pixels are working with full efficiency. (middle) Summary plot of the entire module, with a few dead pixels visible. (right) Summary of the `masktest`, where green indicates success and red implies a failure of the mask bit in a pixel. White is used to indicate dead pixels. A similar color convention is used in the `addressdecodingtest` (not shown).

6.2.3 Alive:addressdecodingtest

For the `Alive:addressdecodingtest` all pixels are tested and enabled/unmasked. Again, the implementation of this test is very similar to `Alive:alivetest` (`VCAL`= 200 and `ntrig`= 10; cf. remarks above), except that the flag `FLAG_CHECK_ORDER` is active. Using the `pXarcore` convention that pixels appearing in the wrong readout position have a negative pulseheight/hit count allows to diagnose pixel with wrong pixel addresses. This is displayed with three-state results as in the `masktest` subtest.

6.3 Bump bonding tests

The bump bonding tests aim at testing the quality of the bump bonds connecting the pixelated sensor to the bump pads on the ROC (cf. the sketch in Fig. 3). Because of the technical details of the bump characteristics and the bump bonding process, in particular the size of the In bump bond, there is not a single solution to this task. In the past, several tests have been developed for various cases. In the following we describe the BB test, for the modules with PSI46digV2.1respin ROCs bump-bonded at DECTRIS, and the BB2 test, developed at DESY² and also used for the proc600v4.

²According to the source code, the authors are A. Vargas and D. Pitzl.

6.3.1 BB

The goal of the bump bonding tests is the characterization of the bump bond quality connecting the pixelated sensor to the bump pad in the PUC. Because different modules are bump-bonded with bumps of different sizes, there is no single bump bonding test that works for all cases reliably.

A bump bond test normally injects a test charge through a capacitive coupling into the sensor (**CALS** instead of **CAL**) and then determines the resulting VTHRCOMP threshold location or behavior). All flavors of bump bonding tests are susceptible to fluctuations and cannot provide reproducible results at the per mille level.

The BB test, illustrated in Fig. 11, is rather straightforward. It activates the high VCAL range, sets the flag **FLAG_CALS**, and invokes **PixTest::scurveMaps(VTHRCOMP, ...)**, with a user-defined VCAL value of **vcals** and a statistics of **ntrig**. The goal of the BB test is to attempt a VTHRCOMP threshold determination using the **CALS** signal path (the **CAL** signal path avoids direct charge injection into the electronics of the PUC and routes the charge capacitively through the sensor, cf. Fig. 3). If no such threshold is found, this is taken to be indicative of a faulty bump bond.

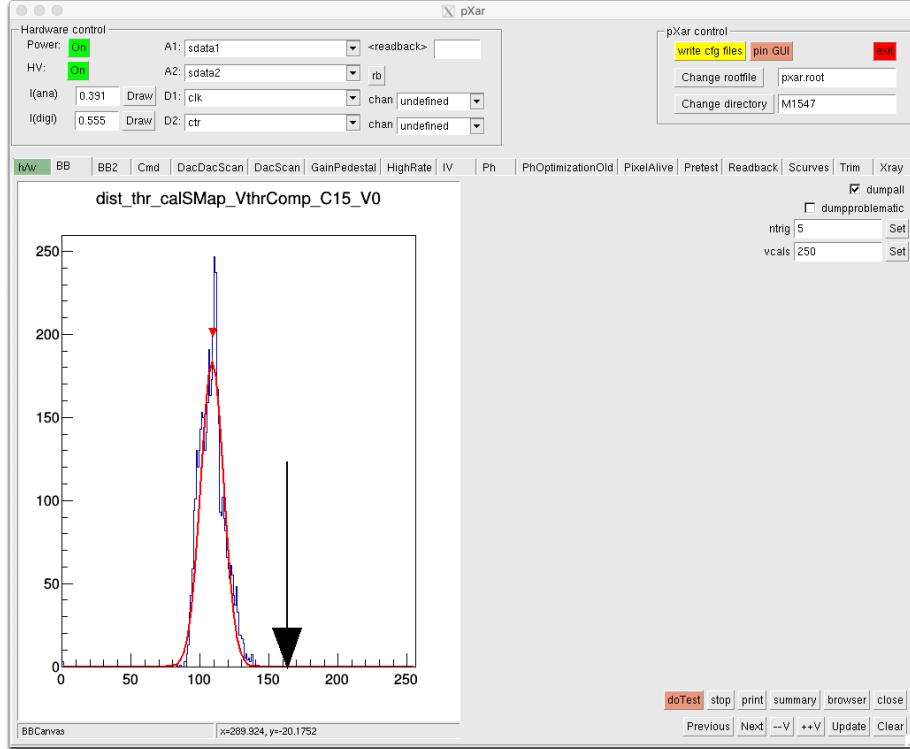


Figure 11: The BB GUI tab after running the **doTest** subtest.

For the analysis of the results, the 2D threshold maps are projected into distribution histogram which are analyzed with ROOT's **TSpectrum::Search(...)** using the options of a significance of 5σ and a threshold of more than 0.01. This will result in a number of peaks found in the threshold distribution. Faulty bump bonds will be characterized by low thresholds (large VTHRCOMP values) where noise can lead to a misidentified threshold

determination (before the double columns are saturated). In `PixTestBB::fitPeaks(...)` each of the identified peaks is fitted with a Gaussian function and the separation between successive peaks is determined. The primary purpose is to determine the `VTHRCOMP` value separating working bump bonds from faulty ones. Depending on how many peaks are found by `TSpectrum::Search()`, the separation is determined from the first and only peak or by interpolating between the first and second peaks. The code evolved over time to accommodate special cases while providing in general stable results.

The final results of the BB test is the number of faulty bump bonds, determined by integrating the `VTHRCOMP` distribution histogram from the separation to the end.

6.3.2 BB2

The BB2 test starts by re-setting VANA to achieve an analog current consumption specified by `targetia`, using code copied from `Pretest`. After that, it determines a specific `CALDEL` and `VTHRCOMP` setting, using the `CALS` signal path (instead of the `CAL` signal path as in the normal `Pretest::setvthrcompcaldel` subtest). The operating point is determined at the mean value of `CALDEL` and at 25% into the plateau region along `VTHRCOMP` (where the plateau is defined as the contiguous region with maximum readout counts). After this preparation, the high-VCAL range is activated, and the `CALS` signal path is used for `api::getEfficiencyVsDAC(VTHRCOMP, ...)`.

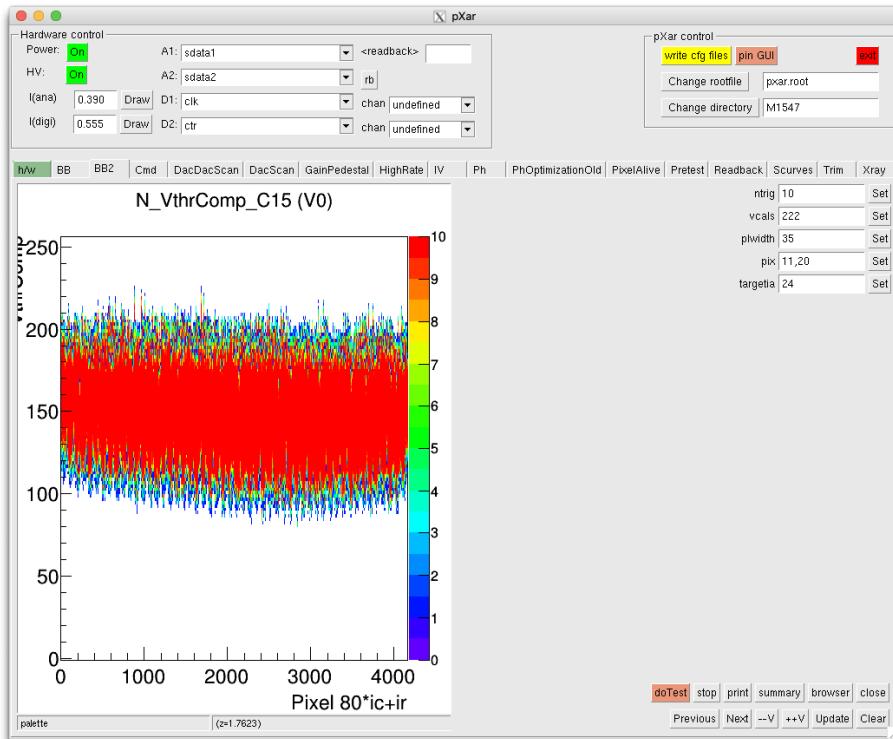


Figure 12: The BB2 GUI tab after running the `doTest` subtest.

The analysis determines a `VTHRCOMP` plateau width (cf. the plot in Fig. 12) and flags pixels with a width smaller than the user-defined parameter `plwidth` as having faulty bump bonds.

The results of the BB2 test are the separation value and the number of faulty bump bonds.

6.3.3 Discussion

The fact that there are multiple instances of the bump bonding test indicates that this is a nontrivial test. Neither the BB nor the BB2 test are perfect (not even for their target bump bond sizes/process), cf. Fig. 13. Both usually manage to determine extended regions with bump bonding issues, but the false negative rate (identifying pixels with faulty bump bonds that are in fact working) is non-negligible in both cases.

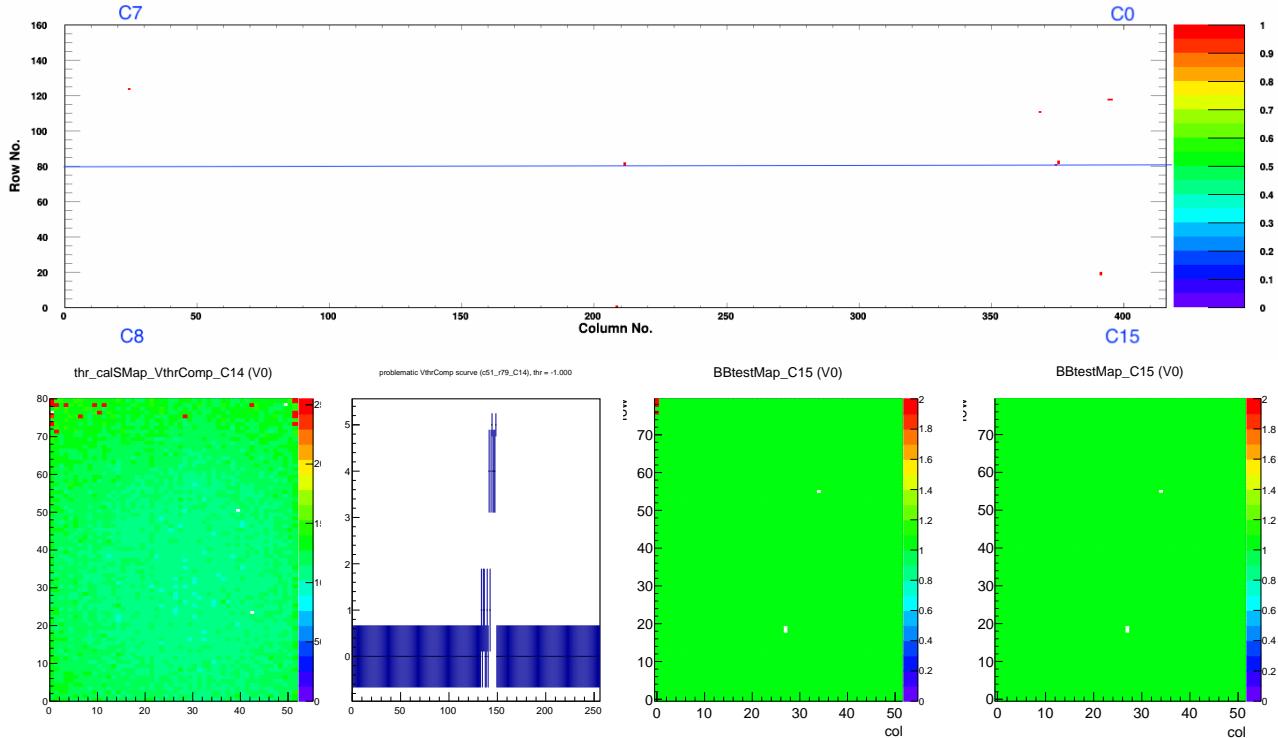


Figure 13: Comparison of the faulty bump bonds obtained with the high-rate x-ray test for module M1547 (top) and exemplary results for the same module from the BB (bottom left two plots) and BB2 tests (bottom right two plots). The large false negative rate in the bottom left plot is evident as there are no matching red pixels in the top plot. The bottom second left plot shows the reason why the BB test diagnosed the pixel in the top right corner as a faulty bump bond. The two bottom plots on the right* show that the diagnosis of bump bond failures is not completely reproducible with the BB2 test (cf. the difference in the faulty bump bonds in the first column), these results were obtained in two successive BB2 tests.

* Despite the name BBTestMap these plots are produced by the BB2 test.

6.4 Trim

The trimming test aims to configure the DUT in such a way that the operational thresholds of all pixels are as closely together as possible. (Operational thresholds denote the sensitivity of a

pixel, given the setting of VTHRCOMP, VTRIM, and its pixel-specific trimbits. Physical thresholds denote the pixel sensitivity for some specified value of VTHRCOMP and VTRIM = 0 and all trimbits turned off.) The operational threshold is specified by the user as a target VCAL DAC value. Test pulses with smaller charge should not result in a pixel signal, while test pulses with larger charge should result in a pixel signal. Ideally, VCAL s-curves have their 50% efficiency at this threshold value. Such a setup results in a uniform sensitivity and optimal position resolution within the CMS detector.

To achieve this goal, the Trim test configures the VTHRCOMP DAC in such a way, that the pixel with the lowest physical threshold passes the operational target VCAL threshold with no trimbits enabled. All other pixels with higher physical thresholds will have their operational thresholds lowered by a combination of $VTRIM > 0$ and pixel-specific trimbit settings. (To reduce the influence of outliers, ‘the’ pixel with the ‘lowest’ physical threshold is not really at the end of the distribution, but rather at the end of the bulk of the distribution.)

The GUI tab ‘Trim’, illustrated in Fig. 14, contains two subtests, `trim` and `trimbits`. The `trim` subtest is in fact a calibration procedure and consists in optimizing the VTHRCOMP VTRIM, and trimbit settings such that the VCAL threshold is as uniform as possible across a ROC. The `trimbits` test is a proper test that checks if the four trimbits can indeed lower the threshold given by VTHRCOMP.

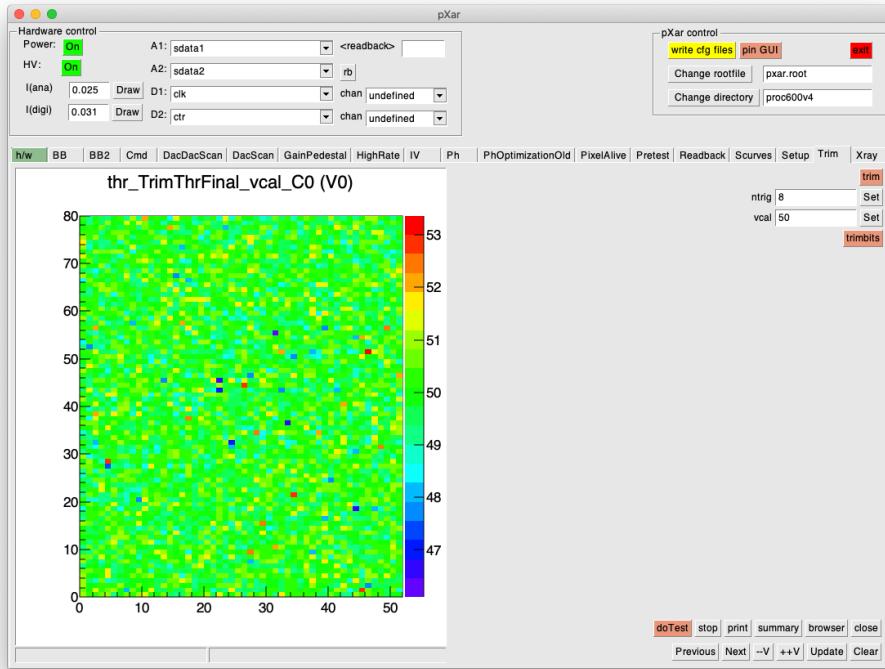


Figure 14: The Trim GUI tab after running the `trim` subtest.

The `doTest` subtest runs the `trim` calibration and the `trimbits` subtest, in that order. In principle, the order could be inverted (since one might argue that first one should check that the trimbits work, before attempting to trim the ROC). However, the `trim` calibration is

the primary concern here in the sense that it more probable that a ROC cannot be trimmed because of other issues than faulty trimbits and that faulty trimbits for a few pixels are of no operational concern.

6.4.1 Trim:trim

The basic idea of the `trim` calibration is to determine first the `VTHRCOMP` value corresponding to the required `VCAL` threshold, second determine a `VTRIM` value that allows modifying all individual pixel `VCAL` thresholds such they are consistent with the desired `VCAL` threshold, and finally to set (iteratively) the trimbits for all pixels. Because the trim bits only allow lowering of the `VTHRCOMP` threshold, the first step determines the maximum `VTHRCOMP` threshold, i.e. the minimum numerical value of `VTHRCOMP` (since `VTHRCOMP` is inverted, cf. section 3.3.1).

There are two input parameters to the `trim` calibration, (1) the `VCAL` threshold to which a ROC should be trimmed (conventional values range from `VCAL` = 35 to `VCAL` = 50, in the low range), and (2) the number of triggers for each measurement (by default, `ntrig`= 8). Trimming is a lengthy procedure (about half an hour for a module of 16 ROCs) and the choice of `ntrig` = 8 is a compromise between a narrow `VCAL` threshold distribution and a too lengthy procedure.

The `trim` calibration starts by setting `VTRIM` = 0, all trimbits to one (i.e. `TB` = 0xf = 15, since the trimbits obey an inverted logic, cf. section 3.3.2), and `VCAL` to the input value chosen by the user. The threshold map for the determination of the minimal `VTHRCOMP` is produced with `ntrig` as fixed by the user (if the choice was smaller than 5, `ntrig` is reset to 5). With this setup, s-curve maps are obtained with `scurveMaps(VTHRCOMP, ...)` scanning `VTHRCOMP`, cf. Fig. 15.

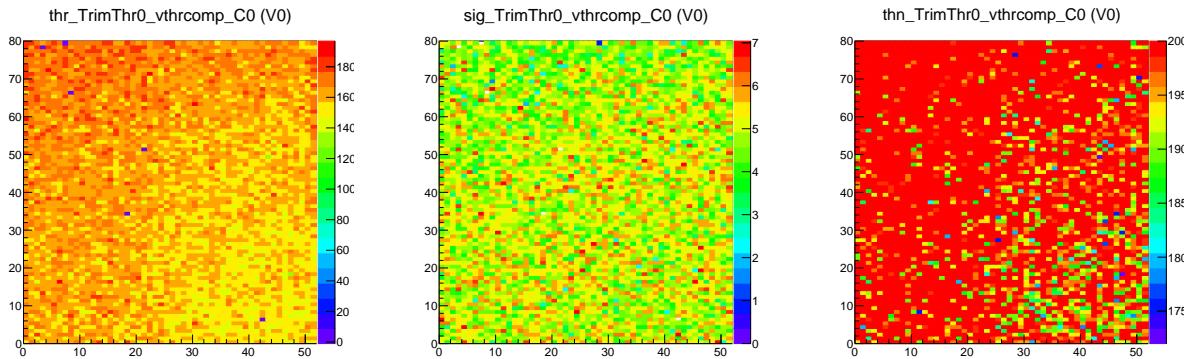


Figure 15: Illustration of the Trim:trim calibration. `VTHRCOMP` maps, obtained with `VTRIM` = 0, `TB` = 15, and `VCAL` = 50, showing (left) the threshold, (middle) the width of the threshold, and (right) the noise level.

The minimum numerical `VTHRCOMP` values are determined as follows:

- The mean and RMS of the `VTHRCOMP` distribution are determined. The minimum numerical `VTHRCOMP` value must be larger than the mean minus $2 \times \text{RMS}$ to remain in a ‘safe’ region and not be affected by outliers (pixels with abnormal `VTHRCOMP` thresholds, often located at the edge of the ROC).

- The minimum noise is determined in a similar way to the threshold, using the corresponding mean and RMS of the noise level distributions (Fig. 15 right). If the difference between the minimum noise value and the minimum VTHRCOMP threshold value is smaller than the `reserve` = 10, the minimum noise value minus 10 is returned instead of the minimum VTHRCOMP value. The origin of this protection mechanism is unknown. Likely there was an issue at some point that was cured with this hack.
- If the desired VCAL threshold is very low, $VCAL \leq 30$, the minimum VTHRCOMP thresholds are decreased by 5 DAC units to compensate for drifting thresholds.

Once the minimum VTHRCOMP value has been determined, all ROCs have their VTHRCOMP DACs updated and the VCAL threshold map is produced with `scurveMaps(VCAL, ...)` to determine the pixels with the largest VCAL thresholds. These initial VCAL threshold maps are stored as TrimThr1 histograms, cf. Fig. 16 (left). It is evident from the figure that the VCAL thresholds do not yet correspond to the desired value of $VCAL = 50$ ('evident' because many pixels have VCAL thresholds shown in yellow and orange, well above the green of $VCAL = 50$). The pixel with the maximum VCAL threshold is determined in the 'safe' region $\text{mean} \pm 2\text{RMS}$ (as above), in this example located at column 21 and row 51 (at the ROC edge there are pixels with higher VCAL threshold values, but they are not contained in the 'safe' region).

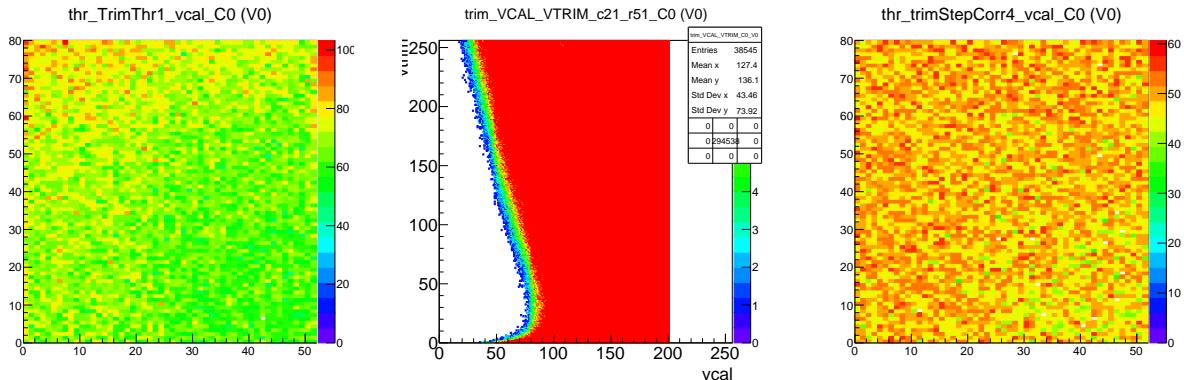


Figure 16: Illustrations of the `Trim:trim` calibration. (left) VCAL threshold map after setting the VTHRCOMP DAC to the minimum value compatible in the safe region. (middle) Scan of VTRIM-VCAL for the pixel with the highest VCAL threshold to obtain the corresponding VTRIM setting. (right) VCAL threshold map after the first trim step with a correction of $TB = 4$. Note that the VCAL threshold variation is already much reduced to the plot on the left.

This pixel is then used to fix the value of VTRIM: The trimbits for this pixels are set to zero to obtain the maximum threshold variation and a VCAL-VTRIM DACDAC-scan is performed, cf. Fig. 16 (middle). At the (arbitrary, but safe) value of $VCAL = 150$, the VTRIM distribution is projected and the last VTRIM bin with more than 50% hit efficiency provides the starting value for the determination of VTRIM. Starting from that point minus 20 (normally this corresponds to $VTRIM = 235$), the VTRIM-VCAL histogram is projected binwise in VTRIM onto the VCAL axis and the corresponding VCAL threshold is determined. As long as the VCAL threshold is lower than the target threshold, VTRIM is lowered. Once the threshold is higher (i.e. the correction was too large), the loop is interrupted and the VTRIM value is stored. At this point it is checked

whether this VCAL threshold is closer to the target or whether that is true for the previous one, and the one closer is stored.

From here on, the trim algorithm is trivial and amounts to properly setting all trimbits in the shortest time possible (this is achieved with a binary search algorithm, starting with $TB = 7$ and halving the TB change for each step; at each step it is checked whether the attempted correction decreases the distance from the desired VCAL threshold or not and no change is implemented if the distance increased). The progress is reflected in a set of threshold maps named 'TrimThr4', 'TrimThr2', 'TrimThr1a', 'TrimThr1b', 'TrimThrFinal', where the numbers correspond to the correction values applied.

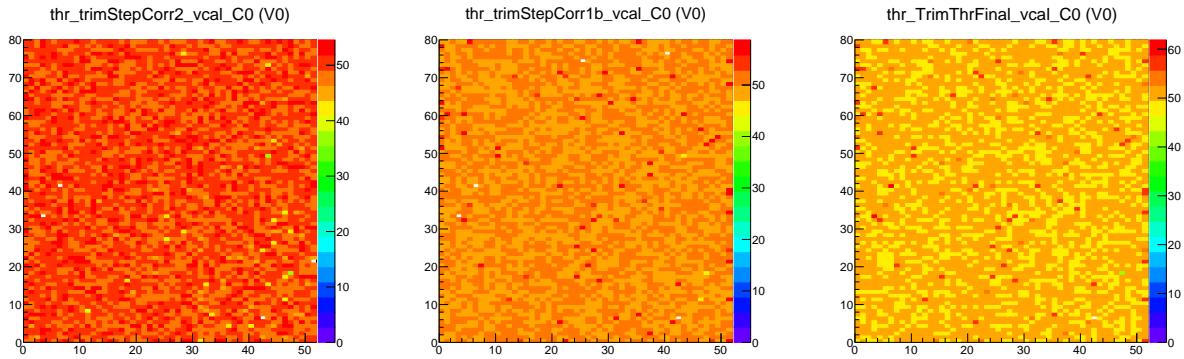


Figure 17: Illustrations of the `Trim:trim` calibration. (left) VCAL threshold map after setting the VTHRCOMP DAC to the minimum value compatible in the safe region. (middle) Scan of VTRIM-VCAL for the pixel with the highest VCAL threshold to obtain the corresponding VTRIM setting. (right) VCAL threshold map after the first trim step with a correction of $TB = 4$. Note that the VCAL threshold variation is already much reduced to the plot on the left.

The final characterization of the `Trim:trim` calibration is done with the histograms shown in Fig. 18. The distribution of the trimbit values TB , shown in Fig. 18 (left), should be a uni-modal distribution, ideally with a peak around the center. This is seldom the case. The distribution of the VCAL thresholds should be a narrow peak centered at the desired input value. In Fig. 18 (middle) a good example is shown where the RMS of the distribution is about 1.2 VCAL DAC units. An example where the VCAL calibration did not achieve the same quality is shown in Fig. 18 (right). The (technical) for this bad trimming is due to missing threshold determinations in an intermediate trimming step ('TrimThr1a' in this example), which cannot be corrected anymore. A possible cure would be to add another iteration, prolonging the calibration procedure.

6.4.2 Trim:trimbits

The `trimbits` subtest determines the four VCAL threshold difference maps between a configuration where all trimbits are off, *i.e.* $TB = 0xf = 15$, and where each trimbit is enabled, *i.e.* $TB \in 14, 13, 11, 7$ (since the trimbits follow inverted logic, cf. section 3.3.2). To achieve threshold differences that are reasonably similar, VTRIM is modified for each step (VTRIM is set to 255, 254, 130, 60 for testing the least-significant trimbit to the most-significant trimbit).

The `trimbits` test starts by disabling all trimbits ($TB = 15$), setting $VTRIM = 0$, and

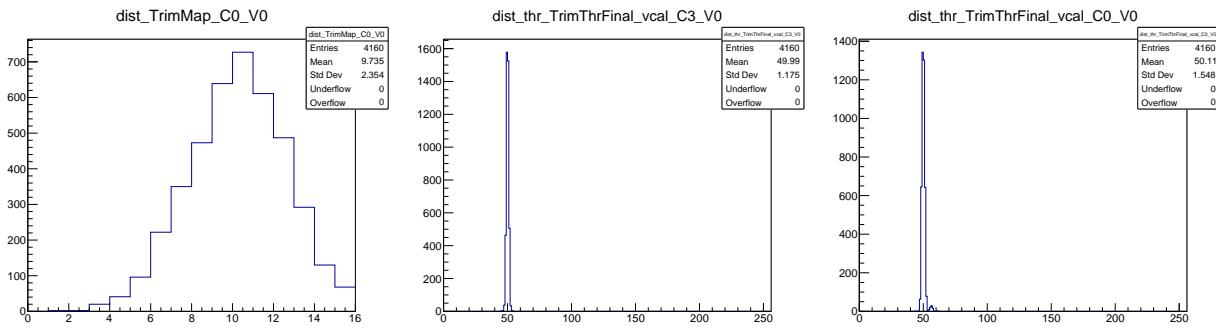


Figure 18: Illustrations of the convergence and quality of the `Trim:trim` calibration. (left) Distribution of the trimbits after the final trimming iteration. (middle) Distribution of the VCAL thresholds for a well-trimmed ROC. (right) The same distribution for a not so well trimmed ROC. This plot is the z -axis projection of Fig. 17 (right). The difference to the middle is evident from the different RMS. The small satellite peak with larger VCAL thresholds is composed of the red pixels visible in Fig. 17 (right).

switching into the low-VCAL range. To speed up the test, it divides `ntrig` (the parameter common with `Trim:trim`) by a factor 2, but making sure that it is at least `ntrig` = 5. In this configuration, a VCAL threshold map is obtained that serves as the basis for comparison. In a second step, all trimbits are individually and successively enabled, `VTRIM` is set to the corresponding value provided above, and for each setting a VCAL threshold map is measured.

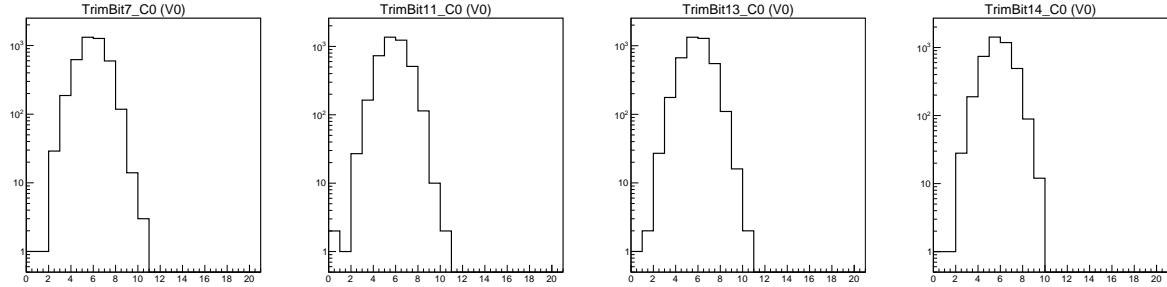


Figure 19: Distributions of the VCAL threshold difference maps between the reference map (obtained with `VTRIM` = 0 and all trimbits disabled) and the maps with the settings `TB` = 7 (most significant trimbit), 11, 13, and 14 (least significant trimbit), from left to right.

The final analysis of the `trimbits` test compares, for each enabled trimbit, the initial and the corresponding VCAL threshold map not including dead pixels (detected by the absence of a threshold in the initial threshold scan). It should be noted that for the two most significant trimbits the threshold difference be been shifted to substantially higher values. It is a choice (and a rather arbitrary one) that this is not done.

6.5 Scurves

The primary purpose of the **Scurves** test is to determine for each pixel a threshold measurement, using `scurveMaps(...)` (cf. section 4.2.4 and Fig. 4), with all associated characteristics (position and width, potentially with the noise level for VTHRCOMP thresholds) and to possibly write the raw data to an ASCII file for independent analysis. In the **FullTest**, the **Scurves** test provides an independent and precise (*i.e.*, the thresholds are measured with large `ntrig` = 50) validation of the trimming algorithm in terms of the **VCAL** threshold. It does not provide a validation of the **Ph** (arguably it should be run after that test and not before it). The **Scurves** test does not switch the **VCAL** range. In the fulltest, it is run after the **Trim** test which operates in the low **VCAL** range.

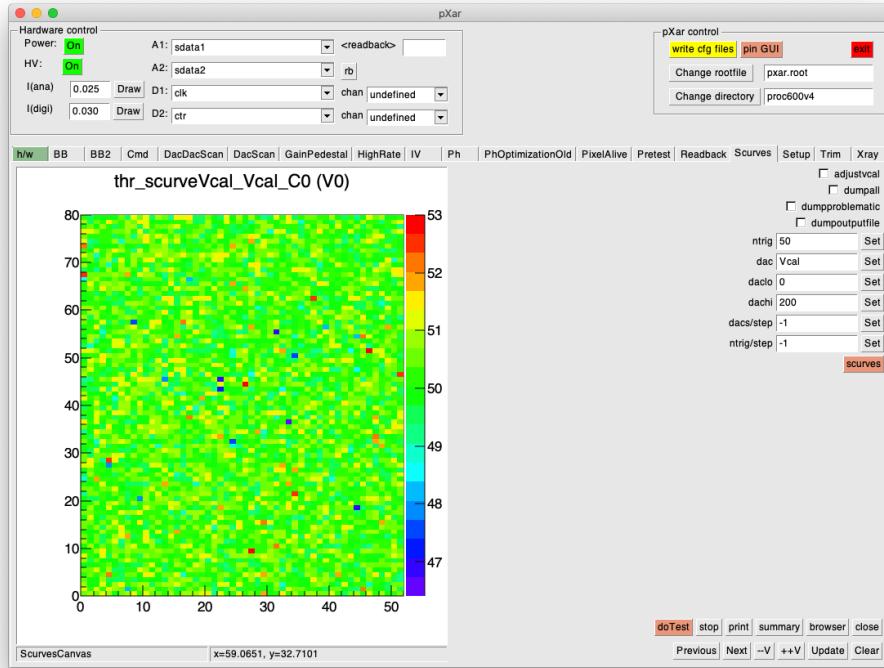


Figure 20: The **Scurves** GUI tab after running the **Scurves** subtest.

Since the **Scurves** test relies entirely on `scurveMaps(...)` for its entire functionality, its source code is very compact and consists basically only in the configuration of the function call to `scurveMaps(...)`.

The possible options for the **Scurves** test are explained below.

- **adjustvcal** By default, the **Scurves** subtest does not modify any DAC parameters. If a VTHRCOMP scan is performed, it may be advantageous to set the **VCAL** DACin such a way that the average VTHRCOMP threshold is at the default VTHRCOMP threshold (as given in the hardware tab or the `dacParameter` file). Checking this button allows to achieve this. Procedurally, the approach is very simple and not fool-proof: The pixel at position (11,20) is used to do a VTHRCOMP-VCAL scan. The resulting histogram is used

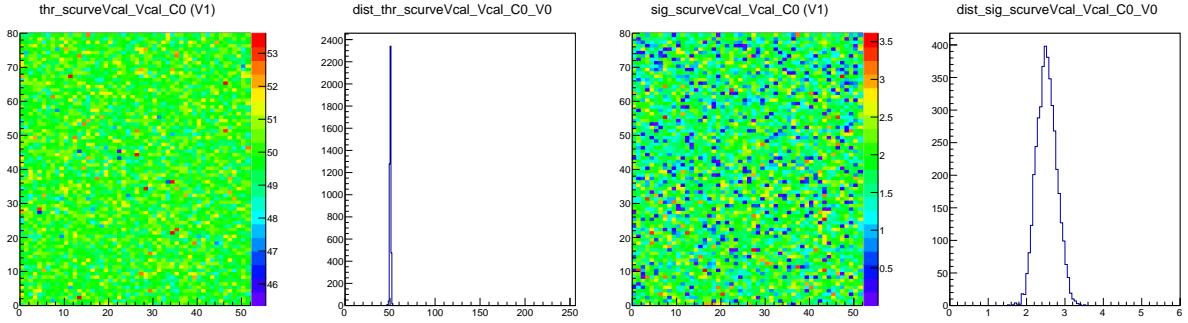


Figure 21: Illustrations of the **Scurves** test. (left) VCAL threshold map for a ROC trimmed to $\text{VCAL} = 50$. (second from left) Projection onto z -axis of the VCAL threshold, *i.e.*, the VCAL threshold distribution. (second from right) VCAL threshold width map. (right) The corresponding distribution.

to determine the VCAL threshold at the predefined VTHRCOMP threshold. Fig. 22 (left) illustrates the concept.

- **dumpall** Adds all s -curves and their fits to the embedded canvas. This corresponds to setting `result = 0x20` in `scurveMaps(...)`, cf. 4.2.4.
- **dumpproblematic** Adds only the problematic s -curves and their fits to the embedded canvas. This corresponds to setting `result = 0x10` in `scurveMaps(...)`, cf. 4.2.4. In Fig. 22 (right) an example of a ‘problematic’ fit is shown (this is not a real problem as the threshold is correctly determined).
- **dumpoutputfile** Writes the s -curves to an ASCII file.

6.6 Ph (Pulse height optimization)

The analog pixel signal is digitized in the ROC periphery. The amplitude of the resulting digital ADC signal is referred to as pulse height and abbreviated as PH.

The purpose of the **Ph:optimize** test is to ensure that small signals are output with small ADC values and that large signals are output with large ADC values in such a way that the largest possible ADC range is used. This is achieved by tuning the DAC parameters PHOFFSET and PHSCALE. The basic idea behind the algorithm is to maximize the range (difference between the pulse height of large signals and small signals).

The test proceeds by determining, per ROC, the pixels with the lowest and highest pulse height for a fixed charge injection corresponding to $\text{VCAL} = 255$ in the low range. The pulse height map shows a significant pixel-to-pixel variation, as shown in Fig. 23. If a random pixel were chosen for the optimization, saturation (at the low and high end) would affect the pulse height distribution. The choice of the pixel for the high-PH optimization is not critical. The choice of the pixel for the low-PH optimization is more delicate: The pixel with the smallest PH (for a given charge injection) may result in a set of PHOFFSET and PHSCALE parameters that leave entire double columns with unobservable signals at the low end. To avoid this, the

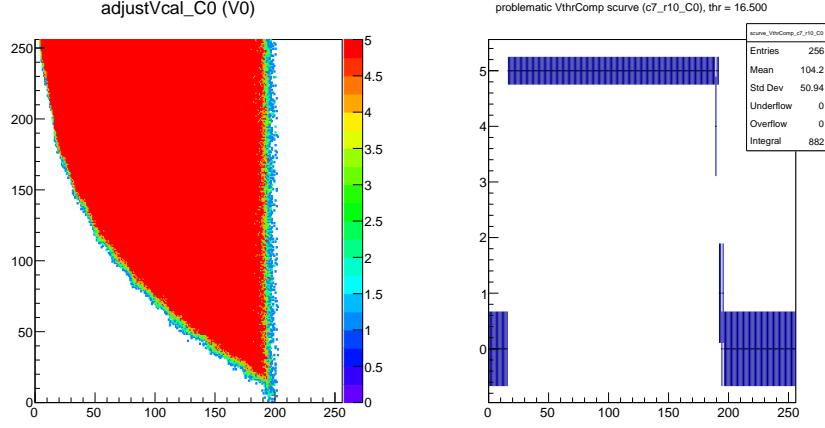


Figure 22: (left) 2D VCAL vs. VTHRCOMP scan used for the VCAL adjustment when running an **Scurves** subtest for VTHRCOMP. In case it is not clear which axis is showing which DAC: the loss of efficiency along the abscissa indicates that the abscissa is VTHRCOMP, and by consequence the ordinate is VCAL. (right) Illustration of an *s*-scurve where no fit with an error function is performed because the data correspond to a sharp step function. These cases are retained when the `dumpproblematic` checkbox in the **Scurves** tab is checked.

pixel with the minimum PH is chosen from the double column with the smallest average PH (not including dead pixels).

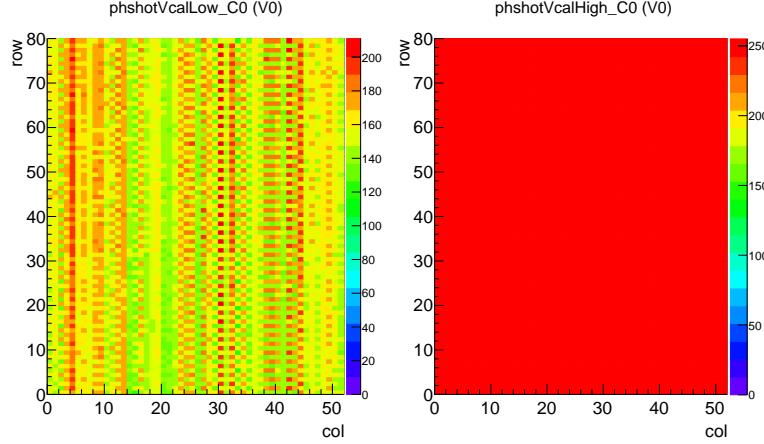


Figure 23: Pulse height map, before PHSCALE and PHOFFSET optimization, for $\text{VCAL}=10$ (left) and $\text{VCAL}=255$ (right), in the high range. The significant pulse height variation for the low-VCAL injection is visible in the left plot, and the saturation for the large-VCAL injection is evident in the right plot.

The two DAC parameters PHOFFSET and PHSCALE are scanned for a low and high VCAL value (in the high range) for the two pixels (the pixel with the lowest PH for the low-VCAL injection and the pixel with the highest PH for the high-VCAL injection). The two 2D distributions of the high-PH and the low-PH values are subtracted and the optimal setup is chosen as that

(PHOFFSET, PHSCALE) point which maximizes the range. These three steps are illustrated in Fig. 24.

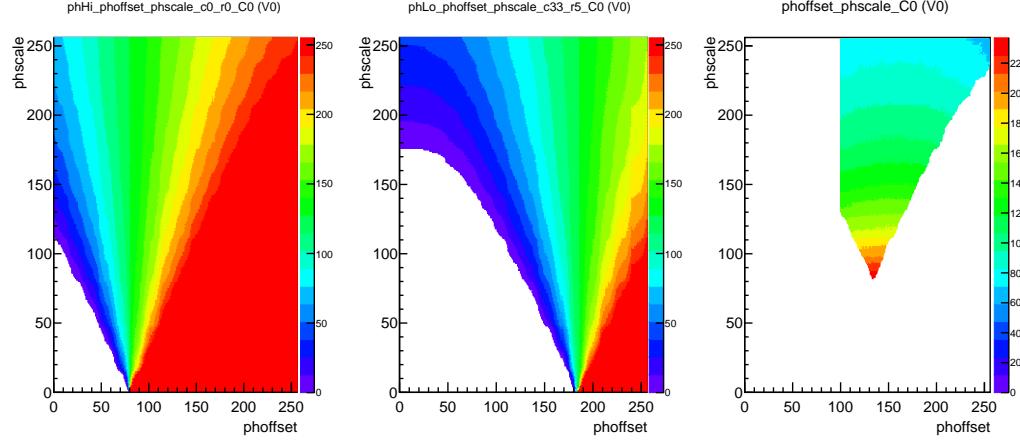


Figure 24: Illustration of the pulse height optimization algorithm. (left) Scan for $\text{VCAL}=255$ of PHSCALE vs. PHOFFSET for pixel $(0,0)$ which is chosen because the entire chip is in saturation (cf. Fig. 23 right). (middle) The corresponding scan for pixel $(33,5)$ for $\text{VCAL}=10$. (right) The difference between the left and middle plot. The shape of the contour is due to requirements that the maximum pulse height be less than 255 ($= \text{phmax}$), that the minimum pulse height be larger than 10 ($= \text{phmin}$), and that phoffsetmin be larger than 100.

The test parameters controlling this algorithm are summarized (in parenthesis the default values are provided) in the following.

- **vcallow** (= 10, high range) set the charge injection when determining the pulse height in the small signal range. It is used to determine to the pixel (per ROC) with the smallest pulse height value. It should be noted that **vcallow** should be above the minimum threshold to which the ROC has been trimmed (beforehand!).
- **vcalhigh** (= 255, high range) set the charge injection when determining the pulse height in the large signal range. It is used to determine to the pixel (per ROC) with the largest pulse height value.
- **phscalemin** (= 50) minimum setting for PHSCALE. Parameters below this value are not considered as valid points.
- **phoffsetmin** (= 100) minimum setting for PHOFFSET. Parameters below this value are not considered as valid points.
- **phmin** (= 2) minimum pulse height acceptable for the pixel with the smallest pulse height response with a calibration pulse of **vcallow**.
- **phmax** (= 250) maximum pulse height acceptable for the pixel with the largest pulse height response with a calibration pulse of **vcalhigh**.

The dependence of the optimization procedure on the exact values of **phmin** and **phmax** is not large, as illustrated in Fig. 25.

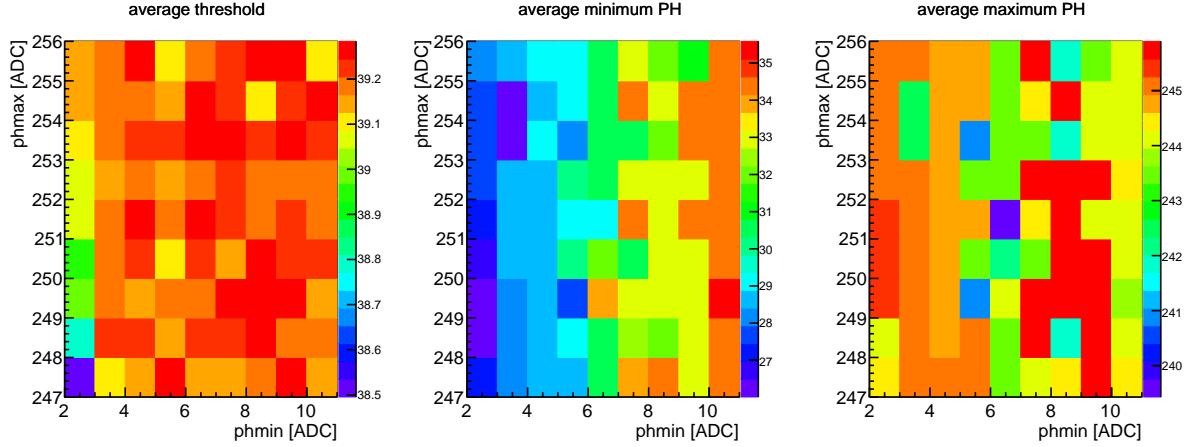


Figure 25: Dependence of the pulse height optimization on phmin and phmax , determined with the `GainPedestal` test. The top plot shows the average threshold (for a ROC trimmed to $\text{VCAL}=35$, low range). The bottom left plots shows the average of the minimum pulse height distribution, the right plots show the average of the maximum pulse height distribution. Note the compressed z -axis scale for the lower two plots.

As validation of the optimization procedure, the pulse height distribution for the small and large VCAL charge injection are provided in Fig. 26.

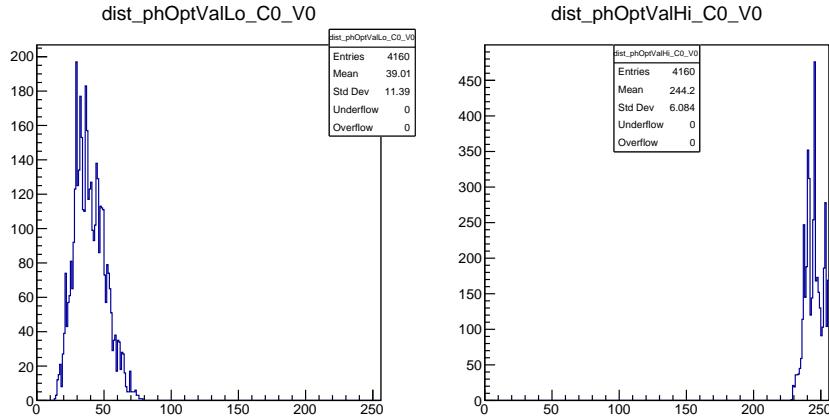


Figure 26: Validation of the `Ph:optimize` test: distribution of the low- VCAL charge injection (left) and the large- VCAL charge injection test.

It should be noted that the optimization results are not well defined if the test is run with sub-optimal parameters on untrimmed ROCs (especially if the parameter `vcallow` is below the VCAL threshold for all pixels).

6.7 GainPedestal calibration

The purpose of the `GainPedestal` calibration is the determination of the relationship between `VCAL` and the readout pulse height. It results in a parametrization of this relationship which can be used to determine the deposited charge given a measured pulse height. Two functional forms for fitting the `VCAL-PH` points are provided:

- The default choice for the `proc600` is based on the error function

$$f = p_3 \times \left(\operatorname{erf}\left(\frac{x - p_0}{p_1}\right) + p_2 \right), \quad (3)$$

where $\operatorname{erf}(x) = (2/\sqrt{\pi}) \int_0^x \exp^{-t^2} dt$. This is the identical function used in fitting *s*-curves (cf. section 4.2.4), though with a different parameter initialization.

- A secondary possibility is based on the hyperbolic tangent function

$$f = p_3 + p_2 \times \tanh(p_0 \times x - p_1) \quad (4)$$

which was found to provide a better description of the `psi46digV2.1` response.

The `doTest` subtest runs the full `GainPedestal` calibration, consisting of the `VCAL`-pulse height measurements (`measure`), the fitting of the data (`fit`), and the output of the resulting calibration files to disk (`save`). In Fig. 27 the `GainPedestal` tab is shown.

The `VCAL` points are hard-coded in `PixTestGainPedestal::measure` and consist of two groups, in the low-`VCAL` range and the high-`VCAL` range. In the low-`VCAL` range, n points are defined in the interval $10 < \text{VCAL} < 255$, separated by the input parameter `vcalstep` (by default set to 10, *i.e.*, by default $n = 25$). In the high-`VCAL` range (the scale factor to the low-`VCAL` range is 7), points are created at $\text{VCAL} \in \{30, 50, 70, 90, 200\}$ by default. If the checkbox `extended` is selected, additional `VCAL` points are created in the high-`VCAL` range: at $\text{VCAL} = 10, 17, 24$, and 120. These additional points help in a better determination of the curvature towards the plateau region and provide additional constraints where determining the scale factor (nominally 7) between the low- and high-`VCAL` ranges. The default `VCAL` points are illustrated in Fig. 28 (left), where all `VCAL` points are shown in the low-`VCAL` range.

To accommodate the large data volume, an array of `shist256` histograms is allocated with the placement-new operator of C++ and initialized. For all defined `VCAL` points in both low- and high-`VCAL` range, `api::getPulseheightVsDAC(...)` are called with the chosen `ntrig` setting (10 by default) and subsequently filled into the corresponding `shist256` histograms. The contents of all histograms is then saved into ASCII files with the name stub `phCalibration_Cxx.dat`. The contents of these files is described in the files (header and last three columns for each row).

The `GainPedestal` fits are performed in standard ROOT histograms, filled from the corresponding `shist256` histograms. By default, the fits are not shown (there would be too many histograms in the embedded canvas); the checkbox `showfits` allows to show the fits nevertheless³. To determine the non-linearity of the pulse height vs. `VCAL` relation at the low `VCAL`-range, the fitted function f (error function or the hyperbolic tangent) is integrated in

³It should be noted that the termination of `pXar` with such a large amount of memory allocated on the heap will take a bit of time.

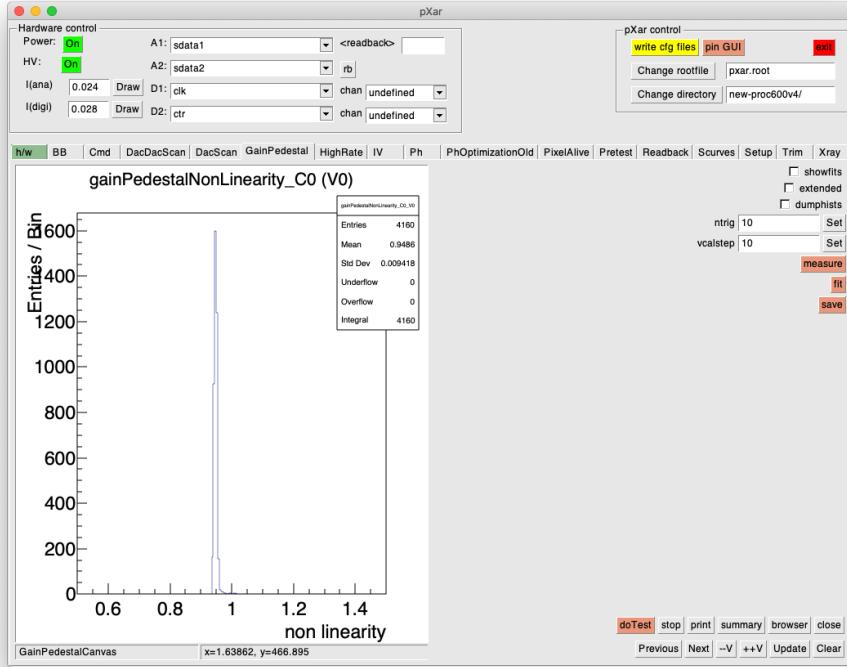


Figure 27: The `GainPedestal` GUI tab after running `doTest`.

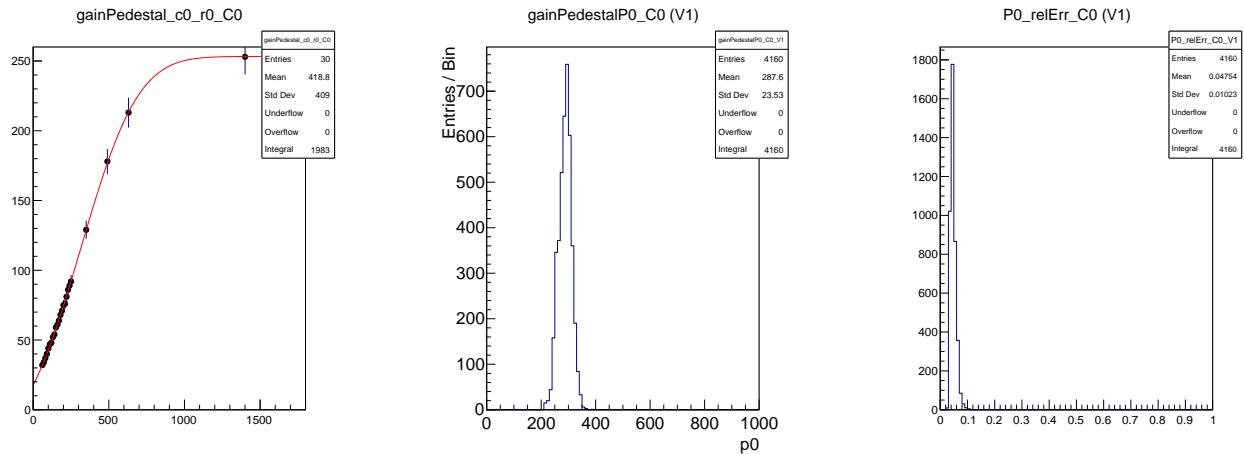


Figure 28: Plots illustrating the `GainPedestal` calibration. (left) Relationship between (low-range) VCAL on the abscissa and PH on the ordinate, with a hyperbolic tangent fitted to the data points. Histograms of (middle) p_0 and (right) its error, cf. Eq. 3.

the low VCAL-range $0 < \text{VCAL} < 200$ and its integral is divided by the integral of a polynomial of first degree going through the points $(0, f(0))$ and $(200, f(200))$. Ideally this ratio would be around 1, in reality it is at ≈ 0.9 as illustrated in the final plot of the `GainPedestal` calibration after running `doTest` (cf. Fig. 27). If the checkbox `dumphists` is selected, the

histograms together with the fitted functions will be written into the `pXar` rootfile.

As the final step of `doTest`, the fit parameters are written into ASCII files with the name stub `phCalibrationFitErrYY_CXX.dat`, where YY indicates the VCAL trim value (if set) and XX indicates the ROC number⁴. The name stub can be changed by defining in `configParameters.dat` the name `gainPedestalParameters` (by default, `mkConfig` does not include this name, but it can be added manually).

6.8 Threshold/efficiency scans and maps (DacScan and DacDacScan)

Two tabs allow for running a 1D DAC scan or a 2D DACDAC scan. In both cases, a specific pixel address can be specified with the box `pix`, the DAC(s) to be scanned can be specified using the box `DAC` (or `dac1` and `dac2`) and the ranges of the scans can be defined. There is no check applied that the DAC name is correctly spelled; specifying a non-existing DAC name simply results in an error message and an empty histogram. (The correct spelling of the DAC names is provided in the h/w tab.)

The differences to the `Scurves` test are as follows

- `Scurves` provide an efficiency scan and do not allow for a pulse height scan
- `Scurves` always run the DAC scan over all pixels of a ROC
- `Scurves` always fit a threshold function to the resulting efficiency scan

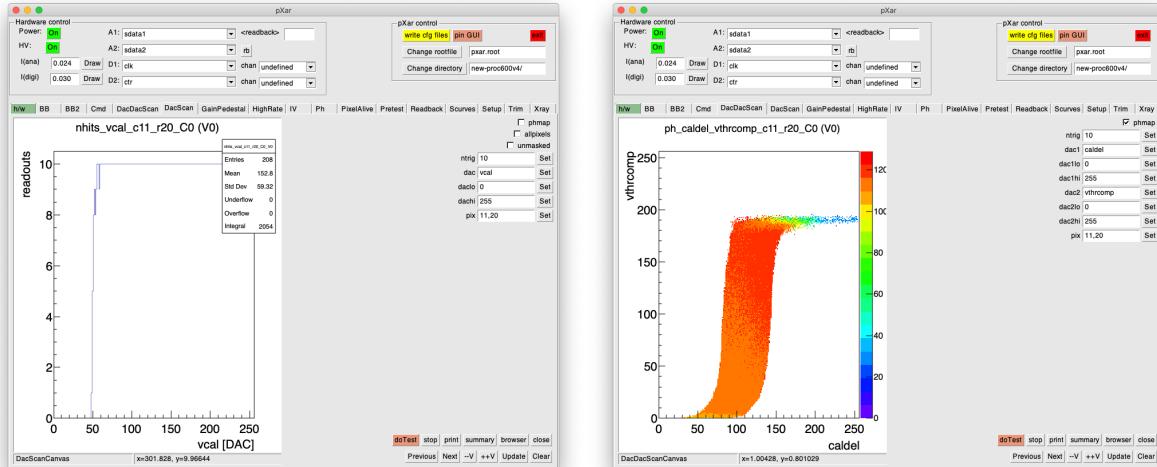


Figure 29: The GUI tabs for (left) `DacScan`, showing a scan of the efficiency vs. VCAL, and (right) `DacDacScan`, showing a pulse height map of VTHRCOMP vs. CALDEL, after running `doTest`.

The `DacScan` test allows a configuration for specific special cases. Checking

- `phmap` will provide a pulse height scan instead of a hit scan (basically an efficiency scan).

⁴There is, of course, a certain inconsistency of having the VCAL trim value as part of the ASCII file names for `GainPedestal` fitted parameters but not the corresponding data files. There are no plans to change this.

- `allpixels` will run the `DacScan` over all pixel of the DUT.
- `unmasked` will unmask the entire DUT using the flag `FLAG_FORCE_UNMASKED`. To measure the background hits in the enabled pixels, the test will book, in addition to the histograms required for the `DacScan`, another set of histograms (one per ROC) to contain the bitmap of the ROC with all hits registered that are not related to the `DacScan` test. These background hits can originate from cross-talk or irradiation (source or beam). To distinguish between the test hits and the background hits, the flag `FLAG_CHECK_ORDER` is enabled.

As usual, `ntrig` allows to adjust the number of triggers to find the optimum balance between speed and accuracy.

`DacScan` calls `api::getPulseheightVsDAC(...)` or `api::getEfficiencyVsDAC(...)`, depending on whether `phmap` is checked. For `DacDacScan` the corresponding 2D API functions are called. Because the initial DAC value(s) is (are) cached in the API calls, the test code does not cache the DACs locally. At the end of a `DacScan` or a `DacDacScan`, the DAC(s) are reset to their initial values before the scan(s).

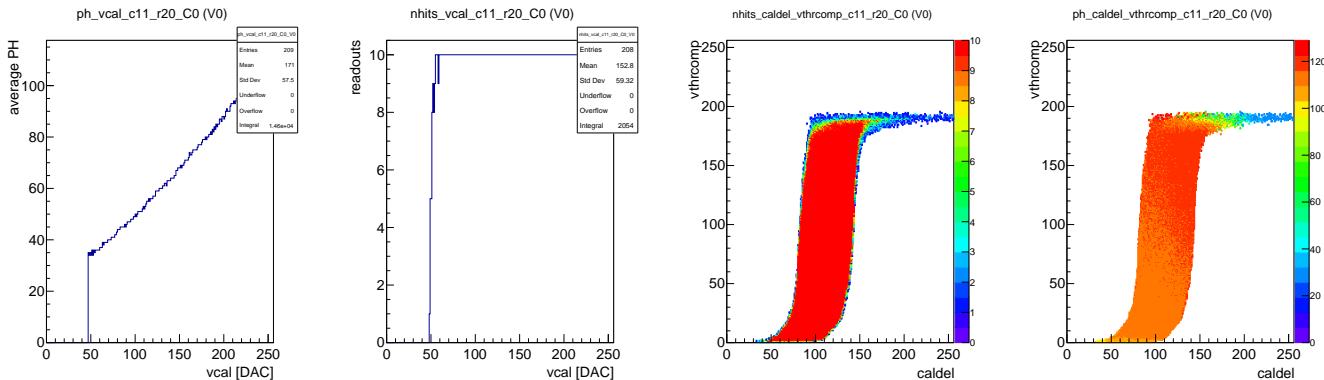


Figure 30: Plots illustrating the `DacScan` and `DacDacScan` tests. (left two plots) pulse height scan and efficiency scan vs VCAL. (right two plots) VTHRCOMP vs. CALDEL scan as a bitmap and a pulse height map.

There is a difference between `DacScan` and `DacDacScan`: The `moduleMap` function, called by hitting the `summary` button, fills all ROC values into one 1D histogram for `DacScan`, while for `DacDacScan` that function should not be called. (There is no possibility to call, or implement in a practical way, a `DacDacScan` for all pixels of a DUT.)

6.9 Readback test and calibration

The `Readback` test, developed by V. Tavolaro [7], provides a means to calibrate the ‘readback’ mechanism of the ROC, where on-chip measurement information is read out bit-wise, appended to the ROC header. The value written into the `readback` register controls the type of information encoded into the `Readback` bit of the readout; it is summarized in Table 1.

The analog `readback` measurements, *i.e.*, values 8–12 in the `Readback` register, need to be calibrated for a proper conversion of ADC units to voltage and current units, respectively. The

Table 1: **Readback** register controlling the information encoded in the **readback** bit of the ROC header. The analog **readback** values are only updated after writing to this register.

Value	Name	Type	Description
0	last data	digital	<i>e.g.</i> last DAC value
1	last address	digital	
2	last pixel column	digital	
3	last pixel row	digital	
8	vd	analog	unregulated digital voltage, $\approx 16 \text{ mV}/\text{ADC}$
9	va	analog	unregulated analog voltage, $\approx 16 \text{ mV}/\text{ADC}$
10	vana	analog	regulated analog voltage, $\approx 8 \text{ mV}/\text{ADC}$
11	vbg	analog	bandgap reference voltage, $\approx 8 \text{ mV}/\text{ADC}$ (should be 1.2 V)
12	iana	analog	analog current, $\approx 0.25 \text{ mA}/\text{ADC}$ (should be 1.2 V)

doTest subtest of the **Readback** test invokes the **calibratevd**, **calibrateva**, **readbackvbg**, **getcalibratedvbg**, and **calibrateia** subtests.

The subtest **calibratevd** relates the on-chip ADC value for vd, corrected for voltage drops due to the module cable resistance $R_{\text{cable}} = 0.338 \Omega$ and a ground level difference of $\Delta_{\text{GND}} = 0.1815 \text{ V}$ between the module adapter card and the HDI (for modules, but not for single-chip setups plugged directly into the adapter card), with the digital current I_d measurements obtained with the DTB as follows:

$$\text{RB}(\text{vd}) = p_0 + p_1 \times (V - R_{\text{cable}} \times I - \Delta_{\text{GND}}), \quad (5)$$

where RB is the **readback** ADC response for the voltage V measured with the DTB. The corrections are determined for average configurations (*e.g.* module cable length of 15 cm). The parameters p_0 and p_1 are the calibration parameters determined in this calibration subtest.

The subtest **calibrateva** relates the on-chip ADC values for va with the measured quantities on the DTB with an approach as in Eq. 5, using the following correction quantities $R_{\text{cable}} = 0.6082 \Omega$ and $\Delta_{\text{GND}} = 0.1815 \text{ V}$ to correlate the **readback** readings with the I_a measurements of the DTB.

The subtest **readbackvbg** averages 10 **readback** runs to measure the band-gap reference voltage on the ROC and stores the values in a class data member for use in the next subtest.

The calibration of the band-gap reference voltage is performed in **getcalibratedvbg** and is based on data obtained in **readbackvbg**, **calibratevd**, and/or **calibrateva** (by default, **calibratevd** is preferred over **calibrateva**). Depending on the setting⁵ of the checkboxes **usecalvd** or **usecalva**, the band-gap reference voltage is determined as

$$V_{\text{BG}} = \frac{(\text{vbg} - p_0)}{2 \times p_1}, \quad (6)$$

where p_0 and p_1 refer to Eq. 5 applied either for vd or va. The factor 2 accounts for the doubled precision for vbg compared to vd or va. The band-gap reference voltage obtained with the **readback** test has been compared against a direct measurement with a probe needle. The **readback** results are found to be accurate at the 0.01 V level.

⁵It should be noted that checking *both* boxes will lead to sub-optimal behavior.

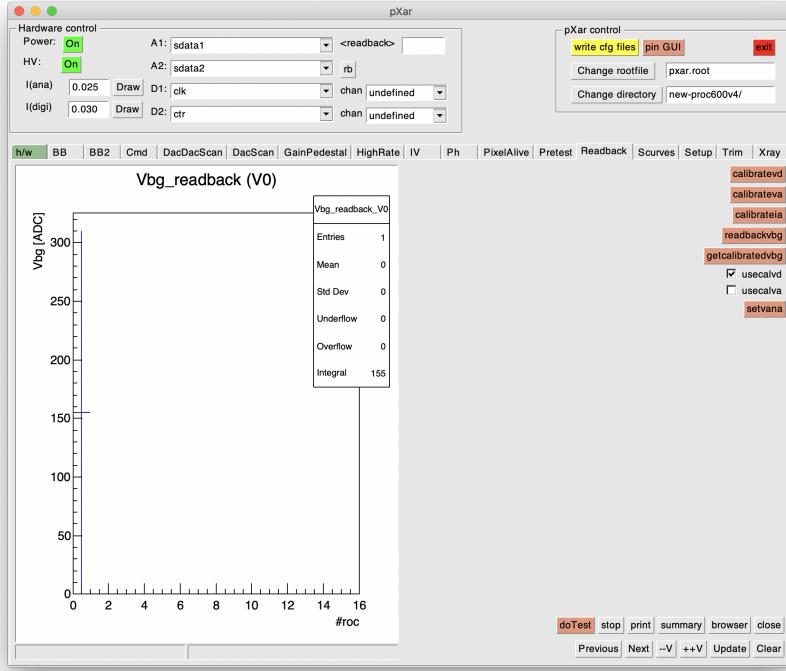


Figure 31: The Readback GUI tab after running `doTest`.

For the subtest `calibrateia`, each ROC is calibrated individually. As in the `Pretest:setvana` subtest, the current contribution of all other chips must be subtracted for the measured I_a . The calibration proceeds in two steps. First a quadratic relation between `VANA` and I_a is determined as

$$I_a(\text{VANA}) = q_0 + q_1 \times \text{VANA} + q_2 \times \text{VANA}^2 \quad (7)$$

and then a linear relation is determined between the `readback iana` ADC value and `VANA` as

$$\text{RB}(\text{iana}) = r_0 + r_1 \times \text{VANA}. \quad (8)$$

6.10 Xray (VCAL calibration)

The `Xray` test provides methods to determine the VCAL calibration. A radioactive source emitting photons with a known energy (possibly including additional background with different energies) is used to provide an well-known energy deposit into the sensor. From the pulse height read out, in combination with the gain-peDESTAL calibration, it is possible to determine the relation between VCAL and the equivalent electron charged to the energy deposit.

There are two subtests in the `Xray` tab: `phrun` and `ratescan`. The `doTest` subtest invokes only the `phrun` subtest. The goal of the `phrun` subtest is to measure x-ray hits during a user-specified duration (by default, `runseconds = 100 sec`) and trigger rate (by default, `trgfrequency = 100 kHz`). Because problematic pixels can derail the measurement, faulty pixels can be masked, using `maskhotpixels`, and/or a dedicated mask-file can be read in (and the one obtained can be saved by checking `savemaskfile`). The user input `source`

allows a specification of the histogram names used to store the readout. By default, the readout hits are only aggregated into histograms, not into a tree. This can be changed with the checkbox `filltree`. This subtest requires the calibration files of a previous `GainPedestal` calibration because it performs a conversion of the measured pulse height distribution into `VCAL` units.

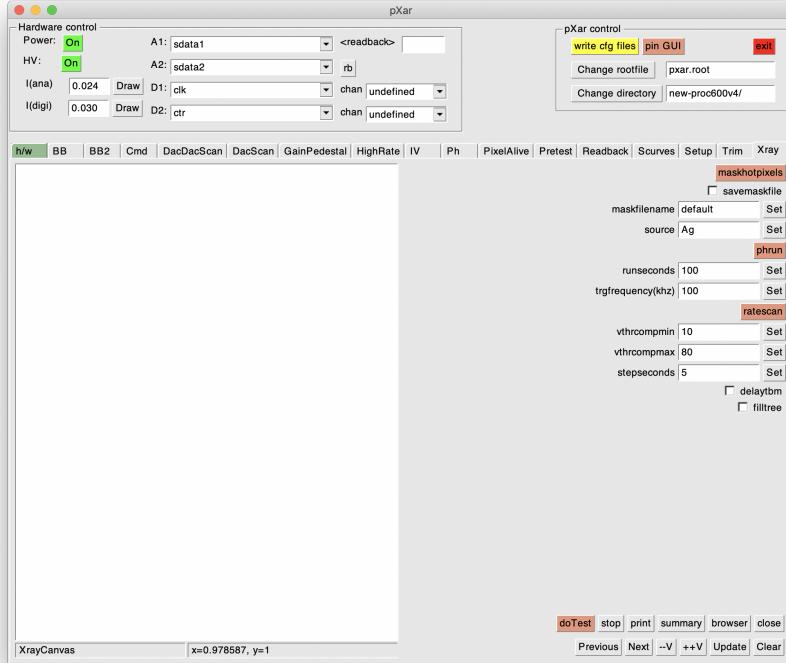


Figure 32: The `Xray` GUI tab before running any test.

The second subtest, `ratescan`, scans `VTHRCOMP` from the user-defined input `vthrcompmin` to `vthrcompmax` in steps of 1DAC unit, taking data for `stepseconds` seconds at each step. Increasing `VTHRCOMP` lowers the threshold and at some point the energy deposit of the x-ray-source will appear cross the threshold. This will be visible as an increase in the measured rate. In the analysis of the data, initially the mean number of hits per pixel is determined (for each ROC separately, as the illumination by the x-ray-source may be very inhomogeneous). Subsequently ideally, a ‘noise’ level would be determined and pixels with hit counts above that level would be masked. However, the current code requires some modifications to really achieve this. In reality, all hits are counted, including those from noisy pixels. The number of hits in the ROC is histogrammed vs. the `VTHRCOMP` value and fit with an error function to determine the `VTHRCOMP` threshold corresponding to the energy deposited by the x-ray-source. In the final step, each ROC is programmed with the corresponding `VTHRCOMP` DAC and a `VCAL` scurve scan is performed to determine the corresponding `VCAL` threshold (cf. Fig. 22 to motivate the determination first of a `VTHRCOMP` threshold and then a `VCAL` threshold). This `VCAL` calibration is independent of a prior `GainPedestal` calibration, since it is based on threshold scans only.

6.11 HighRate

The `HighRate` test provides methods to study the behavior of the DUT under high-rate radiation and with a large volume of readout data and to identify pixels with abnormally high hit rates. For this purpose, normally an xray-tube with variable current settings is employed.

The subtests available in the `HighRate` test include improved configuration possibilities (`trimhotpixels` and `maskhotpixels`) and the main subtests (`caldelscan`, `xpixelalive`, and `xnoisemaps`). The subtest `rundaq` allows for conventional data-taking without any specific test or calibration activity.

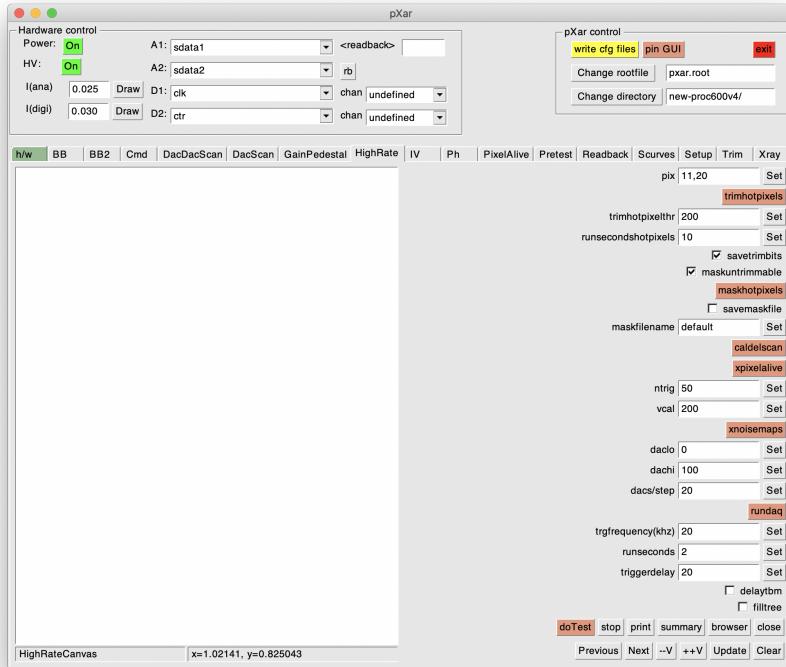


Figure 33: The `HighRate` GUI tab before running any test.

The subtest `trimhotpixels` relies on the `PixTest` method `trimHotPixels` (cf. section 4.2.4 for a detailed description) and allows to write the resulting trimbits to file (clicking the check box `savetrimbits`) and/or to mask pixels that are not trimmable (`maskuntrimmable`). The user-specified parameters `trimhotpixelthr` (= 200 MHz/cm² by default) specifies the hit rate threshold above which pixels are classified as ‘hot’. The parameter `runseconds` (= 20 sec by default) defines the run duration of the hot pixel determination. The trigger frequency of 100 kHz is not configurable with the parameter `trgfrequency(khz)` further down on the tab (that parameter applies to the `rundaq` subtest only).

The subtest `maskhotpixels` relies on the `PixTest` method `maskHotPixels` (cf. section 4.2.4 for a detailed description) and allows to write the resulting mask information to file (if the check box `savemaskfile` is checked). This subtest operates with a fixed (not configurable) threshold of 10⁻⁵ hit occupancy to define ‘hot’ pixels.

The subtest `caldelscan` starts with a short (with `ntrig` = 3) CALDEL scan for the user-

defined pixel `pix` (= 11,20 by default) and `VCAL` (by default `VCAL` = 200, in the low range). This scan is used to extract lower and upper bounds on `CALDEL`, defined in terms of an efficiency level above 80% (with a reserve of `CALDEL`= 1DAC unit). Hot pixels are masked and the pattern generator setup is modified to have all resets removed (i.e. `resetroc` and `resettbm` are excluded). The `CALDEL` range determined in the short scan is scanned with 20 steps and the ROC efficiency is determined with the entire ROC enabled for readout. The hitmaps of these runs are stored in 2D histograms with names `HR_xeff_CalDelScan_stepX` for each step X. The final result of the subtest is a histogram showing the ROC readout efficiency vs. the `CALDEL` setting.

For the subtest `xpixelalive` the parameter `triggerdelay` (= 20 by default) is inserted into the testboard signal delay settings. Furthermore, the reset signals `resetroc` and `resettbm` are removed from the pattern generator setup. After this configuration, an `efficiencMap` is run with the entire DUT enabled and unmasked. The 2D maps for the hit efficiency and the additional xray hits are returned as `highRate_CYY` and `highRate_xraymap_CYY` for ROC YY, respectively. For the analysis, a ‘fiducial’ area of the ROC is defined by excluding the edge columns and the top row with different pixel sizes. Hits in the fiducial area are counted separately from hits in the overall ROC area. ‘Problematic’ and ‘dead’ pixels are identified by having less than `ntrig` hits per pixel or no hits at all, respectively. The results of this test are stored in 1D histograms `HR_Overall_Efficiency_CYY` and `HR_Fiducial_Efficiency_CYY` for ROC YY. In the summary printout, the hit rate is calculated for the effective sensor size (accounting for the increased pixel dimension in the edge columns and top row).

The subtest `xnoisemaps` applies the same changes as above to the testboard delay settings and removes the reset signals from the pattern generator. With these settings, it runs a `scurveMaps(VCAL, ...)`, and the subsequent analysis as described in section 4.2.4, with the entire DUT unmasked and enabled. The results histograms have `xNoiseMap` has part of the name.

6.12 IV (leakage current test)

The IV test provides the capability to measure the leakage current as a function of the applied high voltage.

The IV test relies on a functional interface (via a serial interface) to the HV source meter (by default a Keithley 2410, but other models have been implemented as well). The code for this is implemented in `devices/hvSupply.h` and `devices/hvSupply.keithley2410.cc`. Important note: You must configure `pXar` with `cmake -DBUILD_HVSUPPLY=Keithley2410` for the capability to control a HV source meter (other options are `BUILD_HVSUPPLY=Iseg` or `BUILD_HVSUPPLY=Keithley237`). You may have to configure the USB-serial port interface (most likely, the computer you run `pXar` on will not have a serial port anymore) with something similar as in `sudo main/portSetUp.sh` (note the remark in the script about `/dev/ttyUSB` or `/dev/ttyUSB1`). It is furthermore essential to ensure that your HV source meter is configured consistently for the serial communication. `pXar` helps with the following printout:

```
[17:12:01.945]      INFO: Starting IV curve measurement...
[17:12:01.945]      INFO: Make sure the Keithley2410 is using the following RS232 settings:
[17:12:01.945]      INFO: |Baud Rate: 57600
[17:12:01.945]      INFO: |Bits: 8
```

```
[17:12:01.945]      INFO: |Parity: ODD
[17:12:01.945]      INFO: |Terminator: <CR>+<LF>
[17:12:01.945]      INFO: |Control Flow: XON-XOFF
```

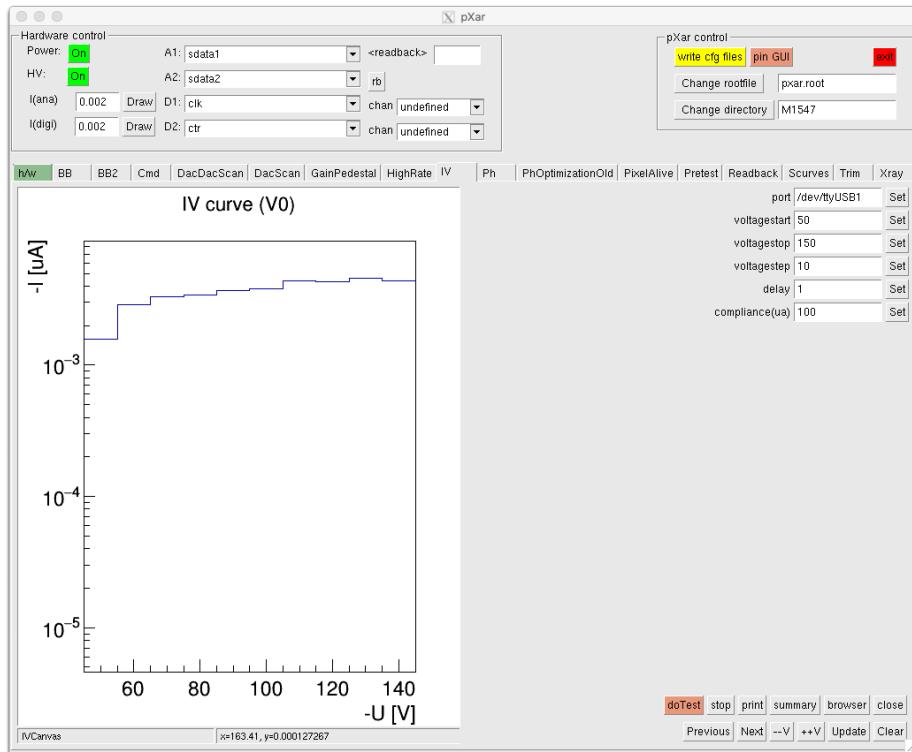


Figure 34: The IV GUI tab after running a short dummy test (no DUT attached).

The configuration possibilities of the IV test are illustrated in Fig. 34. The first mandatory configuration is the definition of the serial interface with `port` (in the example of Fig. 34 this is `/dev/ttUSB1`; trial and error will normally yield the correction one quickly; note that normally this can be diagnosed independently of whether the HV source meter's serial communication setup is correct, see above). The user-definable input fields for (all positive) `voltagestart`, `voltagestop`, and `voltagestep` configure the scan of the applied HV. Note that the start and endpoint of the scan are specified with positive numbers, but are passed as negative voltages to the HV source meter. The step is internally re-configured to be to move in the correcte direction in case the start point is larger (more positive) than the endpoint. The parameter `delay` specifies the delay used in the HV source meter before measuring the leakage current. It is possible to set a compliance value for the HV source meter with `compliance`.

The primary output of the IV test is an ASCII file with the name `ivCurve.log` of the following form (the exact printout depends on the configuration):

```
# IV test from Fri, 18 Sep 2020 17:12:05 +0200 (CEST) +819638000 nsec
#Test Parameters:
#  HV Supply:      Keithley2410
#  Voltage Start:  50
```

```

# Voltage Stop:    150
# Voltage Step:   10
# Delay(s):       1
# Compliance(uA): 100
#voltage(V)      current(A)      timestamp
-50.000          -1.576804e-09  2020-09-18 15:12:05.819638000Z
-60.000          -2.880349e-09  2020-09-18 15:12:07.802595001Z
-70.000          -3.316586e-09  2020-09-18 15:12:09.897432001Z
-80.000          -3.443464e-09  2020-09-18 15:12:11.880461001Z
-90.000          -3.713189e-09  2020-09-18 15:12:13.863308001Z
-100.000         -3.793337e-09  2020-09-18 15:12:15.814259000Z
-110.000         -4.384717e-09  2020-09-18 15:12:17.781142001Z
-120.000         -4.348261e-09  2020-09-18 15:12:19.764071000Z
-130.000         -4.617388e-09  2020-09-18 15:12:21.811133000Z
-140.000         -4.358077e-09  2020-09-18 15:12:23.906016000Z
-150.000         -4.456014e-09  2020-09-18 15:12:25.945099000Z
#Sweep completed Normally

```

In addition, the IV curve is dumped into a ROOT histogram with the name `IVcurve_V0`.

6.13 FullTest

The suite of tests that is run in the course of a complete pixel module qualification is called `FullTest`. It is a sequential test suite that is run multiple times at different temperatures. It is composed of the following tests, which are described in more detail in the preceding sections.

Table 2: Overview of all tests implemented in `pXar`. The order of the tests in this table does not correspond to their arrangement in `pXar`, but rather to their relevance. Notes: In `testparameters.dat`, the test name appears without ‘`Test`’. Test entries marked with (*) are contained in the `doTest` entry of the corresponding test.

Test name	Entry	Type	Description
Pretest	<code>programroc</code> (*) <code>setvana</code> (*) <code>setvthrcmpcaldel</code> (*) <code>findtiming</code> (*) <code>findworkingpixel</code> (*)	basic test configuration configuration configuration functionality	ROCs can be programmed? basic configuration basic configuration configure timing for >TBM08B same working pixel in all ROCs
Alive	<code>alivetest</code> (*) <code>masktest</code> (*) <code>addressdecodingtest</code> (*)	functionality functionality functionality	test response of pixels test masking of pixels test address of pixels
trim	<code>trim</code> (*) <code>trimbits</code> (*)	optimization functionality	unify pixel thresholds test trim bits
Ph	<code>optimize</code> (*) <code>phmap</code>	optimization characterization	ensure maximum usage of ADC range measure 2D pulse height map
GainPedestal	<code>measure</code> (*) <code>fit</code> (*)	optimization optimization	measure pulse height vs. <code>VCAL</code> parametrize pulse height vs. <code>VCAL</code>
BB2	<code>dotest</code>	functionality	test bump bond connectivity

6.14 Additional tests

In addition to the tests discussed above, other 'tests' are available in `pXar`. These include

- `PixTestCmd` provides the command-line interface to the DUT developed and maintained by W. Erdmann.
- `PixTestDaq` runs data acquisition with the DUT. The difference to the DAQ in the `HighRate` test is the possibility to apply a clock-cycle stretching `clockstretch` that helps in collecting more statistics if exposed to a low-fluence radiation source.
- `PixTestPattern`
-

References

- [1] Spannagel, Simon and Meier, Beat and Perrey, Hanno, “The pxarCore Library - Technical documentation, reference manual and sample applications”. 2016. CMS NOTE - 2016/001.
- [2] Meier, Frank et al., “PIXEL DTB Testboard for Readout Chips Manual”.
<https://twiki.cern.ch/twiki/bin/viewauth/CMS/PixelDTB>, 2015.
- [3] Hans-Christian Kästli, “The new ROC for the Pixel Upgrade: results of first test on first submission”. <https://indico.cern.ch/getFile.py/access?contribId=5&resId=0&materialId=slides&confId=180238>, 2012.
- [4] Hans-Christian Kästli, “The new ROC for the Pixel Upgrade: results of first test on first submission”. <https://indico.cern.ch/getFile.py/access?contribId=1&resId=0&materialId=slides&confId=197212>, 2012.
- [5] Hans-Christian Kästli, “Design changes of ROC and thing to verify”.
https://indico.cern.ch/event/305865/contributions/1673362/attachments/581758/800840/Kaestli_PSI46digV2.1_28032014.pdf, 2014.
- [6] Hans-Christian Kästli, “PROC600 V4-to-V3 differences and results of additional V4 tests”. https://indico.cern.ch/event/835792/contributions/3503938/attachments/1886749/3110567/2019_07_26_PROC_ECR.pdf, 2019.
- [7] V. R. Tavolaro, “Study of the differential Higgs boson production in the diphoton decay channel with CMS and characterisation and calibration of detector modules for the Phase1 upgrade of the pixel detector”. PhD thesis, Zurich, ETH, 2018.