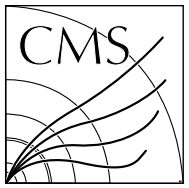
Available on CMS information server

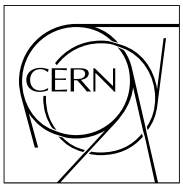
CMS NOTE -2016/001



The Compact Muon Solenoid Experiment

CMS Note

Mailing address: CMS CERN, CH-1211 GENEVA 23, Switzerland



16 October 2015 (v3, 25 February 2016)

The pxarCore Library - Technical Documentation, Reference Manual, and Sample Applications

Simon Spannagel, Beat Meier, Hanno Perrey

Abstract

This document describes the pxarCore library as of version 2.7, a versatile cross-platform library to program and read out devices featuring PSI46-type readout chips (ROCs) via the PSI digital test beoard (DTB). The design principles and features are presented together with an overview over possible configurations and use cases. The pxarCore library is built as flexible hardware interface and is capable of programming and reading out various combinations of ROCs. Most of the DTB firmware complexity is abstracted to allow users to address the attached devices via a simple yet versatile interface.

The pxarCore Library

Technical Documentation, Reference Manual, and Sample Applications

Simon Spannagel, Beat Meier, Hanno Perrey

February 22, 2016

This document describes the pxarCore library as of version 2.7, a versatile cross-platform library to program and read out devices featuring PSI46-type readout chips (ROCs) via the PSI digital test board (DTB). The design principles and features are presented together with an overview over possible configurations and use cases. The pxarCore library is built as flexible hardware interface and is capable of programming and reading out various combinations of ROCs. Most of the DTB firmware complexity is abstracted to allow users to address the attached devices via a simple yet versatile interface.

Contents

List of Tables

List of Figures

1. Introduction

The pxarCore library is currently used in all institutes contributing to the production of the Compact Muon Solenoid (CMS) Phase I Pixel Detector [1] for pixel module functionality tests, quality control, calibration of the detector parameters at operational temperature as well as X-Ray measurements for absolute energy calibration, single hit efficiency tests and others [2]. This document does not describe the actual test procedures and optimization algorithms but focuses on the configuration and operation of the readout electronics as well as on the trigger and data acquisition (TDAQ) of the detector modules. Various software packages are based on the pxarCore library such as the standard test suite *pXar* including test routines and a graphical user interface, or the *pyXar* package [3] implementing a convenient detector test framework in Python.

This document describes the pxarCore library, its working principles and functions, and gives an introduction on how to build applications using the library as back-end for reading out devices equipped with PSI46-type readout chips (ROCs). It is intended as reference manual for both users and future developers or maintainers of the software.

In addition to the pxarCore software, the microprocessor code of the digital test board (DTB) firmware is described, since it has been developed together with the software to optimize the workflow and minimize time consumption for tests.

The document is structured as follows. Chapter 2 describes the prerequisites for compiling and executing the software and gives a reference of all possible build options. Chapter 3 provides an introduction to the DTB as the readout electronics the software is designed for. This includes descriptions of both basic firmware functionalities and the software running on the internal microprocessor. The software architecture of the pxarCore library is described in detail in Chapter 4, while Chapter 5 gives an introduction on how to use the library interface. Code examples and additional resources are given in Chapter 6, while Chapter 7 provides some comprehensible code examples on how to accomplish various typical tasks.

Chapter 8 describes the integration of the pxarCore library into the EUDAQ data acquisition framework for test beam deployment of detector modules.

Finally, the Appendix A provides a reference of all pxarCore internal dictionaries for registers, digital-to-analog converter (DAC) parameters, trigger sources, device names, and other settings.

This document references the pxarCore library version 2.7 series, which has been released on January 15, 2016 [4].

2. Installation

The pxarCore library is a multi-platform C++ software for x86-compatible processors with support for Linux, OS X, and Windows operating systems. The library is distributed as source code only and has to be compiled on the desired platform as described in the following sections.

2.1. Prerequisites

pxarCore has relatively few prerequisites and dependencies on other software, but some features do rely on external packages. The required software packages are listed in the following.

C++ compiler Compilation of the pxarCore source code requires a C++99 compliant compiler and has been tested with GCC, Clang, and MSVC (Visual Studio 2012 and later) on Linux (Scientific Linux 5 and 6, Ubuntu 14.04–15.10), OS X 10.7 and later; and Windows 7 and 8.

CMake In order to configure the pxarCore build process and to generate the build files, the cross-platform, open-source build system CMake is used. CMake is available for all major operating systems from [5], and version 2.8.12 or above is required. On most Linux distributions it can usually be installed via the built-in package manager (aptitude/apt-get/yum etc.) and on OS X using packages provided by e.g. the MacPorts or Fink projects.

libusb 1.0 In order to communicate with the digital test board (DTB) over the USB port, the libusb library version 1.0 as well as its headers are needed.

On Mac OS X, the library can be installed using Fink or MacPorts. If using MacPorts it might be necessary to install the `libusb-compat` package. On Linux libusb might already be installed, otherwise the built-in package manager should be used to install it. The necessary headers are often shipped separately from the library, and the corresponding packages may be named `libusb-1.0-dev` or `libusb-1.0-devel`.

FDT2XX / FTDI library The DTB features a FTDI USB chip which requires a corresponding 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 be installed via the package manager. The package is typically called `libftdi-dev` or similar. However, this driver has shown performance problems in the past and should only be used as fallback option 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 [6]. Version 1.1.12 or higher is required, and the appropriate files for the operating system and system architecture have to be downloaded. The driver library and header files should be placed in the usual install

locations of the operating system, e.g. for a Linux distribution usually `/usr/local/lib` and `/usr/local/include` are a good choice. The following files from the package are required:

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

It might be necessary to create a symbolic link pointing to `libftd2xx.so` omitting any versioning numbers such as

```
libftd2xx.so.1.1.12 -> libftd2xx.so
```

It has been reported, that some versions of the FTD2XX library require a recent version of the glibc library and are thus not compatible with all Linux distributions.

On OS X it might be necessary to remove the FTDI driver provided by the operating system by removing or renaming the file

```
/System/Library/Extensions/IOUSBFamily.kext/
                                    Contents/PlugIns/AppleUSBFTDI.kext
```

Python, Cython, Numpy If the Python bindings of the pxarCore library are to be used e.g. for pyXar [3], for the Python command line interface (CLI) (cf. Section 6.1), or for writing short and simple scripts to program the detector and perform tests, the Python 2.7 libraries, as well as Cython [7] in version 0.19 or later, and the Python-Numpy package [8] are required.

Building pxarCore with GUI By default, the pxarCore library is built together with the pXar graphical user interface (GUI) and test suite, which requires the ROOT package [9] for histogramming. ROOT 5.34 as well as ROOT 6 are supported.

2.2. Downloading the source code

The pxarCore source code is hosted on github, together with the pXar GUI source code. The recommended way to obtain the software is through git [10], since this allows to easily update to more recent versions and to rebuild the software. In order to obtain the source code, the git repository has to be cloned once via

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

or one of the available GUIs for git. This will download the repository and its full development history into a newly created folder `pxar` and add the main development repository as git remote `origin`. To update to the latest upstream version available, the following command should be used inside the pxar directory:

```
git pull origin master
```

For production systems like test stands or probe stations as well as in test beam experiments it is highly recommended to stick with tagged stable versions instead of always updating to the latest commit on the master branch. Tags can be checked out using

```
# Fetch all new tags and commits
git fetch origin
# List the available tags:
git tag
# Check out the selected tag:
git checkout v2.7.X
```

Alternatively, e.g. for installations without direct internet access, source tarball files for tags and releases can be directly downloaded from the repository [4].

2.3. Configuration via CMake

In order to configure the build and to check for the presence of all dependencies, a new folder (e.g. `build`) should be created and CMake run from there, pointing to the `CMakeLists.txt` file in the root directory of the pxar repository. Graphical interfaces to CMake or the MSVC project manager can be used instead of the command line tool. On Linux, the following commands create a new folder and execute CMake:

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

CMake automatically searches for all required packages and verifies that all dependencies are met using the `CMakeLists.txt` script. To select the modules to be built, parameters can be supplied to CMake via

```
cmake -D<parameter>=<ON/OFF> ..
```

The corresponding settings are cached, so that they will again be used next time CMake is run. The following CMake parameters relevant for the pxarCore library are available:

BUILD_pxarui: (default: `ON`) Switch on/off building of the pxar GUI and all its tests. This option requires the ROOT libraries to be available. **In order to only build the pxarCore library, this option should be turned off.**

BUILD_python: (default: `OFF`) Build the optional Python bindings for the pxarCore library. This allows you to use the Python CLI, the pyXar software and custom Python scripts accessing pxarCore functions. This option requires Python, Numpy and Cython as additional dependencies (see Section 2.1).

BUILD_tools: (default: `OFF`) Build additional tools such as the `flash` tool to update the DTB firmware or the `decoder` executable running software-driven unit tests on the pxarCore decoder modules.

USE_FTD2XX: (default: `ON`) Switch to choose between the open source library for FTDI chips `libfdti` and the closed source vendor supplied `libFTD2XX`. The latter one has shown better performance than the open source library and is recommended for now. If both libraries are found on a system, `libFTD2XX` is preferred, otherwise the one available is used automatically.

INTERFACE_USB: (default: `ON`) Switching on/off compilation of the USB interface classes for DTB communication. If this switch is turned off the resulting pxarCore library will not be able to access any DTB via USB!

INTERFACE_ETH: (default: `OFF`) Switching on/off compilation of the Gigabit Ethernet interface classes for DTB communication. Currently Ethernet is not supported by the DTB firmware. Building the Ethernet interface requires the PCap libraries as external dependency.

BUILD_dtbemulator: (default: `OFF`) Switch off all real interfaces but build a emulator hardware class on remote procedure call (RPC) level instead. This is equivalent to setting both `-DINTERFACE_USB=OFF` and `-DINTERFACE_ETH=OFF`. The DTB emulator class will deliver test data just as a real device would. This can be used to test and debug the software without a physical device attached (cf. Section 4.6).

Thus, only building the pxarCore library and its Python bindings can be achieved via

```
cmake -DBUILD_pxarui=OFF -DBUILD_python=ON ..
```

By default, the installation path of the library is the `lib` folder inside the source directory. In order to install to a different location, the `INSTALL_PREFIX` parameter for CMake can be used. e.g.

```
cmake -DINSTALL_PREFIX=/usr/local ..
```

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

In case of configuration problems or errors it is recommended to delete the build directory and to re-run CMake from scratch:

```
rm -r build/
```

In case this does not solve the configuration problem, the developers can be contacted via the github bug tracker of the project [11]. A full CMake output from a clean build directory should be included in any bug report in order to provide the necessary information for receiving help.

2.4. Compilation

On Windows, the cmake command produces a Visual Studio project which can be compiled from within the MSVC GUI. On Linux and OS X, the GNU make utility is used for the library compilation and thus has to be installed, either via the package manager (Linux) or the Xcode developer tools (OS X). With make, the compilation process is started by executing

```
make [-jX] [VERBOSE=1]
make install
```

after a successful CMake execution. Here, the optional parameter `-jX` allows to simultaneously execute the compilation on X CPU cores, while `VERBOSE=1` switches the make utility to high verbosity level.

In case of build errors it is recommended to delete the build directory and to re-run CMake and make from scratch:

```
rm -r build/
```

In case this does not solve the build problem, the developers can be contacted via the github bug tracker of the project [11]. The full CMake and make output from a clean build directory should be included in the bug report.

2.5. DTB firmware

The pxarCore library is always build against a specific version of the DTB firmware. It needs to include the correct interface headers to allow communication and have access to all microprocessor functionality needed for its routines (cf. Section 3.3).

pxarCore tries to check the firmware version at start-up and will report version mismatches. For reference, the required firmware version is stored in the `CMakeLists.txt` file in the pxar repository. The output of a successful comparison will be similar to

```
INFO: RPC call hashes of host and DTB match: 1895922356
```

while a failing hash comparison results in an error:

```
WARNING: RPC Call hashes of DTB and Host do not match!
CRITICAL: <hal.cc/CheckCompatibility:L383> Please update your DTB with
          the correct flash file.
CRITICAL: <hal.cc/CheckCompatibility:L384> Get Firmware v4.6 from
          https://github.com/psi46/pixel-dtb-firmware/tree/master/FLASH
```

All flash files for the different DTB firmware versions can be obtained from [12]. When downloading files from the github website it has to be ensured that the actual raw file is downloaded instead of a HTML document created by the github web server. The compatible version for pxarCore version 2.7 is the DTB firmware version 4.6.

The file has then to be loaded onto the DTB Field Programmable Gate Array (FPGA) with one of the following statements (assuming the current directory being the `pxar` folder):

```
./bin/pXar -f /path/to/pixel-dtb-firmware/FLASH/FLASHFILE
./bin/flash /path/to/pixel-dtb-firmware/FLASH/FLASHFILE
```

where `FLASHFILE` should be replaced with the appropriate firmware file name. The download to the FPGA will take a while, and the on-screen instructions should be

followed. After all 4 status LEDs are off, the DTB should be power-cycled. Without a power cycle the new firmware is only stored in the EEPROM but not yet loaded into the FPGA. The `flash` executable is compiled when enabling the additional tool compilation as described in Section 2.3.

3. The Digital Test Board (DTB)

The DTB is the readout electronics for lab and test beam operation of PSI46 devices such as single readout chip (ROC) assemblies, Compact Muon Solenoid (CMS) Pixel detector modules or beam telescopes. It features an Altera FPGA, 2x64 MB DDR2 RAM, as well as USB2.0 and Gigabit Ethernet ports.

A crosspoint switch allows monitoring of signals via several LEMO outputs and a 68-pin SCSI cable connects the detector devices. Data lines are separated by ground for defined impedance. This also allows the usage of long cables, e.g. for mounting in X-Ray tubes where space constraints prevent the user from placing the DTB directly next to the detector module.

Generic IO pins are available on the board, which could be used for connecting external devices such as temperature sensors. This, however, would require the appropriate firmware modules to be implemented by the user.

The intergrated analog-digital converter (ADC) can be used for both the digitization of input data from analog detector modules, and to sample signals provided by the crosspoint switch. The crosspoint switch can map signals from in- and outputs of the DTB to either the LEMO connectors or to the internal ADC which can sample the data and thus acts as "digital scope". The crosspoint switch configuration is changeable remotely via software.

Deserializer modules for both 160 MHz and 400 MHz signals for single ROC and module readout with Token Bit Manager (TBM) are provided, and inputs for external trigger signals and an externally generated clock are available.

3.1. Connectors

The available sockets and connectors on the DTB are shown in Figures 1 and 2. The power LED should be on as soon as the DTB is connected to a 6 V power supply. When connecting the DTB to the power supply, the four status LEDs light up in a circular pattern indicating a correctly loaded firmware. If the status LEDs do not come on, the firmware has to be flashed via the JTAG connector on the board since no firmware could be loaded from the EEPROM.

During operation, the detector power LED is turned on as soon as the supply voltages (analog and digital) to the detector are activated (automatically at start-up via `pxarCore::initDUT()` or manually using `pxarCore::Pon()`). The high voltage (HV) LED turns on with the sensor bias being switched on (via `pxarCore::HVon()`), and the toggling of the HV relay can be heard.

The status LEDs are used to signal different states of the DTB. LED 1 is active while a data acquisition (DAQ) session is running. LED 3 is used to signal activity of the trigger test loops (cf. Section 4.4.2). The other LEDs are currently unused.

The digital and analog probe outputs *D1*, *D2* and *A1*, *A2*, respectively, can be used to monitor different signals as described in Section 5.2.

Figure 1: The DTB front panel featuring (from left to right) the USB2.0 port, the Gigabit Ethernet connector, the digital scope outputs *D1, D2*, the clock and trigger inputs, four status LEDs, the HV and detector power LED, the AC connector, and the DTB power and CRC LEDs.

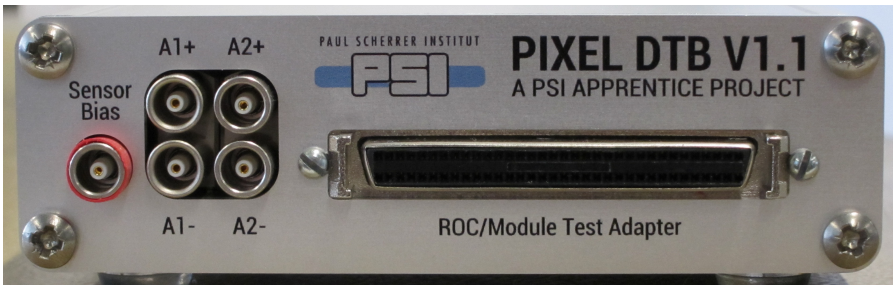


Figure 2: The DTB rear panel featuring the HV sensor bias port, the two analog differential signal probes, and the 68-pin SCSI connector.

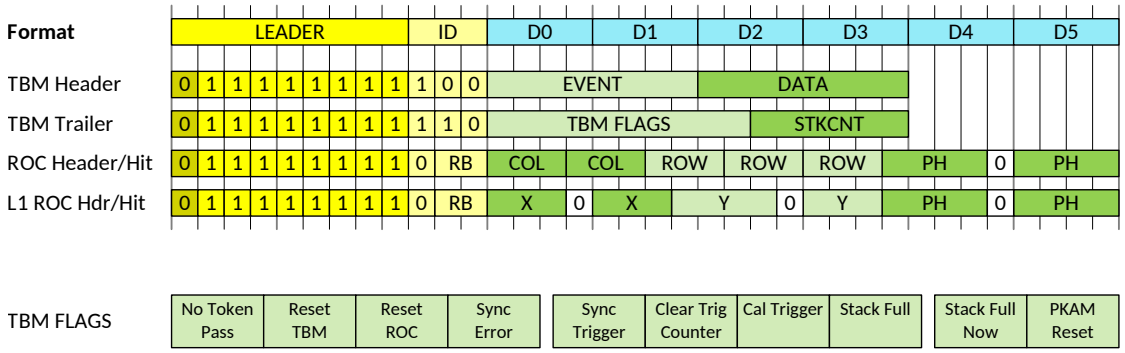


Figure 3: Summary of the output data format of CMS Phase I Pixel Detector modules. The data stream from the ROCs is prepended and appended with the header and trailer from the TBM, the pixel address encoding differs for the ROC of the detector layer 1 and layers 2-4.

3.2. Deserializer Modules

There are different firmware modules available sampling the incoming data stream from the `sdata` lines, depending on the DUT attached to the DTB. The correct module has to be set up and activated in order to receive data from the detector. The detector data format is summarized in Figure 3.

All modules process the DTB `sdata` input channel from the device under test (DUT) and write their output directly into a ring buffer set up in the DTB RAM via direct memory access (DMA). This buffer has to be allocated beforehand as described in Section 4.2.

DESER160: The 160 MHz deserializer module samples data coming directly from one or more ROCs. It regards the Token In and Token Out signals and chops the datastream accordingly. In order to properly sample the signal, the `deser160phase` DTB parameter specifying the relative phase for the sampling point has to be set correctly. This can be done best by operating a single digital ROC and looking at the sampled signal. The phase should then be adjusted until the correct ROC header patterns `0x7f8` can be recognized. This is automatically done e.g. by the pXar *Setup Test*.

DESER400: The 400 MHz deserializer module features a 5-to-4 bit decoder to reverse the encoding of the TBM. Subsequently, the signal is split into the two contained 160 MHz signals from the TBM cores, and a decoder module decodes the data by identifying ROC headers and pixel hits. The data format is changed and added with identifier bits marking each part of the data. These identifiers are used by the pxarCore decoder modules later on (cf. Section 4.3). This however means, that the original `0x7f8` patterns from the ROC headers are removed, and the former header bits are used for information from the deserializer, such as the XOR eye pattern of the 400 MHz signal as indicated in Figure 4. The DESER400 is self-aligning using

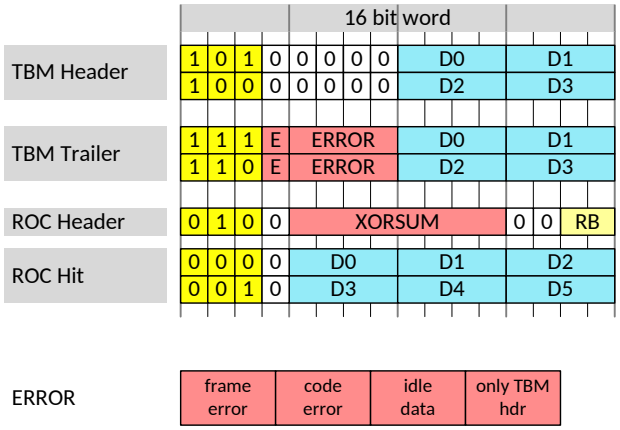


Figure 4: Output data format of the DESER400 deserializer module The first four bits of each 16 bit word are identification markers (yellow), while the input data occupy the lower bits (blue). Additional internal error states of the deserializer module are inserted into unused bits (red).

the idle patterns sent by the TBMs. The sampling frequency (measurement time) of the phase detector can be configured using the `deser400rate` DTB parameter with the possible settings

- 0: 75 ns
- 1: 175 ns (default)
- 2: 375 ns
- 3: 775 ns.

For multi-channel TBMs (`TBM09` or `TBM10` type) two DESER400 modules are activated, and for Layer 1 modules featuring two TBM10 chips, eight data lanes with four DESER400 modules are available.

ADC: The `sdata` signal can be re-routed to the DTB-internal ADC via the crosspoint switch. The ADC then samples the incoming analog 40 MHz ROC data and writes the data into memory. This allows operation of analog PSI46V2 devices with the DTB.

3.3. The DTB Microprocessor Interface

This section is intended to give a detailed description of the available functions of the DTB microprocessor. For operating DUTs using the pxarCore library, the information in this section is not necessary but might however help in understanding problems occurring during programming or readout of the devices.

The FPGA of the DTB features an emulated Nios II microprocessor [13] (the so-called softcore CPU) which is capable of executing user code, providing interfaces to

the FPGA functionality and the communication interfaces of the DTB (USB, Ethernet). The Softcore CPU is running at a clock speed of 75 MHz and is thus not powerful and fast enough to do data processing or decoding on the DTB. Therefore, all data recorded by the DTB is *always* transferred to the host PC and analyzed there.

However, the Softcore CPU is very convenient for executing simple commands such as starting the pattern generator or reprogramming a DAC. Therefore it is mainly used to control the functionality of the DTB FPGA, to start and stop data acquisition and to send Inter-Integrated Circuit (I^2C) commands to the DUT.

A set of functions has been implemented to ease the recording of large amounts of data, scanning through various parameters without too much communication overhead. These functions are described in the following sections. Section 3.3.1 introduces the Nios cache for detector parameters while Sections 3.3.2–3.3.4 describe the implemented Nios functions exploiting these capabilities for performing fast loops over multiple pixels and DAC settings.

3.3.1. Mask and Trim Bit Caching

The pixel unit cells (PUCs) of the PSI46 ROC generation only have one register for the mask/enable bit and the four-bit pixel-by-pixel threshold adjustment (the so-called *trim bits*). After chip start-up, all PUCs have to be programmed with appropriate trim bits to provide a uniform threshold over the whole ROC. However, these settings are lost when masking a pixel since this PUC register is overwritten when setting the mask bit. Thus when re-enabling a previously masked pixel the trim bits need to be programmed again. If the trim values are only known on the host PC side this becomes problematic and causes a large communication overhead when re-trimming every pixel.

To overcome this limitation, all trim bits are cached in the DTB RAM and are accessible to the Nios II Softcore CPU. This allows to mask the full chip and then quickly trim single pixels without communication overhead. The default behavior of the test loops described in Section 3.3.2 is to only enable and trim one pixel at a time keeping all others masked. This behavior can be disabled using the flag FLAG_FORCE_UNMASKED as described in Section 5.3.

In order to cache all trim and mask bit information, the Nios II needs to learn about the devices to be programmed. First, the I^2C addresses of all devices of interest have to be provided. Calling the function

```
CTestboard::SetI2CAddresses(vector<uint8_t> &roc_i2c)
```

sets up the data I^2C storage structures in the Nios II stack. It stores all ROC I^2C addresses to be accessed later by test loop functions for retrieving data for a specific ROC. Then, for every ROC configured via `SetI2CAddresses(...)` the trim values have to be provided using

```
CTestboard::SetTrimValues(uint8_t roc_i2c, vector<uint8_t> &trimvalues)
```

This function uploads all trim values of the ROC defined by the parameter `roc_i2c` to the Nios II storage. Trim values should be provided as linearized vector ordered according to their pixel address (col 0, row 0; col 0, row 1; etc.). The value represents the four trim bits for every PUC. Values larger than 15 are interpreted as *masked* and

the mask bit of the respective PUC will be set.

When using the pxarCore library, all trim and mask bits are automatically transferred and updated from the DUT object (cf. Section 4.4) before a test loop function is called.

3.3.2. Test Loops

The so called *Test Loops* are a set of Nios II interface commands which implement often-used loops over pixels or digital-to-analog converter (DAC) parameters directly in the Nios II. Instead of disabling one pixel and enabling the next one via the host PC, the respective I^2C commands are sent directly from the Nios II Softcore CPU. This comes with a major speed improvement compared to execution on the host PC via USB, which has a latency of a few milliseconds for every block of commands sent to the DTB.

The functions are designed to be easy to handle, both when using pxarCore or just interfacing the firmware directly. The naming scheme describes the scope as well as the action of the command:

`CTestboard::Loop<# ROCS><# Pixels><Action>(...)`

For example calling the function `LoopSingleRocAllPixelsCalibrate(...)` would send calibrate pulses to all pixels of one ROC, while calling the test loop function `MultiRocOnePixelDacDacScan(...)` would perform a two-dimensional DAC-DAC scan for one pixel on multiple ROCs of a module in parallel.

Before the test loops can be used, the trim bit information has to be cached. This information is provided via the functions described in Section 3.3.1.

The behavior of the test loops can be influenced by using the pxarCore flags which will be described in Section 5.3. This allows to e.g. perform cross talk measurements or to send the calibration pulse through the sensor pad.

When using the pxarCore library, the appropriate and most efficient loop is picked automatically via the loop expansion described in Section 4.4.3. The user just has to set up the DUT and request certain test data via the application programming interface (API) functions, e.g. pulse height information for all values of a DAC and for all pixels of the attached device.

Depending on their functionality, the test loops expect a certain set of parameters. Always required are the I^2C address(es) of the device(s) to be tested, the number of trigger signals to be send for every step, and the flags with which this loops should be executed. For one-dimensional DAC scans in addition the DAC register, range and step size has to be provided, in case of two-dimensional DAC scans this set of parameters also has to be provided for the second DAC to be varied.

The test loops allow DAC scans with a step size different from one (*sparse scans*, cf. Section 4.4.2).

3.3.3. Loop Interrupts

Running the test loops might produce a lot of readout data, especially since the data volume increases linearly with the number of triggers, DAC scanning range or the number

of pixels to be tested. The DTB RAM is limited to 128 MB of which about 90 MB are available to the ring-buffered DAQ storage.

Since the Nios II Softcore is not capable of running multiple threads, it is not possible to read out data from the ring buffer while a test loop is running since both operations require access to the RPC interface. The solution chosen is to interrupt the test loop whenever the fill level of the RAM has reached a certain limit and return to the host PC. The RAM can then be read out and the loop resumed. This approach minimizes communication overhead, maximizes test execution speed, and does not require any additional monitoring or controlling on the host PC side.

In the test loops, the fill status of the DAQ ring buffer is checked before each block of triggers is sent. If it is above the threshold of 85% (set by `LOOP_MAX_FILLLEVEL` in the firmware) the DAQ is suspended, all loop iterators such as the current DAC values and the address of the pixel which is currently calibrated, are stored, and an interrupt flag is set by the function `LoopInterruptStore(...)`. The RPC test loop function returns `false` signaling an interrupted loop execution. The host PC software now has to fetch all buffered data and can then simply invoke the same test loop function again. At the loop startup, the function `LoopInterruptResume(...)` is called to check for any previously interrupted loops and to load the iteration parameters again. Every test loop function has a unique ID to ensure that the correct loop is resumed. This means, if one loop command is interrupted and another loop command called afterwards, this loop will reset the interrupt status and start from the beginning. The total number of interrupts and resumes is limited to 150 as a precaution against endless loops (set by `LOOP_MAX_INTERRUPTS` in the firmware). This allows for the collection of about 120 Million triggers with a standard CMS Pixel TBM08c module in one loop.

After successfully finishing the full loop, the test loop functions return `true`, the host PC software reads the remaining data from the RAM, and merges the data from all interrupts to have the full test data available.

The pxarCore library automatically takes care of fully executing a loop through all interrupts, collecting the data and returning it as one consistent block.

3.3.4. DAC Calibration

The full range of 8 bit DACs on the ROC is realized by using multiple transistors in parallel. Turning on or off an additional transistor when incrementing or decrementing the DAC value results in a nonlinear shape of DAC response at these positions in the DAC range.

The response can be corrected and linearized by flipping the DAC values at the boundaries of activating an additional transistor. The correction is automatically performed in the Nios II test loops when scanning over DACs in one or two dimensions. This behavior can be disabled using the flag FLAG_DISABLE_DACCAL (cf. Section 5.3).

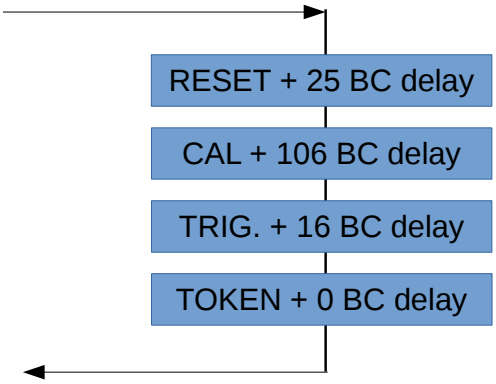


Figure 5: Typical setup of the pattern generator containing a ROC reset signal, a Calibrate pulse, and trigger and readout token. The delay between Calibrate and Trigger has to match the `WBC` setting of the chip plus a chip version-dependent constant offset (5 or 6), and the pattern generator has to be stopped with a delay of 0 BC. One BC is equivalent to one clock cycle of the 40 MHz clock (i.e. 25 ns).

3.4. Pattern Generator

The Pattern Generator is a firmware module that allows to send configurable patterns of signals to the attached devices. It consists of a register bank with 256 addresses, all of which can be programmed with one or more signals to be send. After every signal, a programmable delay is inserted, the pattern generator stops when a delay of zero is set as outlined in Figure 5.

A typical example of a pattern generator setting would be the operation of a detector with calibration pulses. The trigger has to be sent out at a fixed time after the calibrate pulse, defined by the configured trigger latency of the detector (via the `WBC` setting). The pattern generator would then contain both the Calibrate signal with the correct delay, the subsequent trigger signal and possibly also the token if not generated by a TBM.

Multiple pattern generator signals at once can be sent by separating their names with a semicolon. This of course excludes signals which are sent on the same line (e.g. CTR patterns: calibrate, reset, trigger).

Possible Pattern generator signals can be found in Table 8 in the appendix. Since the default trigger source of the DTB is the pattern generator in direct mode bypassing the TBM emulator described below, an initial pattern generator setup should be provided at start-up of the DTB via `pxarCore::initTestboard(...)`, but can be changed any time using the API call `pxarCore::setPatternGenerator(...)` (cf. Section 5.1.1). The total length of a pattern is calculated as the sum of all pattern delays configured. The trigger frequency for the pattern generator has to be larger than the pattern length. This restriction is enforced by the pxarCore library and a warning is printed if the supplied frequency is adjusted.

3.5. TBM Emulator

The TBM emulator is part of the DTB firmware and simulates a full TBM. This can be particularly useful when operating single ROCs without TBM, e.g. in test beams. In this case, the readout token for the ROC has to be generated by the DTB, following every trigger which is sent out. This can be either done using a Pattern Generator, or, when using external triggers, via the TBM emulator.

The TBM emulator also adds header and trailer words to every event, containing useful information like the current trigger number (8 bit) and the phase of the trigger arrival time relative to the 40 MHz clock. This information is stored in the decoded events and can be retrieved directly from the objects (cf. Section 4.8).

3.6. Trigger Sources

The DTB accepts triggers from a range of sources, a full list is given in Table 6 in the appendix. External triggers are accepted via the LEMO port of the DTB as TTL signal. Other sources include the Pattern Generator and an internal trigger generator for random and periodic triggers. The trigger source to be used can be changed using `pxarCore::daqTriggerSource(...)`, multiple trigger sources can be activated by providing all names separated by a semicolon (e.g. `extern;pg_dir`).

Triggers can be configured to either go to the TBM emulator or bypass it and go directly to the attached devices. When operating devices featuring a physical TBM, a direct trigger source has to be chosen.

A special case are *Single Events* which allow to send single signals out without changing e.g. the pattern generator setup. Using pxarCore, these signal can be injected using the API call `pxarCore::daqSingleSignal(...)`.

4. Software Architecture

pxarCore is a library aiming at simplifying the usage and communication with the DTB in order to qualify, calibrate, and operate PSI46-type devices. It provides a set of easy-to-use functions for high-level application programming, e.g. for designing a DAQ software for test beam measurements or for lab calibrations. Instead of interfacing the DTB firmware directly, the pxarCore library should be used for any high-level applications. It includes many safety and consistency checks and tries to abstract the internal DTB structure and functionalities and allows transparent access to the attached devices.

Some functionality of the DTB requires in-depth knowledge about the underlying technologies used (FPGA firmware, USB communication, pattern generators, TBM emulator, trigger routing, correct procedure and order of commands for detector setup, etc.). pxarCore tries to implemented all of this in the "correct" way in order to provide a reference implementation for other software.

pxarCore is a shared library compiled using CMake and compatible with most common operating systems. All data is exchanged via STL containers and pxar C++ class objects. pxarCore is well-maintained (and maintainable), documented, and checked regularly for memory leaks. It compiles with a minimum set of external dependencies (libusb-1.0, libftdi/ftd2xx) under Linux (SL5/6, recent desktop distributions), Windows, and Mac OS X. The API is considered stable, only extensions for new functionalities are added but no backward incompatible changes applied. Even major changes to the internal structure of pxarCore do not (and did not) affect the API. External code can rely on the functions provided by the API, there is no need to rework user code after an upgrade to a later API version.

This section is organized according to the structure of the library outlined in Figure 6. First, Section 4.1 describes the communication of the library with the DTB. Section 4.2 introduces the hardware abstraction layer (HAL), while Section 4.3 provides details on the process of data retrieval and decoding. Section 4.4 describes the public C++ API and Section 4.5 introduces the Python interface. The DTB emulator code is presented in Section 4.6, and Sections 4.7 and 4.8 provide technical details on the implementation of exceptions and data types, respectively.

4.1. Communication with the DTB

The DTB possesses two interfaces for communicating with host PCs. The Gigabit Ethernet port is currently not supported by the firmware, but efforts are ongoing to implement the functionality. pxarCore already provides an Ethernet interface for DTB communication which has to be switched on at build time (cf. Section 2.3). The default interface for the operation of the DTB is the USB 2.0 port. For Linux, two libraries are available to establish the connection with the FTD232 chip on the DTB, the proprietary `libFTD2XX` library provided by the vendor, or the open source alternative `libfdti`. Currently the proprietary library is preferred over the open alternative due to a better performance (cf. Section 2.1).

On the protocol level, all communication is handled by a remote procedure call (RPC)

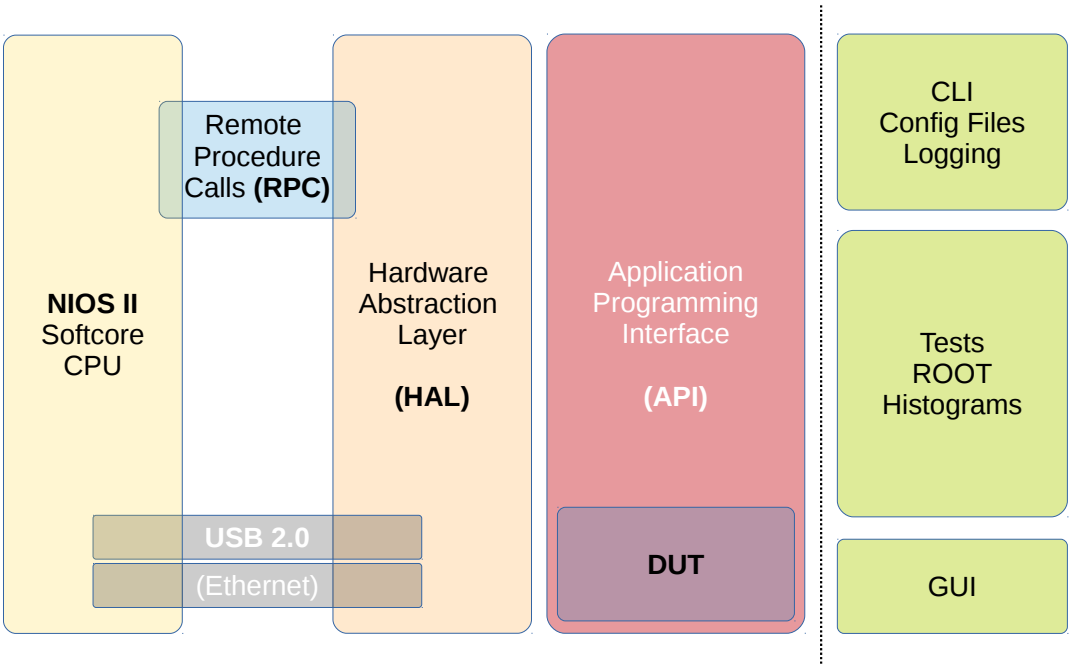


Figure 6: Architecture of the pxarCore library. The DTB is connected via USB or Ethernet, with RPC being the communication layer. The HAL abstracts RPC calls to functions used by the API layer. The API handles all configuration, user input checks, and data packaging. It is the interface to all user code such as a GUI, CLI, or tests.

interface. Both communication partners implement a set of routines which can be called by an identifier and a set of parameters. There are tools available to generate updated versions of the interface files for the DTB Nios II Softcore code and the pxarCore RPC code [14]. The `generate.sh` tool calls the RPC generator which parses the interface header file (`pixel_dtb.h`) of the DTB for the `RPC_EXPORT` preprocessor directive and generates interface implementations for both the DTB and pxarCore. The pxarCore interface in `core/rpc` has then to be updated with the generated file and the header in the same directory adapted accordingly. This step should only be necessary after adding new functions to the Nios II Softcore interface. The firmware's software revision number should be updated if the RPC interface is changed.

4.2. Hardware Abstraction Layer

The hardware abstraction layer aims to provide a transparent layer to the underlying hardware. While the API level of pxarCore abstracts most data handling and configuration for the user, the HAL is responsible for bundling all calls to the DTB which are necessary to perform a certain action, thus the HAL is the only class with direct access to the RPC call stack.

In principle, another implementation of the HAL library could be written to allow operating different readout hardware without the need to rewrite the pxarCore class interface or the configuration and data handling.

Beside combining RPC calls to sequences such as writing parameters, flushing the RPC interface cache, and retrieving information, the HAL also takes care of the data acquisition process by setting up and supervising the decoding chains described in the subsequent sections.

A key element to this is the `pxar::daqStart()` function which sets up the DTB for data acquisition, allocates the DTB RAM and initializes all decoding chains with the correct parameters received from the API class. The HAL is also responsible for the event building. In case of a DAQ with multiple channels (such as a full CMS Pixel Detector module), the HAL functions take care of properly combining the channels into single events by concatenating the content.

Since the test loops described in Section 3.3.2 produce a huge amount of data (sometimes several GB of raw data for tens of millions of triggers) the HAL implements the `condenseTriggers()` function which already combines all events belonging to the same DAC setting and pixel (usually configured in the API test call as `nTriggers`) in order to reduce the amount of data kept in the PC RAM. It either sums the number of responses of a given pixel (efficiency mode) or calculates mean and variance of the pulse heights returned.

4.3. Data Readout & Decoding

The data decoding is performed using separate classes for every task in the decoding chain, which are connected via output operators (`>>`) as shown in Figure 7. First, the raw data is retrieved from the DTB by so-called data sources, and the individual events

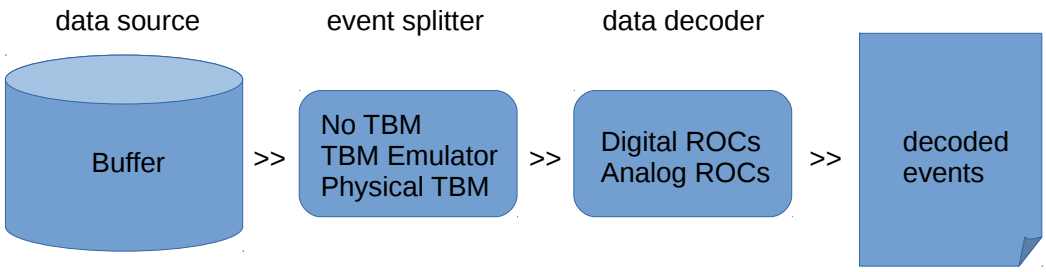


Figure 7: Structure of a single pxarCore data pipe showing the necessary modules. The data is retrieved from a data source and passed through the event splitter and event decoder. The data sink outputs decoded events.

are recovered by the event splitters. Finally, the actual event content is decoded. The classes are used for the pxarCore-internal data decoding of all tests, but can as well be used by external programs as demonstrated in the `decoder` tool provided in the repository (cf. Section 6.3).

4.3.1. Data Sources

The first step in the decoding process is the retrieval of raw data by so-called data sources. Every data source is responsible for one DAQ channel, for TBM operation multiple data sources are instantiated. The following data sources are available:

dtbSource : This is the standard data source for all data which is directly retrieved from the DTB. It accesses the RPC interface and requests data blockwise. The size of the data blocks is defined as `DTB_SOURCE_BLOCK_SIZE` and is set to 8192 words by default. The transmission speed or readout performance does not depend on this value. When the buffer is empty the dtbSource throws an `dsBufferEmpty()` exception which is caught by the calling functions in the `pxar::HAL`. The dtbSource belongs to the HAL library.

evtSource : This is a data source which allows to inject already recorded raw data into the decoding modules. It is e.g. used to perform offline decoding of test beam data which has been retrieved online as raw data and immediately stored without decoding. This class is part of the pxarCore decoder.

All data sources have to be initialized with a set of parameters describing the data to be expected. The data source itself will not make much use of this information, but the subsequent processors retrieve the parameters via the output operators. The following parameters are required:

src: (dtbSource only) the RPC class handle for the DTB connection.

daqchannel: the DAQ channel to read from. This parameter is not strictly necessary for the evtSource but allows to distinguish several decoding instances running in parallel in the log output.

tokenChainLength: the number of ROCs per event expected in this particular data channel. This parameter is internally used to accommodate for modules which have bypassed ROCs as well as for multiple ROCs daisy-chained to one DESER160 input (beam telescopes).

offset: an additional offset for numbering the ROCs found in the data stream. If channel 0 contains 8 ROCs, the `offset` parameter for the channel 1 data source should be set to 8 in order to start the ROC numeration at 8.

tbmtype: type of the TBM used. This parameter expects a correct device identifier from the Device Dictionary (see Section 5.7).

roctype: type of the ROCs used. This is of special importance since some ROC types require special treatment of their data, such as inverting the pixel ID for an early prototype chip, or the linear address space of the layer 1 ROC. This parameter expects a correct device identifier from the Device Dictionary (see Section 5.7).

endlessStream: this setting prevents the data source from throwing the `dsBufferEmpty` exception. The source will continue polling for new data until it can return the next block.

daqflags: this parameter allows to specify additional flags which can influence the decoding process (cf. Section 5.3).

While the dtbSource will fetch the data automatically via the supplied DTB handle, the evtSource buffer needs to be filled explicitly via the `AddData()` member functions which accept either a single 16bit word or a vector of such words.

Data sources can be instantiated with their default constructor, not supplying any configuration parameters. The internal state of the source is then initialized to *not connected* and it will not respond to inquiries until it is properly configured and connected. This allows to allocate member objects but only configure them at run time with the required parameters.

4.3.2. Event Splitters

The subsequent step in the decoding chain is the splitting of events. The data stream contains special marker bits (depending on the configuration) which indicate the start and end of single events. The event splitter routines search for these markers and keep on requesting more data from the data sources until the next event marker is found.

This data is packed in a `pxar::rawEvent` (see Section 4.8), and additional information such as flags for missing event start or end markers are stored.

The default class for event splitting is the `dtbEventSplitter` which provides subroutines for splitting of data streams recorded without a TBM, with the TBM emulator,

and with a physical TBM. For the latter, in addition to packing the data, the channel ID is stored in unused bits of the TBM header word for later reference.

In addition to this flexible event splitter, the `passthroughSplitter` class can be used to just pass through all data provided without checking for event markers. Data is retrieved from the source until the `dsBufferEmpty` exception is thrown. This can be useful if the data is already separated into single events. The implementation of the pxarCore decoder in EUDAQ described in Section 8 makes use of this splitter routine, since all events are already separated at data acquisition time using the `pxarCore::daqGetRawEvent(...)` function.

If only raw event data is requested, this is the endpoint for the decoding chain where the object is retrieved by the data sink.

4.3.3. Event Decoders

The central part of the data pipe is the actual decoding accomplished by the dtbEventDecoder class. It uses the configuration parameters supplied via the data source to select the correct decoder module. The event decoder is also the module collecting all statistics which can be fetched via `pxarCore::getStatistics()`, as described in Section 5.10.

First, a new `pxar::Event` object is instantiated, the next `pxar::rawEvent` is requested from the splitter routines, and the statistics are updated with the flags from event splitting. Then the presence of any TBM in the data is checked via the configured TBM type.

If a TBM is present, its header and trailer are analyzed. The data is stored in the header and trailer members of the `pxar::Event` from where all status information can be retrieved via member functions (cf. Section 4.8). The TBM event ID is stored and compared to the expected event number in the local counter. The TBM header and trailer are then removed from the data, and the `pxar::rawEvent` is passed on to the ROC data decoders.

The dtbEventDecoder class contains modules for every type of PSI46 chip, both digital and analog. The decoder modules iterate over the data and check every word for the appearance of a ROC header identifier. If found, the word is treated as new ROC, otherwise, the data is treated as possible pixel data. The actual translation from the pixel gray code or address levels into column and row addresses is performed in the `pxar::pixel` class itself which provides constructors for the different input data types as described in Section 4.8.

For analog data, the *ultra black* and *black* levels are retrieved from the ROC headers and averaged using a running-average of the last 1000 events. Since the initial values set by the class constructor might not suit the particular setup, it is of special importance that the first word of analog event data contains the first ROC header, otherwise wrong levels will be used for the average, and the address levels cannot be calculated. Also see Section 5.11 for further information on analog ROCs.

At the end of the decoding process, the final event undergoes a validity check. This includes checks for the total number of ROC headers, and collects the statistics about empty events. Additional information such as a possible `NoTokenPass` or `PKAMReset` bit

in the TBM trailer is used to correctly react on special situations. With this bit set, no token has been sent by the TBM and hence no ROC headers are expected in the data stream.

In addition to the actual data decoding, the dtbEventDecoder also collects the readback information and makes it available via the `pxarCore::daqGetReadback()` function as described in Section 5.9. For data recorded by the DESER400 modules described in Section 3.2, the XOR sum indicated in Figure 4 is collected from the ROC headers if the FLAG_ENABLE_XORSUM_LOGGING flag is set. The data can be fetched from the pxarCore library via the `pxarCore::daqGetXORsum(uint8_t channel)` API call.

If the FLAG_DUMP_FLAWED_EVENTS flag is passed via the data source, the dtbEventDecoder will keep the last seven events in a ring buffer and prints them in case of an event exhibiting problems. This allows to inspect the broken event itself as well as the preceding and successive events. As described in Section 5.3, this should only be used if necessary since the decoding process is slowed down significantly. The printout is restricted to the first 100 erroneous events.

4.3.4. The Data Sink

In order to retrieve data from the decoding pipe, a data sink has to be used. The dataSink class provides an interface to all possible modules in the decoding chain and returns objects of the selected type. The return data type has to match the output type of the module to be connected, e.g.

```
dataSink<Event*> pump;
```

The `pxar::rawEvent` and `pxar::Event` data types are returned as pointers, and the actual objects have to be copied by the requesting code. Data obtained directly from the data source is returned word by word as fixed-size `uint16_t` integers.

4.3.5. Building the Decoding Chain

The full decoding chain is built from the classes described above by creating instances and connecting them with the output operator `>>`. The decoding chain always has to start with a data source and end with a data sink. The pxarCore library automatically sets up all necessary decoding chains depending on the configuration of the DUT (see Section 4.4.1).

An example of how to interface the decoder modules from user code is provided in the example code `decoder` described in Section 6.3.

Decoding chains are very flexible and the output can be changed at run time. It would for example be possible to hook up the same decoding chain to two different data sinks, and then request every odd event as decoded `pxar::Event` and all others as `pxar::rawEvent`.

4.4. Application Programming Interface

The application programming interface is the central interface through which all calls from tests and user space functions are routed in order to interact with the hardware.

The API provides a set of high-level functions from which the "user" (or test implementation) can choose without needing deep knowledge about the readout electronics and device programming procedures. This approach allows to hide hardware-specific functions and calls from the user space code and automatize e.g. start-up or data acquisition procedures. All input from user space is checked before programming it to the devices to minimize the probability of a misconfiguration and thus corrupt test data. This includes DAC names (valid register?), DAC values (within range of this register?), pattern generator validity (having a zero delay to end the pattern), trigger frequency, DTB power limits, and many more. Register addresses have an internal look-up mechanism so the user only has to provide e.g. the DAC name to be programmed as human-readable string (see Section 5.7).

Unless otherwise specified[1] all data returned by API functions is fully decoded and stored in C++ structures using standard containers such as vectors to ease further handling and plotting. Most functions return a vector containing `pxar::pixel` objects storing the readout data. All data is pre-processed and already reduced to average values, e.g. the result of a test requesting 100 triggers to every pixel will just be one value per pixel, averaged over all 100 measurements. A more detailed description of this processing is given in Section 4.4.4.

All detector parameters are stored in the API member `pxarCore::_dut()` and can be retrieved or altered via API calls as described in Section 4.4.1.

Calls to test functions are automatically expanded in a way that they cover the full device in the most efficient way available. Instead of scanning 4160 pixels one after another the code will select the function to scan a full ROC in one go automatically. This expansion procedure is described in detail in Section 4.4.3.

Every API function available is well-documented and states the required input parameters, the return values as well as possible exceptions which might be thrown during execution. The function documentation can be either retrieved from the auto-generated Doxygen documentation [15], the github repository code viewer [16] or from the source code directly (in `core/api/api.h`).

4.4.1. Device Under Test

The pxarCore library keeps track of all settings sent to the detector by storing them in the so-called device under test (DUT) class `pxar::dut`. The DUT class represents the hardware attached to the DTB. In idle state, the detector is kept in a quiet configuration (i.e. all pixels masked, double columns detached from the readout) and only for tests or data acquisition the settings are read from the DUT class object and programmed into the detector.

In order to change the detector configuration, the user space code interacts with the `pxarCore::_dut()` object and alters its settings. All settings contained in the DUT are programmed into the devices automatically before the next test is executed. This approach allows both the efficient execution of many RPC calls at once and the retrieval

[1] some DAQ functions allow this, see Section 4.4.5

of actual device configuration at any time during the lifetime of the pxarCore object. From the DUT settings also further internal configurations such as the correct deserializer module (cf. Section 3.2) or the required number of DAQ channels are derived.

The DUT has to be initialized once at start-up via the `pxarCore::initDUT(...)` API function and can be altered any time. The initialization function accepts a set of vectors containing all important parameters for the detector. For every TBM Core, a set of register settings in the form of a `std::pair(<register name>,<register value>)` as well as the type of the TBM used has to be provided. Leaving the TBM register vector empty will configure a detector without physical TBM.

For every ROC in the setup, a set of parameters have to be provided via `std::pair(<DAC name>,<DAC value>)` entries. In addition, every ROC needs 4160 pixel configurations which contain initial settings for the mask bit, the trim bits and the `enable` state which adds it to the test range. The loop expansion of the test functions reads these `enable` settings from all pixels and adjusts the test range accordingly (cf. Section 4.4.3).

All name parameters are accepted as human-readable strings and are interpreted using the pxarCore dictionaries (see Section 5.7). All attached devices must have the same revision, i.e. it is impossible to configure modules with a mixed set of psi46digv2 and psi46digv2.1 ROCs.

4.4.2. Test Functions

In order to ease tests of detector modules and ROCs, a set of test functions has been implemented into the pxarCore library. The main goal of these functions is to offload functionality to the DTB Nios II Softcore CPU for faster execution. This is especially important for tests where pixels constantly have to be masked or trimmed, and DAC register values are changed regularly. Implementing such an algorithm on the PC host side would entail a large overhead of USB traffic with latencies of the order of milliseconds per call.

The test functions implemented in the Nios II Softcore CPU provide the possibility of looping over a range of parameters (being it pixels or DAC values) in the most efficient way, recording the data and then downloading the full raw data to the host PC. All decoding and other involved processing, such as averaging of pulse heights or threshold calculation is done on the PC side due to the lack of computing power on the FPGA. The functions should be able to carry out almost all desired tests and are additionally configurable using a set of combinable flags which are described in Section 5.3.

There are three types of return values. The test can either return an averaged pulse height for every pixel, an efficiency value as number of responses per pixel, and a threshold value for a certain DAC register with configurable threshold requirement (cf. Section 4.4.2). Figure 8 shows the matrix of implemented test functions for the three output variables. It is possible to just loop over all active pixels ("Map"), to loop over active pixels and one DAC register ("1D DAC") or over active pixels and two DAC registers respectively ("2D DAC"). A 2D DAC with additional threshold DAC is not implemented since the data volume and test execution time would exceed reasonable limits and is unlikely to be needed at all.

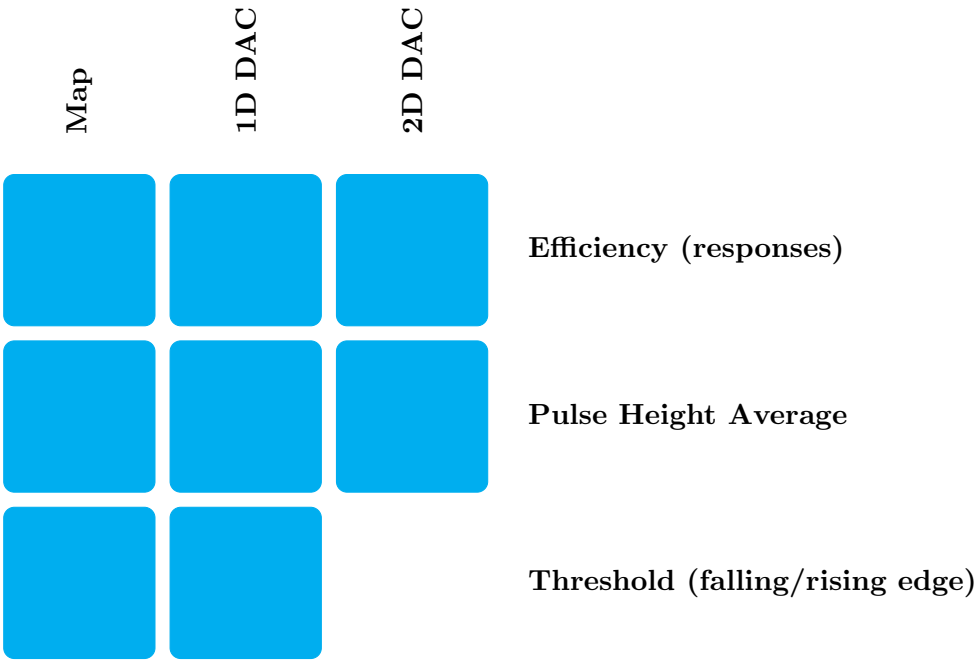


Figure 8: Test matrix of implemented test functions in the pxarCore library. Rows represent the different data output formats (just number of responses, the averaged pulse heights, or a threshold value) while the columns are the different scopes of the test (just calibrating, calibrating for every DAC value in a certain range, calibrating while scanning two DACs)

There are a few things which need to be considered when running a test function. The following paragraphs will provide more details on the functionality and configuration possibilities.

Execution Time The execution time for a certain test depends on several parameters. First of all, all tests scale linearly with the number of triggers requested per combination. Thus limiting the test to the necessary number of triggers for the precision required will optimize the overall execution time. The timing also depends on whether the tests are executed in serial or in parallel. Here, serial denotes testing every ROC in the DUT sequentially, while parallel refers to running a test on all ROCs of the DUT (e.g. a full module) at the same time. Measuring all ROCs in parallel is faster by a factor six or more compared to serial execution. The reason for this is twofold. On the one hand, the number of triggers (and thus loop iterations) is reduced by a factor `nROCs` if one trigger generates data from every ROC in the chain. On the other hand, events generating data from only one ROC still contain the headers from all other ROCs and thus a lot of unnecessary data overhead which needs to be read via USB. Section 5.3 provides further information on how to switch on or off serial execution of tests.

The overall test time also depends on the setting of the WBC register since the Nios II loop needs to wait for this trigger latency after a calibrate pulse before a trigger signal can be issued. Thus setting WBC to a very high value like 255 requires a waiting time of $255 \times 25\,\text{ns} \approx 6.3\,\text{us}$ before every trigger in the loop. It has to be noted that setting the WBC too low (< 20) is not recommended and might lead to unexpected behavior of the ROC. The delay between calibrate and trigger signal is configured via the pattern generator and should be set up automatically by the application used. The pxarCore library itself does not explicitly check for a correct setup of the pattern generator and only provides cross-checks for the total pattern length and a valid termination (cf. Section 4.4).

Finally, also reading data from the detector takes time. With the digital ROC interface at 160 MHz the ROC header requires three 40 MHz clock cycles to be transmitted, every pixel hit needs another six clock cycles. When running with a TBM, its header and trailer add another seven clock cycles each [17]. For single ROC operation, the total event readout time is usually covered by the length of the pattern generator (see above) but for full modules with possibly multiple hits on every ROC this is easily exceeded. If the next token for readout (or even a reset signal to ROCs or the TBM) is already sent out before the event readout has finished the event gets chopped off and is considered incomplete. The TBM internal trigger stack does not help here since the tests are triggering with a fixed frequency and thus the stack goes into overflow after 32 consecutive events with long readout. For this case, pxarCore tries to estimate the readout time per event required for the current test and inserts additional delays between the triggers.

If, for some reason, this automatically introduced delay is not enough and the test results in many decoding errors with corrupt data (e.g. event end markers missing) it might be necessary to manually increase the delay. This can be done by supplying the `triggerdelay` setting to the DTB as described in Section 5.3. This will add another delay in units of 10 clock cycles for every trigger. The time between triggers scales

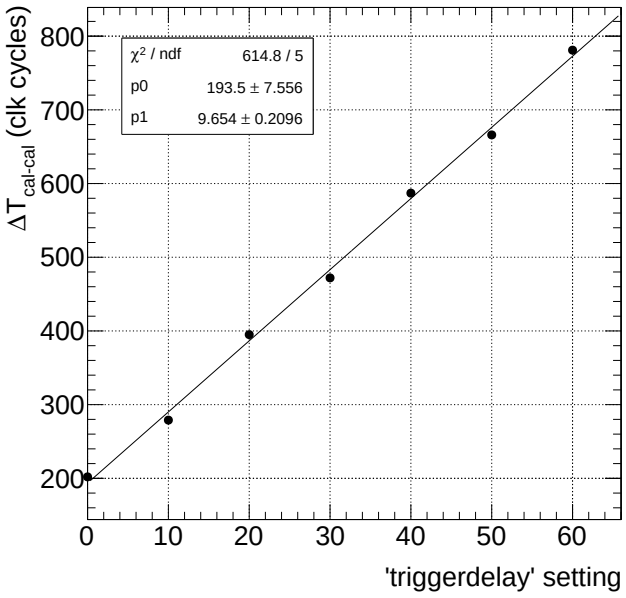


Figure 9: Effect of the `triggerdelay` setting on the time between two calibrate signals sent to the detector, measured with an oscilloscope. The behavior is linear with an offset corresponding to the automatically calculated delay derived from the pattern generator length. Figure by [18].

linearly with the delay introduced as can be inferred from Figure 9.

Threshold Measurements Some applications in the detector qualification require figuring out the setting of a certain DAC register at which the pixels respond to an external stimulus. This is usually referred to as "threshold" for this DAC register. To ease the measurement of thresholds, an automated threshold scan is implemented in the pxarCore test functions. The pxarCore function executes a DAC scan over the specified range of the selected DAC parameter and retrieves the full data block from the DTB. For every pixel, this DAC is then scanned for the number of responses, and the DAC value at which the response threshold level is reached (default is 50% of the transmitted triggers, other levels can be configured) is returned as threshold value. The analysis of the recorded data is executed in the Data Repacking Routines of the pxarCore library as described in Section 4.4.4.

Since some DACs have a rising edge efficiency behavior while others feature a falling edge when scanning the DAC range from low to high values, the slope of the threshold curve to be measured can be specified using FLAG_RISING_EDGE (cf. Section 5.3). The default assumed is a falling edge behavior.

Sparse Scanning Another possibility to speed up tests is reducing the granularity of scans over DAC registers. The pxarCore test functions allow the specification of a step size for all 1D and 2D DAC scans. This effectively reduces the number of triggers to be sent and thus the test execution time.

Application scenarios could e.g. be a fast threshold finding algorithm that first scans the whole DAC range with a coarse granularity (e.g. $\Delta\text{DAC} = 8$) to find the threshold region, and then applies a fine grained scan with $\Delta\text{DAC} = 1$ only in this region of interest of 8-10 DAC units. This region of interest has to be defined by the user based on the output of the sparse scan, and the test has to be repeated with the new limits and stepping configuration.

An exemplary scenario could be the scan of all 4160 pixels of one single ROC for a VCal DAC threshold of 50%. A scan over the full DAC range from 0 to 255 with a step size of 1 DAC unit requires approximately 65 s (depending on other parameters). The threshold could be determined with the same precision by first running a scan with a step size of 8 over the full range (execution time 9 s) and an additional scan with step size 1 within 10 DAC units around the found threshold from the coarse scan (execution time 6 s). The total time for this test would then be just 15 s with a speedup factor of four.

4.4.3. Loop Expansion

In the API function `pxarCore::expandLoop(...)`, the pxarCore library provides a so-called *loop expansion* algorithm which tries to figure out the most efficient way to deliver the data requested. In order to do so, it checks which parts of the DUT are marked for testing (i.e. ROCs and pixels which have the *enable* bit set) and then automatically selects the correct test function to call on the Nios II. The Nios II code has loops available to run either on single pixels or on full ROCs, both one by one (ROC after ROC) or in parallel (requesting one pixel at a time from every ROC in the readout chain) as described in Section 3.3.2.

For example, with a module and the same pixel set for testing on every ROC, the loop expansion will call the function `pxarCore::MultiRocOnePixel<...>(...)` once in order to get all the data requested. If there are five pixels configured for testing on ROC 3 of a module, the expansion will execute five calls to the API function `pxarCore::SingleRocOnePixel<...>(...)` and merge the data before delivery to the calling code.

For user code this has some implications on the suggested usage of the test functions. The most efficient way is always to request all data at once instead of running a set of tests one after another. All interesting parts of the DUT should be marked as *enable* using the `pxarCore::_dut->testPixel(...)` functions before the test is executed. It is also not recommended to set up loops in user code to e.g. loop over all ROCs (which can be done in parallel, see Section 4.4.2) but to run the test on all. It is also not recommended to split the test itself e.g. by doing something like

```
for(uint8_t dac = dac_min; dac <= dac_max; dac++) {
```

```
  api->setDAC(mydac1,dac);
  data = api->getEfficiencyVsDAC(mydac2,...)
}
```

instead of implementing a single call to the two-dimensional DAC efficiency function `pxarCore::getEfficiencyVsDACDAC(...)` since it will invoke a series of tests including all overhead from reprogramming the full DUT and allocating memory space on the DTB.

Other approaches will of course also work but make the testing less efficient due to overhead in USB communication, memory allocation, and more.

4.4.4. Data Repacking Routines

The pxarCore library provides four internal routines for data preprocessing and packaging. The goal is to unify and compress the data read from the detector into a common data format. All repacking functions work in a similar way. The pixels are sorted and checked for their correct appearance order if FLAG_CHECK_ORDER has been set (cf. Section 5.3). If this flag is set, the pulse height is marked with a negative sign in case a pixel is found out-of-order. Usually this means it is a background hit and does not stem from an internal calibration pulse.

The `pxarCore::repackMapData(...)` function takes care of data returned from tests that do not alter any DAC settings and just returns a linear vector of all pixels found in the data. Before returning, the vector of pixels is sorted according to the pixel addresses.

`pxarCore::repackDacScan(...)` processes one dimensional DAC scans and in addition to the previously described routine assigns every event to the correct DAC value used to record this particular event. The returned data are a vector of pairs containing both the corresponding DAC value and all responding pixels with their pulse height average or response efficiency, depending on the test configuration.

The same is done by the `pxarCore::repackDacDacScan(...)` function in two dimensions, and a structure reflecting the two DAC dimensions and their pixel responses is returned.

The more involved functions are the repacking routines for threshold scans. In addition to the previously described functions, the additional DAC dimension is scanned for the threshold level set by the test function (default is 50% of the triggers). The repacking function `pxarCore::repackThresholdMap(...)` works similar as the 1D DAC scan function `pxarCore::repackDacScan(...)` but treats the additional dimension differently. The full DAC range is checked for the current efficiency and the slope of the curve to figure out the exact threshold point. This is done separately for every pixel found in the data.

The spectrum of the DAC under investigation is scanned for every pixel, and the absolute efficiency value as well as the slope is calculated at each point. By also comparing the slope, fluctuations in the spectrum can be excluded from the threshold search by requiring a specific slope sign: Either positive for rising edge searches or negative for falling threshold edges. If the absolute value is above the configured threshold level and

the slope exhibits the correct sign, the value is taken as the threshold DAC value for the respective pixel.

Finally, `pxarCore::repackThresholdDacScan(...)` employs the same algorithm as the previous function but allows for an additional DAC dimension to be scanned and being returned.

All functions described here also handle data from sparse DAC scans.

4.4.5. Data Acquisition Functions

A set of dedicated functions for data acquisition (DAQ) is provided by the pxarCore library. These functions are meant for data taking in test beams, X-Ray machines or with cosmic rays, i.e. for all purposes that do not need involved algorithms changing chip parameters like the test functions described in Section 4.4.2.

The DAQ functions prepare all low-level settings for the data acquisition and take care of e.g. allocating RAM on the DTB, activating the appropriate number of DAQ channels (single ROC, full module, dual-channel TBM...), and setting up and attaching the correct deserializer module.

Other settings are left for the user to configure and choose, such as the trigger mode (cf. Table 6), trigger patterns, frequency, or number of triggers. The following functions are available:

daqStart() Sets up and initializes a new data acquisition session.

daqStop() Stops the running data acquisition.

daqStatus() Retrieves the status of the current DAQ session. For a running DAQ with free buffer memory left, this function returns TRUE. In case of a problem with the DAQ (not started, buffer overflow or full) it returns FALSE. This function is supposed to be used for a continuous check e.g. in a DAQ thread.

daqStatus(level) As above, but in addition provides current the buffer fill level in percent in the pass-by-reference variable `level`.

daqTriggerSource(source,rate) Selects the trigger source to be used, some sources allow the specification of a trigger rate. Table 6 lists all possible parameters.

daqTrigger(nTriggers,period) Fires the previously defined pattern command list "nTriggers" times, the function parameter defaults to 1. The function returns the triggering period actually used after cross-check with the pattern generator cycle length. This function will only return after all triggers have been sent!

daqTriggerLoop(period) Fires the previously defined pattern command list continuously every "period" clock cycles (default: 1000). The function returns the triggering period actually used after cross-check with the pattern generator cycle length.

daqTriggerLoopHalt Halt the pattern generator loop which has been started using `pxarCore::daqTriggerLoop(...)`.

The trigger frequency will be limited by pxarCore to the cycle time of the pattern generator if used. However, if the programmed pattern starts with a ROC reset command all data might be deleted before the readout from the previous trigger is finished. This would lead to chopped-off event data and thus to invalid data. In case the data acquisition has to be interrupted it is not necessary to call `pxarCore::daqStop()` and `pxarCore::daqStart()` every time. It is more convenient to just stop the triggers, read out data and resume the triggers without stopping and restarting the data acquisition itself. The start-up process is comparatively time consuming since the detector is re-initialized and RAM newly allocated every time. It is recommended to check the status of the DAQ and the RAM filling level regularly using `pxarCore::daqStatus()`.

There are different ways of retrieving the recorded data. By default, all data are passed through the decoder modules of pxarCore yielding fully decoded detector events. In some cases this might not be desired, and other functions can be used to intercept the data earlier in the data pipes:

"memory dump" : reads the data from the DTB RAM as it has been recorded and returns it as a vector, `pxarCore::daqGetBuffer()`. If multiple DAQ channels are opened (TBM operation) all channels are read and simply appended. It should be noted that this might lead to event mixing as all DAQ channels are merged.

Raw detector data : data are read from the DTB, and split into single events but not decoded (still raw 16bit integers). If multiple channels are active, data from all channels are merged. The returned data type is a vector of `pxar::rawEvent` objects. Either a single event can be read using `pxarCore::daqGetRawEvent()` or all events left in memory can be returned using `pxarCore::daqGetRawEventBuffer()`. This function throws a `pxar::DataNoEvent` exception if no event is found in memory.

Decoded data: data are read, split into single events and fully decoded into `pxar::pixel` objects containing the hit information. The return data format is `pxar::Event` which holds the pixel information as well as additional information read from TBM header and trailer. It is possible to either read single events with the function `pxarCore::daqGetEvent()` or all events in memory using the API call `pxarCore::daqGetEventBuffer()`. This function throws a `pxar::DataNoEvent` exception if no event is found in memory.

In case multiple DAQ channels are active, the output of all channels will be combined for every event. Thus every event retrieved from pxarCore (either as raw data or decoded) will already contain all information available for a certain trigger. If a mismatch between the different channels occurs, a `pxar::DataChannelMismatch` exception is thrown. This is a fatal exception and should lead to a restart of the current data acquisition.

If no events are available in the DTB buffer at the time of the request, a non-fatal `pxar::DataNoEvent` exception is thrown indicating that no data was available. It has to be noted that this is very different from returning an empty event, since the latter

one would imply a correct trigger and token pass through the detector with no pixels firing. This is of special importance in cases where an external trigger is used.

All data type classes defined within the `pxar` name space are described in Section 4.8.

4.5. Python Bindings

In addition to the C++ API, the pxarCore library provides a Python interface realized using Cython. This allows writing of simple scripts carrying out complex tests in Python. The Cython interface code is manually written instead of automatically generated. This has proven to be more robust and efficient and is not a major effort to maintain.

All interface code can be found in the `core/cython` directory, and the compilation of the interface has to be enabled explicitly via

```
cmake -DBUILD_python=ON ..
```

and requires Python, Numpy and Cython as additional dependencies (cf. Section 2.1).

The Python interface is currently used by e.g. the pxarCore Python CLI (Section 6.1) or pyXar (Section 6.2). At the moment not all C++ API functions are available in the Python interface since they were not required yet, but implementing new functions is straightforward and not involved since all the infrastructure and data containers are present already.

4.6. DTB Emulator

To allow detector-independent unit tests of the pxarCore library and the pxar GUI software suite a DTB emulator class is available, which emulates a DTB on RPC-level, responding to the different requests made by the software. This can be useful for software development if no DTB is connected or available, as well as for unit and system tests validating software changes. Furthermore, the generator can be used to provoke certain error conditions and evaluate the response of the decoding algorithms.

The emulator produces pseudo data in the original detector raw format and passes them through the normal pxarCore decoding chain. This approach includes almost all software components and allows for the most complete system test possible without actual hardware involved.

All tests from the GUI or any other application can be executed, and detector hit maps, DAC scans, and other requests yield (more or less) sensible data which can be analyzed as demonstrated in Figure 10. More involved algorithms such as trimming a device will obviously not work as expected since no full chip simulation is implemented. However, some advanced features are taken care of, such as the `NoTokenPass` register of the TBM cores. These register values are picked up by the emulator and no ROC data is returned for the respective channels. Also plain data acquisition is implemented, where every call to `pxarCore::daqGetEvent()` will yield an event with some noise pixel hits.

In order to enable the DTB emulator at build time, CMake has to be called with

```
cmake -DBUILD_dtbemulator=ON ..
```

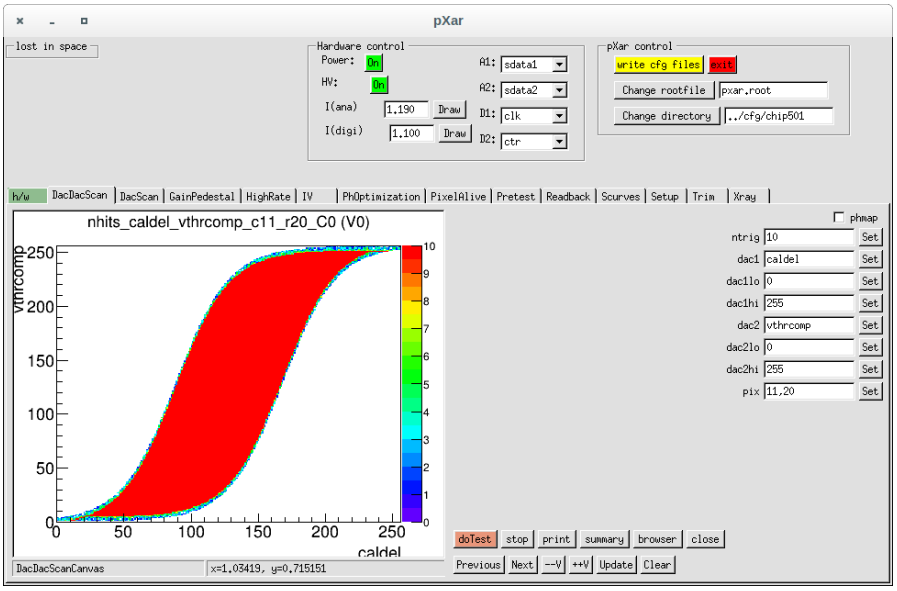


Figure 10: Analysis of the data returned by the DTB emulator for a 2D DAC-DAC scan. The shape of the tornado stays the same irrespective of the DACs chosen to be scanned.

as described in Section 2.3. Turning off the compilation of all interfaces available also results in the DTB emulator being built.

All code of the DTB emulator can be found in the directory `core/emulator`. Technically this directory contains an alternative RPC interface class which is compiled and linked by CMake instead of the one implementing USB and Ethernet connectivity (which can be found in the `core/rpc` directory). The CMake code for switching between the directories can be found in `core/CMakeLists.txt`.

4.7. Exceptions

STL exceptions are used to propagate information about problems, timeouts and decoding issues to higher levels of the software package. User code such as test routines or DAQ programs should be written such that exceptions are caught and handled correctly. All exceptions thrown by pxarCore inherit from the `pxar::pxarException` class, which itself is derived from the `std::exception` class of the STL.

Currently the following exceptions are implemented and used:

InvalidConfig : An invalid configuration exception is thrown if configuration parameters supplied to pxarCore do not match or are otherwise invalid. This only covers severe problems such as

- missing (crucial) parameters,
- inconsistent or mismatched configuration sets.

In case of minor problems such as range overflow of a DAC parameter, no exception is thrown but the DAC value is set to its maximum and a WARNING message is issued (cf. Section 5.8).

FirmwareVersionMismatch : This exception class covers issues with DTB firmware version mismatches between the pxarCore RPC and the Nios II Softcore interfaces and requires the flashing of an alternative firmware to the DTB.

UsbConnectionError : This exception class covers read/write issues during the USB communication or problems opening the connection to the specified test board. In most cases this is either a hardware problem with the USB connection such as a broken cable, or a problem with the operating system kernel.

UsbConnectionTimeout : This exception class is used to notify about communication timeouts occurring during USB calls to the DTB.

DataException : This exception class is the base class for all pxar data exceptions. All exceptions thrown due to problems with the handling and decoding of detector data inherit from this exception.

DataNoEvent : This exception class is used in case a new event is requested but no data are available. Usually this is not critical and should be caught by the caller. When running a DAQ with external triggering and constant event polling from the DTB it can not be ensured that data is always available, but returning an empty event is not an option as it would mess up the trigger synchronization and lead to event mixing. Throwing the exception circumvents this problem.

DataChannelMismatch : This exception class is used whenever multiple DAQ channels are active and there is a mismatch in event number across the channels (i.e. channel 0 still returns one event but channel 1 is already drained).

DataMissingEvent : This exception class is used when the DAQ readout is incomplete, e.g. one thousand triggers have been sent, but only 999 events could be retrieved (missing events).

DataDecodingError : This exception class is used when a problem in data decoding occurred, such as undecodable pixel addresses. This exception is internally caught by the decoder classes and used for accumulating statistics as described in Section 5.10. The `DataDecodingError` class contains a set of child classes to further specify the type of decoding error which has been encountered. The full list including short descriptions of the content and use cases can be obtained from the Doxygen documentation [15] or the source code in `core/api/exceptions.h`.

4.8. Data Types

The pxarCore library implements a set of data types for simplification of data handling, which are described in the following.

`pxar::pixel` is a class for storage of pixel hit information. It contains a set of constructors which decode Gray code pixel addresses into column and row information for both digital and analog ROC data:

pixel(): default constructor, initializing all members to zero.

pixel(roc_id,column,row,value): constructor for column/row coordinates. Parameters are just stored and not immediately sent to the DUT.

pixel(rawdata,rocid,invertedAddress): constructor for raw data from digital PSI46 devices. The `invertedAddress` flag notifies about the inverted pixel id present in the PSI46digV1 version of the chip. The data is translated from Gray code to column and row and stored. If the decoding fails, a `DataDecodingError` exception or one of its child exceptions is thrown.

pixel(analogdata,rocid,ultrablack,black): constructor to decode analog level-encoded data from PSI46V2 devices. The *ultrablack* and *black* levels are used to calculate the five address levels for column and row decoding. If the decoding fails, a `DataInvalidAddressError` exception is thrown.

The following member functions allow to retrieve information from the `pxar::pixel` object:

roc() ID of the ROC the pixel hit was recorded on

column() Column address of the pixel

row() Row address of the pixel

value() Pulse height or averaged response efficiency for the pixel

variance() Variance of the averaged pulse height, if applicable.

Internally the mean value and variance are stored as compressed floating point values in order to minimize the memory footprint of the object. Thus the member variables can't be altered directly but only through set-functions.

In addition the comparison operators `==` and `<` are overloaded for this class to be able to easily compare pixel hits against each other. In both cases the ROC ID as well as the pixel column and row are compared. Also the `ostream` operator `<<` is overloaded to ease printing of pixel information to `stdout`.

The `encode()` member function allows to retrieve encoded raw data from an existing pixel hit. This is mainly used by the DTB emulator to produce pseudo data hits (cf. Section 4.6).

`pxar::rawEvent` holds undecoded event data. The member `data` holds 16 bit unsigned integer values containing the event data. In addition, a `flags` field keeps track of decoding problems when isolating this particular event, such as misplaced or missing event start and end markers.

To allow adding data from different data streams into one raw event, the augmented assignment operator `+=` is overloaded for this class. Also the `ostream` operator `<<` is overloaded to allow simple printing of raw event information to `stdout`.

`pxar::Event` contains fully decoded event data. All pixel hit information is stored as `pxar::pixel` objects in the `pixels` member of the class. In addition, TBM header and trailer are stored and can be read either directly via the members `header` and `trailer` or by means of the member functions which allow direct access to information contained in the header and trailer data:

triggerCount() TBM Header Information: returns the internal 8 bit trigger/event counter of the respective TBM core. This information is compared to an internal event counter of the pxarCore library.

triggerPhase() TBM Emulator Header: returns the phase of the trigger relative to the clock. This information is stored by the TBM emulator (cf. Section 3.5) and is equivalent to the value returned by `dataValue()`.

dataID() TBM Header Information: returns the Data ID bits for each TBM core indicated in Figure 3. The ID specifies the content of the data value bits.

dataValue() TBM Header Information: returns the value for the data bits. The contents has to be interpreted according to the data ID bit setting, for the TBM emulator the bits contain the trigger phase described above.

hasNoTokenPass() TBM Trailer Information: reports if no token has been sent to the ROCs and the event content should be discarded. This bit is also set in case of a PKAM reset (see below).

hasResetTBM() TBM Trailer Information: reports if a TBM reset has been sent in the corresponding CTR pattern.

hasResetROC() TBM Trailer Information: reports if a ROC reset has been sent in the corresponding CTR pattern.

hasSyncError() TBM Trailer Information: reports if a sync error occurred.

hasSyncTrigger() TBM Trailer Information: reports if the event contains a sync trigger.

hasClearTriggerCount() TBM Trailer Information: reports if the trigger count has been reset.

hasCalTrigger() TBM Trailer Information: reports if the event contained a calibrate signal.

stackFull() TBM Trailer Information: reports if the TBM stack is full.

hasAutoReset() TBM Trailer Information: reports if a auto reset has been sent.

hasPkamReset() TBM Trailer Information: reports if a reset for the beam-scraping event counter has been sent.

stackCount() TBM Trailer Information: reports the current 6 bit trigger stack count.

To allow merging data from different data streams into one event, the augmented assignment operator `+=` is overloaded for this class. Also the `ostream` operator `<<` is overloaded to allow simple printing of event information to `stdout`.

`pxar::pixelconfig` : Class to store the configuration for single pixels, i.e., their mask state, trim bit settings and whether they belong to the currently run test ("enable"). By default, pixelConfigs have the mask bit set, and the enable bit inactive.

`pxar::rocconfig` : Class to hold configuration data for one ROC. This comprises a map with DAC registers and values, the chip type identifier as well as its I2C address and an enable bit.

`pxar::tbmconfig` : Class to hold the configuration of one TBM core. Members are a register map, the TBM device type identifier, the hub ID it responds on together with the core identifier (α, β) and the token chain length(s).

The latter one define how many ROCs are present in every TBM token chain. From this information, the expectation of ROC headers in the decoding modules are derived. In addition, the member function `NoTokenPass()` checks the corresponding register for the no token pass setting and returns its state. This is also used by the decoder modules to adapt their expectation of ROC header data.

`pxar::statistics` : The statistics class hold a set of counters which are filled during data decoding. The `dtbEventDecoder` is defined as friend class and is thus able to directly increment the member variables for book keeping.

All variables can either be retrieved as single values or as cumulative sum of error groups:

info_words_read() Total number of 16 bit words read from the DTB memory.

info_events_empty() Number of events without a pixel hit.

info_events_valid() Number of events with at least one pixel hit.

info_events_total() Sum of the above two: total number of events read.

info_pixels_valid() Number of pixel hits which have been successfully decoded.

errors() Total number of errors encountered during decoding. This is the sum of all errors listed below.

errors_event() Number of errors encountered during event decoding. This is the sum of all event decoding errors.

errors_tbm() Number of errors encountered during processing of TBM header or trailer. This is the sum of all TBM errors.

errors_roc() Number of all errors encountered when processing the event content. This is the sum of all ROC errors listed below.

errors_pixel() Total number of errors encountered during decoding of pixel actual data. This is the sum of all possible pixel data error states.

errors_event_start() Number of missing event start markers.

errors_event_stop() Number of missing event stop markers.

errors_event_overflow() Number of events with overflow marker set. This bit is set if the event readout was too long and the event has been chopped.

errors_event_invalid_words() Total number of invalid 5 bit words detected by the DESER400 module.

errors_event_invalid_xor() Total number of events with invalid XOR eye diagram. This information is stored in the ROC header, a pattern of only high bits indicate that the 400 MHz signal dies not allow the separation of the high and low state.

errors_event_frame() Total number of DESER400 Frame errors indicating a failed synchronization of the deserializer module.

errors_event_idledata() Total number of DESER400 idle data errors. This indicates that the DESER400 module received the TBM header and event data, but the TBM trailer was missing.

errors_event_nodata() Total number of DESER400 no-data error. This indicates that the DESER400 module only received the TBM header but no event data and TBM trailer.

errors_tbm_header() Total number of events with flawed TBM header. This indicates that the first event words do not contain the required TBM header identifier bits.

errors_tbm_trailer() Total number of events with flawed TBM trailer. This indicates that the last event words do not contain the required TBM trailer identifier bits.

errors_tbm_eventid_mismatch() Total number of event ID mismatches in the data stream. The event ID provided by the TBM is cross-checked with a local counter, independently for each of the active TBM channels. In case of a mismatch, the error is recorded and the local counter is re-synchronized with the TBM event ID.

errors_roc_missing() Total number of events with the wrong number of ROC header(s). Headers can either be missing or pixel data can be misidentified as header in case of corrupt data. Both leads to the event being returned as empty.

errors_roc_readback() Total number of misplaced ROC readback start markers. The readback start marker should appear every 16 readouts. If a new start marker appears before or after, it is counted as error and the readback cycle counter is reset for all ROCs in the same token chain.

errors_pixel_incomplete() Total number of undecodable pixels due to missing data missing. This error appears if less than 24 bits of event data are left and no new pixel hit can be formed.

errors_pixel_address() Total number of undecodable pixels due to a non-existing address outside the actual pixel array coordinates. These are mostly induced

by bit errors in the data stream. Noisy pixels are known to sometimes produce invalid addresses.

errors_pixel_pulseheight() Total number of undecodable pixels due to the pulse height fill bit not being zero (cf. Figure 3). Since the fill bit is required to always be zero, this indicates invalid pixel data.

errors_pixel_buffer_corrupt() Total number of pixels with row 80. This invalid address marks a special case and indicates a buffer corruption in the double column buffers of the digital PSI46 chips.

The member function `dump()` prints a summary of all the variables to the INFO logging level (see Section 5.8). To allow summation of statistics from different sources the augmented assignment operator `+=` is overloaded for this class.

5. Configuration and Usage

This section provides an overview of the features and possibilities provided by the pxarCore library. It commences with a description of the start-up procedure of the DTB and the attached devices in Section 5.1, followed by a reference for the DTB signal probe outputs (Section 5.2). The different flags provided for tests and DAQ are described in Section 5.3.

Furthermore, a description of peculiarities and special features in treating different settings for the DTB (Section 5.4), the TBM registers (Section 5.5) and ROC DACs (Section 5.6) are provided. The use of dictionaries is described in Section 5.7, and the logging mechanism as well as its verbosity levels are introduced in Section 5.8. Section 5.9 describes the implementation of the readback data retrieval. The information provided by the collected decoding statistics and the options to retrieve them are detailed in Section 5.10. The section closes with a description for programming and reading out analog PSI46V2 devices in Section 5.11.

5.1. Start-up Procedure

The start-up procedure of a pxarCore library instance is always the same. First, a set of parameters for the connection to the DTB such as the USB ID of the board has to be provided via the constructor of the class, `pxarCore::pxarCore(...)`. With this, the pxarCore object is initialized and the connection to the DTB is established. After successfully connecting to the board, both the DTB and the attached devices are configured as described in the following.

5.1.1. Initialization of the DTB

The first step after creating an instance of pxarCore is the preparation of the DTB for I^2C programming and DAQ. This requires a set of parameters, namely the signal delay settings such *clk, ctr, tin, sda*, the detector supply voltage configuration and current limit settings (*va, vd, ia, id*, given in Volts and Ampere, respectively) and the initial pattern generator setup (cf. Section 3.4).

All inputs have to be provided via vectors of pairs with the setting name as human-readable string, and the value of the parameter. The name look up is performed via the central API dictionaries (cf. Section 5.7). All user inputs are checked for sanity including range checks on the current limits and a validity check for the pattern generator command list ($delay = 0$ at the end of the list). If the settings are found to be out-of-range, a `pxar::InvalidConfig` exception is thrown (cf. Section 4.7).

The initial DTB configuration is performed by the `pxarCore::initTestboard(...)` API function which has to be called at least once after creating a new instance. After the initial setup, the DTB settings can either be changed by calling the same function `pxarCore::initTestboard(...)` again with the updated parameters, or by using one of the functions altering only a subset of the settings:

```
void setTestboardPower(...)
```

```
void setTestboardDelays(...)
void setPatternGenerator(...)
```

5.1.2. Initialization of the DUT

After the DTB has been initialized, the DUT has to be configured using the API function `pxarCore::initDUT(...)`. If the DTB has not yet been initialized, this function returns `false` and prints an error to the console.

The `pxarCore::initDUT(...)` function is overloaded several times for convenience, featuring function calls with some parameters already predefined. The call described in this document features all possible parameters, others can be found in the source code at `core/api/api.h`.

The first argument of the function is a vector of hub IDs which are required for the I^2C protocol for programming the devices. For Layer 2–4 modules featuring one physical TBM, only one value is necessary. Layer 1 modules require two vector entries for the addresses of the two physical TBMs. The hub ID parameter is followed by the human-readable TBM type. The TBM type defines e.g. the number of data acquisition channels required (two 160 MHz signals for TBM08 and four channels for TBM09 and TBM10).

The subsequent TBM register settings should be provided in vectors, with one vector being the parameter set for one single TBM core, i.e. not the physical TBM chip.

Next, the ROC type has to be defined, again as human-readable string. This is of special importance since the pxarCore decoder modules are adapted according to the chip type, e.g. to correctly handle inverted pixel IDs returned by PSI46dig chips, or to decode signals from analog PSI46V2 chips, see Section 5.11.

The ROC DACs should again be provided as vectors, one per ROC in the DUT. The DACs are followed by settings for all 4160 pixels of a ROC. These `pixelConfig` objects (cf. Section 4.8) contain settings for the trim bits, the mask bit and an enable setting which later on defines the test range.

Finally, a vector of ROC I^2C addresses has to be provided. If the vector is left empty, pxarCore assumes consecutive I^2C addresses for all ROCs, starting with address 0.

The `pxarCore::initDUT(...)` checks all settings for validity. This includes DAC ranges, position and number of pixels, number of TBM cores and hub IDs, I^2C addresses, and more. A `pxar::InvalidConfig` exception is thrown if any critical errors are encountered. Non-critical problems such as DAC value overflows are logged with `WARNING` level (cf. Section 5.8) and corrected automatically.

All parameters are supplied via vectors, the size of the vector represents the number of devices. DAC names and device types should be provided as strings. The respective register addresses will be looked up internally via the dictionary mechanism described in Section 5.7. String look-ups are case-insensitively, old and new DAC names are both supported, and all possible names are listed in the Appendix in Table 5.

5.2. Using the Signal Outputs

The DTB features LEMO outputs (cf. Section 3.1) which can be used as scopes for several internal DTB signals. The signals are either provided as digital pulses, which connect to the D1 and D2 outputs, or as analog differential signals on A1± and A2±.

The API call `pxarCore::SignalProbe(std::string probe, std::string name)` is responsible for connecting a specific signal to the outputs and requires two strings as arguments. The first string describes the port to be switched. Accepted values are *D1*, *D2*, *A1*, and *A2*. The second argument is the signal itself, which also has to be supplied as human-readable strings. The corresponding register value is looked up via the internal dictionaries. The signal *off* turns off the output. In case the provided signal identifier is invalid (i.e. it cannot be found in the dictionary) the output is turned off. The full list of currently available signals can be found in the Appendix in Table 7.

5.3. Optional Flags for API Methods

Many functions of the pxarCore API such as the test functions or the DAQ function accept a flags parameter. Several flags can be combined using a logical *OR* and allow to change the behavior of the command to be executed. However, they do not change the internal configuration of the library and do not affect or alter the DUT settings. The flags are only valid for the current command called and have to be shipped again for the next call to the API if desired.

The following list provides an overview of the currently available flags.

FLAG_FORCE_SERIAL: Flag to force the API loop expansion (see Section 4.4.3) to perform tests ROC-by-ROC instead of executing the test for the full DUT (e.g. a module) in one go. This flag should be used with care: it slows down the test procedure considerably in two ways. On the one hand it basically starts several (i.e. for a module 16) completely separate tests and only collects and merges the data afterwards, right before returning it to the user. On the other hand it creates a large overhead in collected data. Reading only one pixel of one ROC in a module results in 16 ROC headers plus TBM header and trailer being sent for every pixel. Without FLAG_FORCE_SERIAL there are 16 pixel hits in every event.

However, there might be some tests where simultaneous readout of all ROCs is not desired, and where the flag has to be applied.

FLAG_CALS: Flag to send the calibration pulses through the sensor via capacitive coupling of the calibrate pad and the sensor instead of sending the signal directly to the preamplifier. This can be used e.g. to test the bump bond connections between sensor and ROC.

FLAG_XTALK: Flag to enable cross-talk measurements. When enabling this flag for a test, one pixel in the neighboring row of the same column of the pixel under test is calibrated instead of the actual pixel under test. If the regarded row is the uppermost in the chip, the row before will be selected for the calibration pulse, otherwise the next higher row. This allows measurements of inter-pixel cross-talk.

FLAG_RISING_EDGE: Flag to define the threshold edge for threshold scans. In threshold scan functions (see Section 4.4.2), a DAC is scanned for an efficiency threshold, and only the DAC value at which the defined threshold level is reached is returned. Since there are both DACs with a falling edge and DACs with a rising edge behavior (and also some that exhibit both) in the spectrum, this flag allows to select the edge on which the threshold measurement triggers.

In other test functions as well as the DAQ functions this flag has no effect.

FLAG_DISABLE_DACCAL: Flag to disable the standard procedure of flipping certain DAC values before programming. This is done by default to flatten the DAC response, taking into account the chip layout of the DAC transistor bank. This function is implemented as Nios II look-up table as described in Section 3.3.4.

FLAG_NOSORT: By default all data returned by pxarCore test functions are sorted according to their ROC and pixel ID. This flag allows to disable sorting of the output data. This flag should only be used in specific cases requiring the original order of the data as read out from the DUT.

FLAG_CHECK_ORDER: Flag to check the order in which the pixels appear in the raw detector readout. The readout follows a specific order defined by the ROC-internal path of the readout token. The token starts at double column 1 and is subsequently transmitted to the consecutive double columns until column 26, and then returned to the periphery. Within one double column, the token is passed upwards in even columns and is returning downwards (to lower row counts) in odd columns. Pixels which are not appearing in exactly this order but in a different position in the readout (e.g. expecting pixel `(13,8)` but receiving pixel `(13,9)` first) are flagged with a negative pulse height. This flag can be used e.g. for pixel address tests to make sure all pixels are answering with their correct address. When running with FLAG_FORCE_UNMASKED in addition, all noise or background pixel hits will be flagged as such by setting the negative pulse height. This allows to easily separate the retrieved pixel data into calibrate hit and noise hit maps.

The current implementation of FLAG_CHECK_ORDER only allows to check the readout order if at least one full ROC is tested, not for single pixel tests (cf. 4.4.3).

FLAG_FORCE_UNMASKED: Flag to unmask and trim all pixels before starting the test. This flag can be used to record background activity during tests, such as a high-rate efficiency map using X-Rays as background. In addition with FLAG_CHECK_ORDER this will return two sets of pixels: the actual efficiency map with positive pulse heights, and the background map with negative pulse height values.

FLAG_DUMP_FLAWED_EVENTS: Flag to dump erroneous events during decoding. With this flag active, every event which triggers a decoding error is printed to stdout together with the previous and subsequent three events. This can be very useful in tracing down errors in the data stream but slows down the decoding

process significantly (roughly by a factor of 3) since additional ring buffers have to be updated holding the history of events to be printed. To be used only if absolutely necessary.

Currently the printout is limited to the first 100 erroneous events encountered during one test or DAQ session.

FLAG_DISABLE_READBACK_COLLECTION: Flag to disable the collection of readback data from ROC headers. This might be desirable when sending several million triggers at once since the amount of readback data collected can be huge. Especially when operating analog PSI46V2 chips it should carefully be chosen whether the readback data is collected or dropped since one readback value is recorded per trigger per ROC header (cf. Section 5.9).

FLAG_DISABLE_EVENTID_CHECK: Flag to disable cross-checking the TBM event id against the local event counter of the pxarCore decoder modules. This flag allows to e.g. only pass every n-th event to the decoder modules without provoking event id mismatch errors.

FLAG_ENABLE_XORSUM_LOGGING: Flag to enable the collection of XOR sum values from the DESER400 modules for every event. The XOR sum is written to the ROC headers as described in Section 3.2 and can be recorded using this flag. The XOR sum values of all ROC headers of the respective DAQ channel are collected in one vector and can be retrieved once via the `pxarCore::daqGetXORsum(channel)` API function for the selected DAQ channel.

5.4. DTB Delays

The DTB delays are programmed initially via the `pxarCore::initTestboard(...)` function but can be altered any time by either calling the same function again or by using `pxarCore::setTestboardDelays(...)`. All values are checked for overflow, the human-readable names are translated into register addresses via the dictionary mechanism (cf. Section 5.7).

A full list of all available DTB settings can be found in the appendix in Table 3. The setting *level* changes the signal gain for all outgoing DTB signals at once, individual gain changes for single signals are currently not supported. The settings *triggerdelay* and *trimdelay* are only active in test loops (cf. Section 4.4.2) and should be given in units of 10 BC.

5.5. TBM Parameters

The TBM Core register settings have to be provided via the `pxarCore::initDUT(...)` function at start-up. Afterwards, single register values can be changed via the DUT object (cf. Section 4.4.1). All possible TBM settings are described in Table 4 in the appendix.

Some TBM settings are no actual TBM registers in the chip but pxarCore-internal values which can be used to influence the behavior of the software. The settings *nrocs* or *nrocs1* and *nrocs2* can be used to change the number of ROCs attached to a specific token chain of the TBM core the settings are supplied for. This can e.g. be used if a module does not feature 16 ROCs but is missing one ROC, and the token is bypassed. With default settings this would lead to decoding errors, since one ROC header is missing in the incoming data stream. Adjusting the *nrocs* parameter for the correct TBM core can mitigate this. The TBM09 and TBM10 feature two token chains with four ROCs each per TBM core, hence the two settings.

5.6. ROC Parameters

The ROCs feature digital-to-analog converters and registers which need to be set correctly in order to set the chip in a working state. Since all functions accepting ROC parameters make use of the pxarCore dictionaries, parameters are supplied using their human readable names and are not case sensitive. This eases code review and understanding of test procedures when reading code.

All common names for DACs are understood by the dictionaries. The names returned for DACs are the ones defined in [19] for the PSI46digV2.1 chip and in [20] for older digital chip prototypes.

5.6.1. DAC Programming

DACs are always programmed in the same order, which is defined by their ROC-internal register number. This means that e.g. *Vana* will always be programmed before *VthrComp*. This also ensures, that the *RangeTemp* DAC with register 255 is always the last DAC to be programmed. This DAC is only available on analog PSI46V2 chips and enables temperature readout via the lastDAC functionality (see Section 5.9).

If the *WBC* register is programmed, an automatic ROC reset command will be issued after all DACs have been set in order to restart the bunch crossing counter with the new offset defined by the WBC trigger latency.

All DACs are programmed directly at start-up of the DUT, i.e. when calling the function `pxarCore::initDUT(...)`. DACs subsequently supplied via `pxarCore::setDAC(...)` are also directly sent to the device.

5.7. Using the Dictionaries

To allow input variables being identified with their commonly used names instead of easy-to-forget registers numbers, the pxarCore library provides dictionaries translating the common names into their corresponding register values automatically. In addition, the valid range for values of every register is stored, and input values are checked to comply with them.

The dictionary includes DTB setting names, TBM registers, and ROC DACs as well as device names (for TBMs and ROCs), the different signals for the DTB probe outputs,

possible Pattern Generator signals, and trigger sources. A reference to all dictionary entries can be found in the appendix of this document (Dictionary Reference, Section A).

Dictionaries can as well be used in user code. The header file to include can be found at `core/util/dictionary.h`. A dictionary is instantiated and used e.g. via

```
// Get singleton register dictionary object:
RegisterDictionary * _dict = RegisterDictionary::getInstance();

// Translate the register name to its address:
uint8_t id = _dict->getRegister(name,ROC_REG);

// Read register value limit:
uint8_t regLimit = _dict->getSize(id, ROC_REG);
```

and similar for other dictionaries. The list of all available dictionary entries can be fetched via their `getAllNames` directive:

```
// Get singleton trigger source dictionary object:
TriggerDictionary * _dict = TriggerDictionary::getInstance();

// Fetch all possible trigger sources from the dictionary:
std::vector<std::string> names = _dict->getAllNames();
```

This function only returns the preferred name for the respective dictionary entry, while still all options and synonyms are understood in the lookup process.

5.8. Logging and Verbosity Levels

The pxarCore library employs a flexible logging system which allows to select and switch logging levels at run time without requiring recompilation of the source code or a restart of the program. The logging is implemented such that calls with a verbosity level lower than the currently selected are not executed and thus do not affect the performance of the program.

The verbosity level can be changed any time at run time by calling

`Log::ReportingLevel() = Log::FromString(LEVEL);`

where `LEVEL` is one of the below listed verbosity levels as string literal. In addition to logging to the console, all logging messages can be redirected to a log file by supplying a file pointer `FILE*` to the logging mechanism:

`SetLogOutput::Stream() = filepointer;`

When logging to file, the printing to `std::cout` is switched off by default. In order to have the messages available in both the log file and on-screen, the duplication of the logging screen has to be requested explicitly:

`SetLogOutput::Duplicate() = true;`

Currently the following verbosity levels are implemented:

QUIET: Turns off all messages except for the pxarCore constructor's welcome message and messages about opened or closed connections to DTBs.

CRITICAL: This verbosity level contains all above messages plus critical errors which occurred during the execution of a command. Critical messages are e.g. missing events during readout or bad signal quality at the DTB inputs. In most cases, CRITICAL messages are accompanied by an exception being thrown.

ERROR: Contains all above messages plus errors which are not critical. Not being critical does not imply that the errors can be ignored, the error condition just does not necessarily call for an immediate abortion of the command being executed.

WARNING: Contains all error messages plus information which is classified as warning such as decoding problems with pixel addresses or similar.

INFO: This is the default verbosity level. It contains all error and warning messages as well as additional information e.g. about the run time for a test. This level is optimized to give useful information without flooding the console with messages.

DEBUG: The DEBUG level includes all INFO messages and above, but can include additional messages for debugging purposes. Currently this level is not used by pxarCore but can be populated by user code and scripts.

DEBUGAPI: This verbosity level is the highest pxarCore-internal debugging level and contains information from API-level function calls such as the test functions (cf. Section 4.4.2), data repacking routines (cf. Section 4.4.4), or the initialization functions for DTB and DUT (cf. Sections 5.1.1 and 5.1.2, respectively).

DEBUGHAL: The HAL verbosity level includes all above messages and gives additional information about functions called on HAL level (cf. Section 4.2). This can be useful to see which routines are finally called and what the parameters are which have been calculated or propagated. Also the low-level `pxarCore::daqStart(...)` function resides in HAL and prints useful information on DEBUGHAL level.

DEBUGRPC: This logging level is quite verbose since it prints every single RPC call sent to the DTB, together with all other messages from higher levels. This can be very useful to compare pxarCore behavior with other software packages on a RPC call-by-call level to find differences in how the hardware is programmed.

DEBUGPIPES: The DEBUGPIPES verbosity level can be used to look at the data decoding routines bit-by-bit. Decoding information including TBM header and trailer information will be printed for every single event read from the DUT. This level is highly verbose and slows down the decoding procedure considerably. It is advised not to enable this level during data acquisition of any kind, be it during module production tests or in test beam. This setting is only to be used for debugging purposes.

INTERFACE: This verbosity level is the lowest available and prints (together with all other messages) information on the system's interface level (USB or Ethernet).

5.9. Retrieving Slow Readback Data

Most PSI46-type ROCs feature a mechanism to retrieve detector control data from the chip via the normal data stream of ROC header and pixel hit information. The way these values are transferred is different for analog and digital ROCs.

The pxarCore library automatically collects these data for all ROCs attached to the readout whenever triggers are sent and the data stream is decoded, unless the collection is disables using the appropriate flag (cf. Section 5.3). The readback values are only collected if the data is decoded. If raw data is requested, no readback data is collected. The cached readback data can be retrieved using the `pxarCore::daqGetReadback(...)` function which yields one vector of readings for every ROC. The internal readback value cache is reset once a new DAQ session or another test is started.

Digital PSI46digV2 ROCs and later: Readback Bits The digitally read out PSI46digV2 chips and later versions use an on-chip ADC to digitize different internal parameters of the chip. The data is then sent out via the data bit `D` in the ROC header. The transmission start is marked by the status bit `S` going high. The next 16 data bits form one 16 bit word representing the quantity selected via the `readback` register (cf. Table 5). Possible values for the register can be found in [19].

Analog PSI46V2 ROCs: lastDAC For analog ROCs, the third word of the ROC header is an analog value representing the setting of the last DAC programmed. When setting the `vrangetemp`, an on-chip temperature reading can be retrieved since the pxarCore library automatically programs this DAC last if supplied.

5.10. Using the Statistics Collection & Error Reporting

To allow an in-depth analysis of problems occurring during data acquisition, the pxarCore library features a statistics collection class embedded into the decoder modules. An extensive set of parameters and error conditions are collected and can be retrieved by the user.

The errors are classified according to their occurrence. Event errors are recorded if the start or end bits of an event do not exist or are corrupted, an overflow is detected, or error bits set by the deserializer modules are present in the event. This most likely points to a problem with handling of the incoming data stream in the DTB. In most cases the deserializer module failed to latch to the idle pattern or the data quality is too bad to correctly decode the event.

If the TBM header or trailer is missing its identifier bits and thus is invalid, TBM errors are recorded. Also, the trigger ID reported by the TBM is checked against an pxarCore-internal event counter, and mismatches are registered and reported.

Errors appearing on ROC level are e.g. missing ROC headers or misplaced readback markers (PSI46digV2 and later only).

All errors encountered when decoding pixel hits are classified as pixel errors. Usually these are invalid pixel addresses, but also invalid pulse height fill bits, incomplete pixel

hits (missing data) and corrupt pixel buffers are reported.

The latest `pxar::statistics` object can be retrieved by calling `pxarCore::getStatistics()` after a test or during a DAQ session. The returned class implements some convenient functions returning the total number of errors either in one particular value, in a class of errors, or in total. A list of all collected data can be found in Section 4.8. In addition, some purely informational data are recorded, such as the total number of words read, the number of overall pixel hits, empty events, and events containing pixel data. In addition, the `pxar::statistics::dump` function can be used to print an overview table of all values on INFO verbosity level (cf. Section 5.8). The overloaded augmented summation operator `+=` eases summing up statistics from several calls to `pxarCore::getStatistics()`.

It has to be noted that statistics are meant for one-time-reading. After calling `pxarCore::getStatistics()` the object will be returned and the internal statistics reset. Calling `pxarCore::getStatistics()` a second time will yield an empty object. This implies that the object should be stored locally in a variable if several values are supposed to be read from the object, i.e. accessing the stored variable class members using `statistics mystat.pixel_errors()` instead of reading single values directly from the pxarCore object using `pxarCore::getStatistics().pixel_errors()`.

The statistics class object gets filled by the pxarCore decoder modules. This has implications when not running standard tests but dedicated DAQ sessions using the API methods `pxarCore::daqStart()` and `pxarCore::daqStop()`. If the data is fetched via `pxarCore::daqGetRawEvent()` or `pxarCore::daqGetRawEventBuffer()` the statistics object will remain empty because the data has not been passed through the decoder. If events are read through the API methods `pxarCore::daqGetEvent()` or `pxarCore::daqGetEventBuffer()` the statistics object will be properly filled. New incoming events will continue to be added to these numbers until either the statistics are read out or the DAQ session is restarted since `pxarCore::daqStart()` re-initializes the decoder modules and thus resets the counters.

5.11. Reading Out Analog PSI46V2 ROCs

Since the DTB features an internal ADC which can be connected to a set of input signals, it is also possible to apply analog signal sampling on the incoming data lines. This enables data acquisition for analog PSI46V2 ROCs. The pxarCore library provides all necessary code to seamlessly program, read out, and decode data from analog PSI46V2 chips. Beside the ROC type supplied via the `pxarCore::initDUT(...)` function, which should be set to *psi46v2*, there are a couple of other parameters which need to be tuned in order to obtain a functioning data acquisition and decoding setup. These parameters and their determination procedures are described in the following.

The chip type already triggers all necessary changes in the DTB firmware: the ADC is selected as data input source of the *sdata1* signal, and the signal gain is adjusted. Since the data sampling is controlled by the readout token, two delays need to be adjusted to get the correct number of samples to match the event length. In addition, the sampling point of the ADC has to be adjusted within one clock cycle to ensure stable signal sampling.

With these settings properly configured most of the test supplied in the pXar test suite can be executed. Tests which are tailored to a specific behavior of a certain chip, such as the pulse height optimization, will not work without changes to the algorithm.

5.11.1. Token In, Token Out

The sampling of the analog input data is triggered by the flying token. After the Token In command has been sent to the chip, and an additional delay *tindelay* (in units of BC) the ADC starts sampling the incoming 40 MHz data on *sdata1*. The sampling is stopped after the Token Out from the attached device has been received back and the additional delay *toutdelay* has passed. The timeout to cut off the readout for a non-returning token signal can be configured using the *adctimeout* setting in units of 10 BC and defaults to 300 BC.

The two token delays have to be adjusted according to the data in the events. Every event should start with a ROC header marked by the Ultra Black - Black patterns followed by the lastDAC word (see Section 5.9), every pixel hit is six samples long. A more detailed reference of the PSI46V2 analog data format can be found in [21].

The adjustment can be either performed by hand by looking at the raw, undecoded data, or by using the automated script provided on the Python command line interface (cf. Section 6.1) called `FindAnalogueTBDelays`. This will automatically scan through *tindelay* and *toutdelay* and search for the Ultra Black in the ROC header and the pulse height in the last sample of a pixel hit.

Adjusting the delays by hand can be accomplished using the Python CLI. It should be made sure that a set of DAC parameters appropriate for analog PSI46V2 chips is used.

```
$ # Running with DEBUGPIPES verbosity here reveals lots of information
$ ./python/cmdline.sh -d myConfigDir -v DEBUGPIPES
> # Power up the device and check if it draws current:
> getTBia
> getTBid
> # Activate one pixel and unmask it:
> testPixel 13 8 1
> maskPixel 13 8 0
> # Call the varyDelays function with some settings for tindelay and toutdelay:
> varyDelays <tindelay> <toutdelay> True
> # Look at the data printed, try to identify the Ultra Black and Black levels
...
```

It might be necessary to readjust the delays if the sampling point is shifted (cf. next section).

5.11.2. ADC Sampling Point / Address Levels

The sampling point of the ADC within one clock cycle has to be set correctly in order to have a pulse height sampling independent of the preceding pulse. If the sampling point

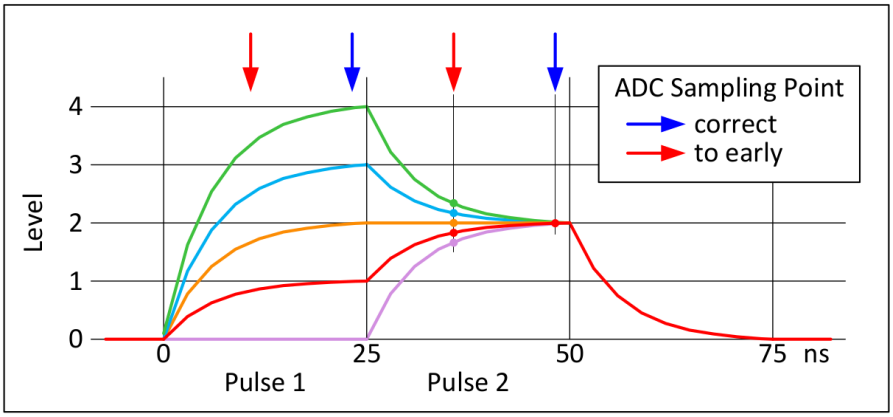


Figure 11: Influence of the ADC sampling point on the measured level, depending on the preceding pulse. In order to achieve a stable sampling the phase should be adjusted such that the sampling is done late in the clock cycle.

is selected incorrectly, the address levels are split up depending on the preceding level as indicated in Figure 11.

The ADC sampling point is controlled by the *clk* DTB delay setting. The sampling can be adjusted by hand by looking at an address levels histogram as the one collected and plotted by the Python CLI function `analogLevelScan` (cf. Section 6.1) or by running the Python CLI script `find_clk_delay` which automatically picks the optimal setting. The script checks for the sampled value of a specific level for all transitions from other levels. It then scans through the different clock delay settings and chooses the optimal working point where the minimum level split can be found as demonstrated in Figure 12.

The procedure to record an address levels histogram using the Python CLI could be:

```
> # Activate one pixel:
> testPixel 26 9 1
> maskPixel 26 9 0
> # Open the ROOT Canvas to display the histogram:
> gui
> # Run the test (simple raw DAQ):
> analogLevelScan
```

Which address levels appear in the plot depends on the Gray code address of the pixel activated. An example is shown in Figure 13. All levels should be nicely separated and the peaks equally spaced for an optimal working point.

Shifting the clock phase might require re-adjustment of the *tindelay/toutdelay* delays as described in the previous section.

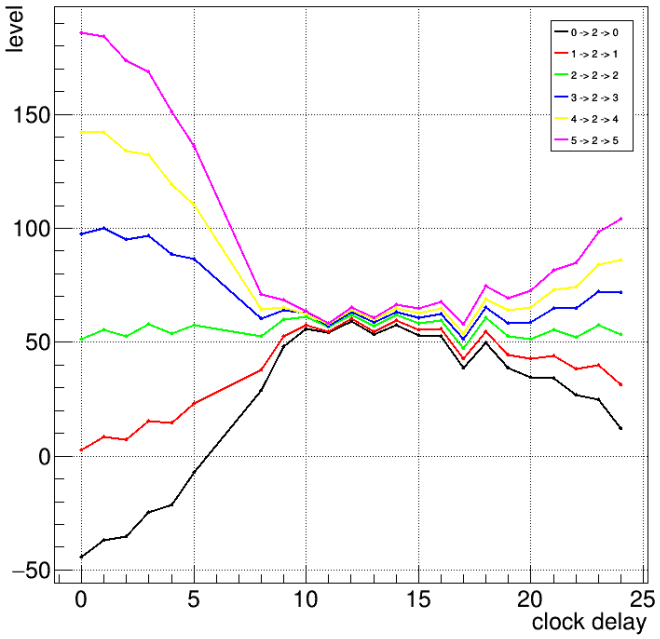


Figure 12: Automated adjustment of address levels separation. Shown is the sampled amplitude of address level 2 for different transitions as a function of the ADC sampling point (clock phase delay). The delay should be chosen such that the sampled amplitude is independent from the transition from previous pulses. Figure by [22].

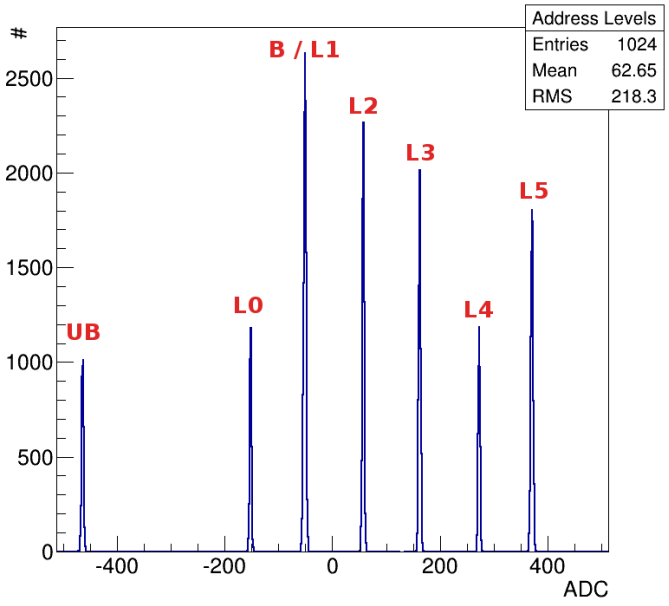


Figure 13: Ultra black (UB), black (B), and address levels (L) from an analog PSI46V2 chip recorded using pxarCore. All levels show a nice separation and equidistant spacing.

5.11.3. Data Decoding

The analog decoder module of pxarCore is self-calibrating. It uses the Ultra Black and Black levels of the first ROC header to calculate the binning of all other address levels. The Ultra Black and Black levels are taken from a sliding window average of the last one thousand events decoded in the same DAQ session. If the DAQ is stopped and re-started, the average levels will be reset and re-calculated with the next incoming events.

This is a very clean solution for analog data decoding since no additional address level files have to be provided by the user. The approach has proven to be stable in various environments, including test beams with a multi-plane telescope featuring four PSI46V2 planes. However, for irradiated chips or in very noisy environments, this approach might fail and may not produce valid output.

Currently, there is no support for analog TBM (`TBM05`) data, hence analog modules cannot be read out.

6. Additional Tools & Resources

In addition to the standard ROOT GUI *pXar* there are a series of other tools using the pxarCore library as their backend. This section gives a non-exhaustive overview of tools developed by the user community. Usage scenarios and example code will be reviewed in Section 7.

6.1. Python Command Line & Plotter

The Python CLI is a convenient tool to directly access pxarCore API calls to test settings, software, or to even perform full detector tests. It makes use of the Python bindings of the API functions as described in Section 4.5. With most of the API calls available, the command line is very powerful and allows the setup of various testing environments, switching between different trigger sources and direct reconfiguration of the DUT object at run time.

The interface comes with tab completion for both the API commands as well as for their parameters such as DAC names. At start-up, configuration files compatible to the ones produced by the pXar GUI can be loaded and the pxarCore object is automatically initialized. This allows to quickly switch between running tests picked from the pXar test suite and calling single API commands from the Python CLI.

To ease the test of a sequence of commands, simple scripts of PythonCLI commands can be executed consisting. Lines starting with a hash symbol are interpreted as comments and ignored. The scripts can be either run at start-up of the CLI using the "-r" command line parameter or from the command line directly by the `run` command followed by the file name containing the script. A simple script activating external triggers and clock and starting the data acquisition could e.g. look like

```
# Switch off calibrate signal for all pixels and unmaks them:
testAllPixels 0
maskAllPixels 0
# Enable external triggers as trigger source:
daqTriggerSource extern
# Switch the DTB to the externally supplied clock:
setExternalClock 1
# Start the data acquisition, enable the correct deserializer:
daqStart
```

Then, single events can be read out, decoded and printed to either console or the ROOT canvas as event display:

```
# Read one event and decode it:
daqGetEvent
```

More complex scripts requiring more Python syntax such as loops or conditional statements can easily be implemented making use of the API calls as well as of the Python

helper functions which parse configuration files and initialize the pxarCore object correctly. An example serving as starting point for own implementations can be found in the file "python/script.py".

All pxar data type classes have a Python pendant and can thus easily be used within Python, e.g. events can simply be printed in an already formatted string. If the ROOT libraries are present at start-up of the command line, a simple ROOT Canvas display is available to plot test results or event displays. The Canvas window can be opened using the `gui` command any time or the "-g" command line parameter at start-up.

The following command line parameters are available:

```
cmdline.py [-h] [--dir DIR] [--verbosity LEVEL] [--gui] [--run FILE]
```

`-h, --help` : show the help message and exit

`--dir DIR, -d DIR` : path to the directory with all required configuration files, compatible with the pXar configuration file format

`--verbosity LEVEL, -v LEVEL` : logging output verbosity of the pxarCore library (cf. Section 5.8)

`--gui, -g` : start the ROOT Canvas display for test results and event displays

`--run FILE, -r FILE` : load the command line script `FILE` to be executed before entering the prompt

6.2. pyXar

pyXar [3] is a small command line-based tool written in Python that implements many tests and calibration procedures for PSI46 chips as well as high-rate tests under X-Ray background. This tool was originally developed at ETH Zürich as independent tool, and later ported to using the pxarCore library as backend.

pyXar uses the Python bindings provided by the pxarCore library. All test loops and command sequences sent to the detector and DTB are completely identical to other software packages using the pxarCore library.

6.3. decoder

The decoder executable is a small tool in the pxar repository demonstrating how to use the decoder modules and data pipes of the pxarCore library in an external context, e.g. decoding already recorded PSI46 device data from binary files.

The decoder tool sets up a decoding chain using the pxarCore decoder modules:

```
#include "datasource_evt.h"
#include "constants.h"

  // Define all components of the decoding chain:
  evtSource src;
```

```
dtbEventSplitter splitter;
dtbEventDecoder decoder;
dataSink<Event*> Eventpump;

// Set up the chain using the output operator:
src >> splitter >> decoder >> Eventpump;

// Create the data source with correct configuration
// Here the source is set up for Channel 0 of a TBM08c module
// with 8 ROCS of type PSI46digV2.1
src = evtSource(0,8,0,TBM_08C,ROC_PSI46DIGV21);

// Now we can add data to the event source:
std::vector<uint16_t> myrawdata = read_data_from_somwehere();
src.AddData(myrawdata);

// And retrieve all events until the buffer is empty:
while(1) {
  try { pxar::Event * evt = Eventpump.Get(); }
  catch(dsBufferEmpty &) { break; }
}
```

6.4. flash

The flash tool is a tiny helper which allows to flash new firmware releases onto the DTB. Invoke with:

```
./flash  FLASH-FILE
```

This tool is only there for convenience when not building the pxar GUI executable, but provides the same functionality as calling

```
./pXar -f FLASH-FILE
```

Section 2.5 provides a more detailed description of the DTB firmware flashing procedure.

6.5. Resources

The following list provides some additional resources concerning the pxarCore library, the user community and related tools.

- The pxar repository:
  `https://github.com/psi46/pxar`
- pxar Releases:
  `https://github.com/psi46/pxar/releases`
- pxar Bug Tracker:
  `https://github.com/psi46/pxar/issues`
- pxarCore API Doxygen reference:
  `http://psi46.github.io/classpxar_1_1pxarCore.html`
- The DTB Firmware repository:
  `https://github.com/psi46/pixel-dtb-firmware`
- The pyXar repository:
  `https://github.com/simonspa/pyxar`
- The psi46 tools repository with the `rpcgen` tool:
  `https://github.com/psi46/tools`
- CMS internal pXar TWiki page:
  `https://twiki.cern.ch/twiki/bin/view/CMS/Pxar`
- CMS internal HyperNews channel:
  `hn-cms-pixel-psi46-testboard@cern.ch`

7. Using pxarCore - Examples

This section contains a few examples for use cases of the pxarCore library. The code here should be seen as pseudo-code since it can not be compiled as is. However, the syntax, command sequences and structure reflect the required functionality.

7.1. Simple Data Acquisition

This section gives some examples on how to set up a DAQ using the pxarCore library. The examples are kept simple while a more elaborate implementation into a full-featured DAQ framework is described in detail in Section 8.

7.1.1. C++

Initialization of the testboard:

```
// Create new API instance:
try {
  _api = new pxar::pxarCore("*","INFO");

  // Initialize the testboard:
  if(!_api->initTestboard(sig_delays, power_settings, pg_setup)) {
    return;
  }

  // Initialize the DUT:
  if (!_api->initDUT(hubid,"tbm08c",tbmDACs,"psi46digv2.1",rocDACs,rocPixels)) {
    return;
  }

  // Unmask all pixels:
  _api->_dut->maskAllPixels(false);
}
```

DAQ with external triggers:

```
// Set the trigger and clock sources to extern:
_api->daqTriggerSource("extern");
_api->setExternalClock(true);

// Start the DAQ:
_api->daqStart();

// Read 1000 events:
size_t nevents = 0;
```

```
std::vector<pxar::Event> events;

while(1) {
  try {
    events.push_back(_api->daqGetEvent());
    nevents++;
  } catch(dsBufferEmpty &) { break; }
  if(nevents >= 1000) { break;}
}

// Stop the DAQ:
_api->daqStop();

// Read the remaining events from the buffer:
std::vector<pxar::Event> tmp = _api->daqGetEventBuffer();
events.insert(events.end(), tmp.begin(), tmp.end());
```

DAQ with triggers from the pattern generator:

```
// Start the DAQ:
_api->daqStart();

// Send 1000 triggers with a distance of 500 BC:
_api->daqTrigger(1000,500);

// Read all events:
std::vector<pxar::Event> events = _api->daqGetEventBuffer();

// Stop the DAQ:
_api->daqStop();
```

DAQ with randomly generated triggers:

```
// Activate the random trigger generator (direct)
// with a trigger frequency of 100 kHz:
_api->daqTriggerSource("random_dir",100000);

// Start the DAQ:
_api->daqStart();

// Triggers are being sent...

// Stop the DAQ:
_api->daqStop();
```

```
// Read all events:
std::vector<pxar::Event> events = _api->daqGetEventBuffer();
```

7.1.2. Python

Initialization of the testboard:

```
import PyPxarCore
from pxar_helpers import *

# Use the startup Python helper:
api = PxarStartup("config/mychip","INFO")
```

DAQ with external triggers:

```
# Enable external clock and trigger:
self.api.daqTriggerSource("extern")
self.api.setExternalClock(1)

# Start DAQ
self.api.daqStart()

# Get events:
events = list()
while len(events) < 1000:
  try:
    event = self.api.daqGetEvent()
    events.append(event)
  except RuntimeError:
    pass

# Stop the DAQ:
self.api.daqStop()
```

DAQ with triggers from the pattern generator:

```
# Start DAQ
self.api.daqStart()

# Generate 1000 triggers with 500 BC spacing:
self.api.daqTrigger(1000,500)

# Retrieve the data:
try:
```

```
  data = self.api.daqGetEventBuffer()
except RuntimeError:
  pass

# Stop the DAQ:
self.api.daqStop()
```

DAQ with randomly generated triggers:

```
# Activate the random trigger generator (direct)
# with a trigger frequency of 100 kHz:
self.api.daqTriggerSource("random_dir",100000)

# Start DAQ
self.api.daqStart()

# Stop the DAQ:
self.api.daqStop()

# Retrieve the data:
try:
  data = self.api.daqGetEventBuffer()
except RuntimeError:
  pass
```

7.2. Running Tests with the Python Command Line Interface

The Python CLI already contains helper functions to completely configure and initialize the devices at start-up. For this, a configuration directory with the standard pXar configuration files is needed.

On the command line, all test loops such as `getEfficiencyVsDAC` are available and can be executed. All commands feature tab completion and print information about the required parameters.

If the ROOT libraries can be found in the system `PATH` and the ROOT Python interface is available, a ROOT canvas can be opened by calling

```
> gui
```

which will be used to plot the test results. If no canvas is available, the result will be printed in the terminal.

7.3. Running Tests with a Simple Python Script

More complex test algorithms might require their own script with the full power of Python available. At the beginning the pxarCore Python bindings have to be loaded, and also the `pxar_helpers` are helpful to read configuration files and start up the DTB.

The repository contains a sample script in the file

```
python/script.py
```

demonstrating the simplicity of such an implementation. The script `module_test.py` in the same directory provides a real-world example of a script used for long-term stability tests of the module readout.

7.4. Running Non-Standard Devices

This section aims to give some advice on how to proceed with non-standard devices. The pxarCore library is able to cope with most device configurations but requires specific configuration parameters in order to correctly address all attached detectors.

Multiple ROCs, but no TBM: in some setups, ROCs are chained together, appending their data to the previous ROC and passing the token on without being coordinated by a TBM. The DESER160 module can sample the data from all ROCs since it is steered only by the flying token. In order to correctly decode this data with pxarCore, only the correct number of ROC DACs have to be presented to the `pxarCore::initDUT(...)` function at start-up.

Modules with bypassed/fewer ROCs: Some modules either feature less then the usual 16 ROCs or have broken ROCs which are bypassed and never receive the token. In these cases, pxarCore has to be notified in order not to throw error messages about the missing ROC header in the data stream. There are two possibilities of doing this.

If the module has a standard I^2C address configuration, it is sufficient to supply the I^2C addresses of the functioning ROCs via `pxarCore::initDUT(...)`, excluding the non-existing/bypassed ROC. pxarCore will then automatically assign the available ROCs to the correct TBM channels. If for example, the ROC with I^2C address 2 is broken and bypassed, only the configuration of the other fifteen ROCs together with the list of valid I^2C addresses

```
0,1,3,4,5,6,7,8,9,10,11,12,13,14,15
```

has to be provided at startup. The library would automatically assign the missing ROC to the token chain of TBM core α according to its I^2C address.

The other possibility is to explicitly state which TBM channels are missing the ROC (s). This can be done by specifying the TBM parameters `nrocs (nrocs1)` (and also `nrocs2` for TBM09 with two channels per core) in the TBM configuration vector supplied to the `pxarCore::initDUT(...)` function. The number of expected ROCs is then reduced for the respective channels.

Multiple DTBs in Sync: When running multiple DTBs in synchronization with external triggers (e.g. in test beams) it is advised to use the same external clock for all

boards. This synchronizes the device clocks and allows truly synchronous triggering and data acquisition.

Furthermore, the configuration files for the different DTBs are required to contain the actual DTB USB id instead of the wildcard *. With the wildcard present, the assignment of pxarCore instances to the DTBs depends on the order of the USB ports in the system.

Non-standard I^{2}C addresses: Non-standard I^{2}C addresses can be specified using the I^{2}C address vector in the `pxarCore::initDUT(...)` function. The addresses will be assigned to the ROCs in the same order as the DAC vectors are provided.

8. pxarCore EUDAQ Integration

The pxarCore library has been integrated with the EUDAQ software framework [23] to enable data taking with the DTB in a more complex environment such as test beams. Usually the requirements on integration, communication and control in such experiments are higher than in simple laboratory setups. This section describes how all functionality needed for such experiments is seamlessly integrated into the existing data acquisition framework allowing data taking together with other detectors, externally supplied triggers and a unified event building and data storage.

8.1. The EUDAQ Software Framework

EUDAQ is a modular cross platform data acquisition framework designed in the context of the EUDET project [24]. It consist of completely independent modules such as Run Control, Data Collector and Producers. The communication between individual modules is implemented as TCP/IP which allows for running them on separate machines, only linked by a (preferably Gigabit) Ethernet network.

The central interaction point for the user is the Run Control and its GUI. All producers connect to the Run Control at start-up and retrieve additional information from there during operation (such as the commands for starting and stopping a DAQ run). The GUI provides all controls necessary to the user on shift.

Producers are the links between the EUDAQ framework and the user DAQ system. They interface with the EUDAQ library and have to provide a certain set of functions to be called by the Run Control. The data read out from user DAQ systems by the individual producer is sent to the so-called Data Collector. This Data Collector is responsible for the event building, i.e. the correlation of events from all subdetector systems to single global events comprising all data belonging to one trigger. Basic sanity checks are performed, such as matching event serial numbers from the individual subdetectors.

In order to ensure a high data quality already during the acquisition the Online Monitor is available. It connects to the Data Collector requesting a fixed fraction of the recorded events (e.g. one out of a hundred) to fully decode all subdetector data and build basic plots such as hit maps or correlation plots showing that the different devices are synchronized in time and space, and are all placed within the geometrical trigger acceptance.

The data decoding is performed using Data Converter plugins for every detector type attached to the Run Control. The plugin to be called is deducted from the event type transmitted by the producer and written to the data stream by the Data Collector. Each Data Converter plugin can implement several data format end points to allow e.g. both the conversion to the EUDAQ internal format `StandardEvent` for the Online Monitor, and to `LCIO` which is used by the EUTelescope offline reconstruction software [25].

8.2. The CMSPixelProducer

The *CMSPixelProducer* integrates the pxarCore library with the EUDAQ run control. It provides the interface functions for the EUDAQ finite state machine (`OnConfigure`, `OnStartRun`, `OnStopRun`, `OnTerminate`, and the `ReadoutLoop`) to allow integrated data acquisition with other producers via the Run Control. It is included in the main EUDAQ repository since the release EUDAQ v1.4.5 but a more up-to-date version with additional features might be present in the development repository, branch *cmspixproducer* [26].

When running a DAQ with a TBM and multiple ROCs configured, the CMSPixelProducer will automatically assemble all ROCs found in the readout stream to a module-like pixel plane for both online monitoring and correlation as well as for the conversion to other formats.

All functions provided by pxarCore are supported, including operation of all sorts of PSI46 devices such as analog PSI46V2 ROCs as well as custom-built telescopes using PSI46 devices as telescope planes. Even beam telescopes with several planes consisting of full CMS Pixel modules featuring 16 ROCs each have been successfully operated [27].

The CMSPixelProducers checks all supplied configuration parameters for consistency and catches exceptions thrown by pxarCore (cf. Section 4.7). Error messages and exceptions are displayed in the Run Control status as shown in Figure 14 and also sent to the Log Collector. This allows to identify improper detector configuration already at configuration stage before the actual data acquisition starts.

Important parameters for offline interpretation of the data recorded such as the type and the number of ROCs operated, all DAC parameters as well as the TBM type are written into the so-called begin-of-run event (BORE) in form of tags consisting of a name-value pair. These tags can be read from the raw EUDAQ data files and used for correct interpretation of the data (cf. Section 8.7).

Following the paradigm of the EUDAQ framework, the detector data read during data taking is stored as-is, meaning that no data decoding is performed online prior to storing the data to disk. This has the advantage that possible flaws in the decoding methods do not affect the data taken, but the conversion to other event formats can just be re-run on the raw and unaltered detector data.

8.3. Installation

Compilation of the CMSPixelProducer together with the EUDAQ library has to be enabled explicitly at configuration by using a CMake switch:

```
cmake -DBUILD_cmspixel=ON ..
```

In order to build the module successfully, the dependency on the pxarCore library has to be satisfied. CMake searches for the pxarCore library in a path provided by the environment variable `$PXARPATH`. It should be set up to point to the root directory of the local pxar installation.

After a successful compilation and installation via `make && make install` the producers are ready for data taking. More detailed installation instructions for the EUDAQ software framework can be found in [23].

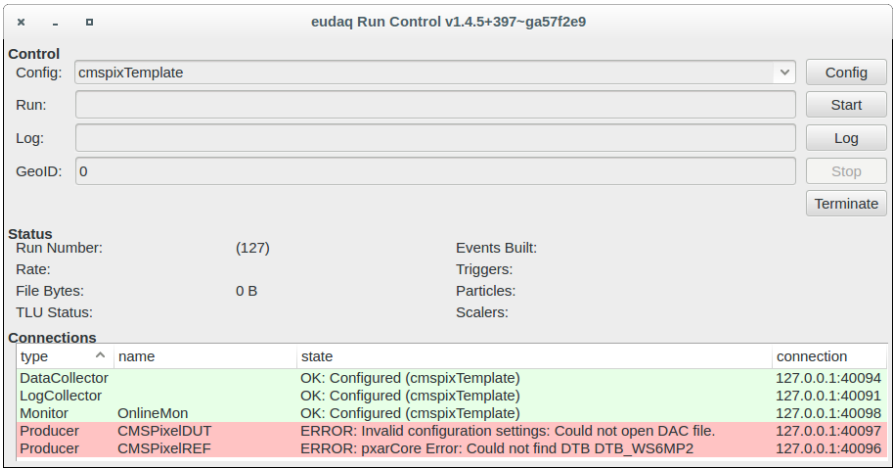


Figure 14: EUDAQ Run Control with two CMSPixelProducer instances attached. The two producer instances are in error mode, the DUT producer due to a non-existing DAC parameter file, the REF producer due to a missing DTB USB connection. This error message display allows the person on shift to quickly identify problems.

8.4. Starting Instances of the CMSPixelProducer

In EUDAQ, each decoding module being called for a certain subdetector event data is currently only deduced from the event type written. This means that as soon as two producers write exactly the same event type, the decoding factory will only instantiate one Data Converter plugins and then call it for every subdetector event. The result would be a mixing of the data recorded from different DTBs and ROCs. To overcome this current EUDAQ limitation, the CMSPixelProducer provides several different event types allowing to clearly separate the data streams both during data acquisition and data decoding.

The event type has to be defined at start-up of the producer. For the DUT data stream (the first DTB to be used) no additional command line option has to be provided when starting the CMSPixelProducer, the event type `CMSPixelDUT` is automatically selected. For additional DTBs a different producer name has to be specified, which in addition allows to separate e.g. the logging output from the two producers:

```
./CMSPixelProducer.exe -n NAME
```

For the second producer (and DTB) the name should contain the word `REF` in all capitals (operating the timing reference plane, name e.g. `CMSPixelREF`), additional producers should be started using a name containing the patterns `TRP` or `QUAD`, such as `CMSPixelTRP` (for *triplet*) or `CMSPixelQUAD` (for *quadruplet*), respectively. An example of two producers connected to the Run Control can be seen in Figure 14.

8.5. Configuration Parameters

The configuration parameters for every producer are read from a global configuration file by EUDAQ and distributed to the producers. The configuration file is a plain text file which contains one section for each producer. Each of the sections consists of tag-value pairs for the configuration of the respective producer. Sections for producers which are currently not connected are ignored. All lines starting with a hash symbol are interpreted as comments and ignored.

The full content of the configuration file including comment lines is included in the BORE header and is thus available later for offline analysis and reference. This greatly simplifies logging during test beam shifts since all parameters, such as chip settings, are stored automatically.

Below is a list of currently available parameters for configuration of CMSPixelProducer instances. When running several producers, the settings have to be duplicated to a new section featuring the correct name of the producer.

roctype: The device type of the ROC to be operated. This will be fed to the function `pxarCore::initDUT(...)` as described in Section 5.1.2. The list of available devices can be found in Table 2.

pcbtype: Type of the carrier printed circuit board (PCB) the ROC is mounted on. Content of this is free and it can be used to keep track of different PCB types, e.g. distinguishing different material budgets. If the PCB type parameter contains the pattern `-rot` indicating a PCB with the ROC mounted in a $\pi/2$ rotated position, columns and rows will automatically be swapped in order to allow for correlations in the Online Monitor.

i2c: This is an optional parameter for specifying non-standard (non-0) I^2C addresses the devices are listening on. This parameter takes a vector of integers which allows to run more than one ROC attached to a single DTB such as token-chained ROCs in beam telescopes, or full CMS Pixel modules. For this, the I^2C address of every ROC has to be specified, e.g.

```
i2c = 0 1 2 4 5 6
```

would set up the DUT in a way that six ROCs are programmed (using six DAC and trim bit files) and read out. If no I^2C parameter is specified, the I^2C address defaults to 0 and the `dacFile` and `trimFile` parameters are assumed to represent the full path and name of the files to be read. If the I^2C parameter is specified (even for a single ROC) the file names will be appended with the pattern $\_Cx$ where x is the I^2C address from this parameter. This is possible for both single chips and multiple ROCs.

dacFile: standard formatted pxar configuration file containing all DAC parameters for the ROC. These values will be provided to the `pxarCore::initDUT(...)` function. All possible DAC names can be found in Table 5. If the `i2c` parameter is specified, the name will be appended with the trailing $\_Cx$ pattern automatically.

tbmFile: standard formatted pxar configuration file containing all register parameters for one TBM core. These values will be provided to the `pxarCore::initDUT(...)` function. All possible register names can be found in Table 4. The file name will be appended with the trailing $\_C0a/b$ pattern for the two cores.

trimFile: standard formatted pxar configuration file containing the trim bits for all 4160 pixels of the ROC. These values will be provided to the `pxarCore::initDUT(...)` function. If the `i2c` parameter is specified, the name will be appended with the trailing $\_Cx$ pattern automatically.

maskFile: global, standard formatted pxar configuration file containing a list of masked pixels for all ROCs attached. Pixels which appear in this list will be masked during data acquisition. Pixels have to be specified with the pattern `pix ROC COL ROW`, multiple consecutive pixels in the same column can be masked using `pix ROC COL ROW1:ROW2`.

external_clock: Boolean switch to select running on externally supplied clock (via the DTB LEMO connector, see Section 3.1) or the DTB-internally generated clock. This calls the function `pxarCore::setExternalClock(bool enable)`. If no external clock is present but requested, the producer will return with an error requiring reconfiguration.

trigger_source: string literal to select the trigger source to be set up in the DTB. It is also possible to activate more than one trigger source by concatenating them using the semicolon `;` as separator. The trigger sources are configured via the API function `pxarCore::daqTriggerSource(std::string src)`. A full list of available trigger sources and their description can be found in Table 6.

usbId: USB identification string of the DTB to be connected. It is recommended to always specify this parameter instead of supplying the wildcard *. This is needed in the pxarCore constructor.

hubid: Hub address the DUT is attached to. See Section 5.1.2 for more information. This value is required by `pxarCore::initDUT(...)`.

signalprobe_d1/d2/a1/a2: Expects a string literal as value. Allows setting the DTB LEMO outputs to the selected signal. For more information see Section 3.1. The default setting (no parameter given) is `off`. A full list of available signal outputs it given in Table 7.

vd/va/id/ia: Mandatory parameters representing the DTB current limits (`ia`,`id`) and the supply voltage for the attached DUT (`va`,`vd`). These parameters are passed to the function `pxarCore::initTestboard(...)` at the initial configuration stage. More detailed information can be found in Section 5.1.1.

clk/ctr/sda/tin: Phase settings for DTB output signals. A more detailed description is given in Section 5.1.1.

level: Signal gain of the above signals, with 0 being off and 15 being maximum gain. Section 5.1.1 provides more information on these configuration parameters.

deser160phase: Relative phase of the 160 MHz deserializer module clock to the 40 MHz clock. More information can be found in Section 3.2.

triggerlatency: Additional delay for the trigger to match the overall trigger latency including WBC. This setting can be used to match the actual trigger latency from the trigger logic unit (TLU) and cabling to the ROC's WBC setting and thus allows to run with different WBC settings.

tindelay: ADC DAQ only (analog PSI46v2 chips): delay for the ADC to start sampling the incoming data after the token in has been sent out. For more information see Section 5.11.

toutdelay: ADC DAQ only (analog PSI46v2 chips): delay for the ADC to stop sampling the incoming data after the token out from the DUT has been received back. For more information see Section 5.11.

verbosity: Verbosity setting of the pxarCore library. All verbosity levels and their outputs are described in Section 5.8.

tlu_waiting_time: Additional waiting time in Milliseconds between stopping a run and stopping the pxarCore DAQ. This is required since the TLU is comparatively slow in executing the Run Stop signal and keeps on sending triggers. If this setting is too low, the number of triggers from the various devices will not match since some triggers have not been received at the end of the run. The default setting is 4000 ms.

testpulses: Boolean parameter which allows to run the DUT with testpulses and the pattern generator instead of external triggers. The `trigger_source` parameter has to be set accordingly. In addition, the delays in 40 MHz clock cycles after the different signals contained in the pattern generator setup can be changed using the parameters `resetroc/calibrate/trigger/token`. The overall delays between two calls of the pattern generator (so the rough overall trigger frequency) can be adjusted using the `patternDelay` parameter. The testpulses functionality is mainly intended for software development with no actual particle beam present. More information on the pattern generator can be found in Section 3.4.

rocresetperiod: Parameter allowing to send a periodic reset signal to the ROC. The value is given in Milliseconds, a value of 0 turns the periodic reset off. Internally this uses the *direct signal* trigger mode, switching on this source in addition to the one selected by the `trigger_source` parameter. However, there is no guarantee that the two signals will not coincide, and some triggers might get lost while sending the reset. This leads to a loss of synchronization between the DUT and other detectors in the DAQ. This feature should be used with caution and only when really necessary!

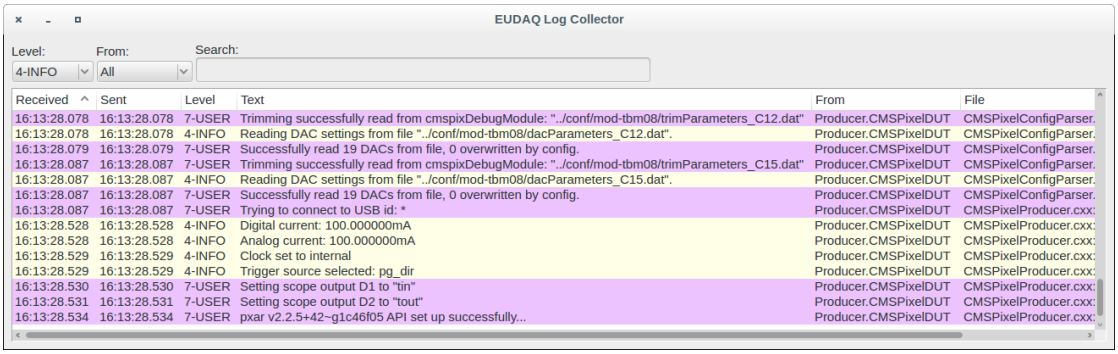


Figure 15: EUDAQ Log Collector with messages from an attached CMSPixelProducer instance. The log messages show the configuration stage of a full module using the DTB emulator (see Section 4.6).

In addition to the above parameters, it is also possible to overwrite DAC parameters. This is useful in cases when several DAC files (e.g. with different threshold settings) have been prepared in the laboratory beforehand, but the WBC is only known at time of the test beam. Instead of changing this parameter in all DAC files which is error prone and cumbersome, the DAC in question can be set as parameter in the EUDAQ configuration file. A list of all possible DAC parameters can be found in Table 5.

To overwrite a DAC parameter, its name and the desired value have to be specified in lower case in the configuration file, e.g.

```
wbc = 186
```

First, all parameters from the DAC file are read in, and then their values are updated and potentially overwritten by settings from the global configuration file. It has to be noted that only DACs present in the DAC file will be updated, while DACs missing from the file will not be taken into account even if specified in the configuration file. In case a DAC parameter has been overwritten by a configuration file setting, this will be noted in the EUDAQ log file.

8.6. Logging

The CMSPixelProducer connects to the EUDAQ Log Collector and informs about events at configuration and running stage as shown in Figure 15. It informs about parameters read from file, about changes made to the DTB configuration and also prints analog and digital currents after start-up.

In addition, the event yield for every detector is calculated as a total average and as running average of the last 1000 triggers. The event yield is the fraction of events that contain at least one pixel hit over the total number of events recorded.

At the end of a run, the total event yield is sent to the Log Collector and thus stored in the EUDAQ log file for later reference.

8.7. The CMSPixelConverterPlugin

Within the EUDAQ framework, a so-called converter plugin should be provided for every event type. This allows EUDAQ to decode the stored raw detector data for various purposes, be it online monitoring or conversion into another format for offline reconstruction.

Since several CMSPixelProducer instances can be operated at the same time, also several decoder plugins for every event type written have to be provided. In order not to duplicate code the current solution implements the decoding in the file

`main/include/eudaq/CMSPixelHelper.h`

and provides wrapper plugins for the various event types supported (cf. Section 8.4). Internally the decoder modules of the pxarCore library are used and thus all features such as decoding statistics are available (cf. Section 5.10). The accumulated decoding statistics are printed to *stdout* when the CMSPixelConverterPlugin receives an end-of-run event (EORE).

All parameters necessary for correct data decoding such as the ROC types, the TBM type in use, and the number of ROCs to be expected, are read from the tags written into the BORE event by the CMSPixelProducer.

8.7.1. Decoding to StandardEvents

EUDAQ features an internal even class called *StandardEvent*. This event format assumes pixelated detector planes of a certain size, and just stores the hit information in X and Y coordinates without any additional information. This very basic class is mostly used for internal purposes such as online data quality monitoring.

8.7.2. Decoding to LCIO Events

For offline analysis and reconstruction of telescope test beam data the EUTelescope software package [28, 29] is available and features a close integration of the EUDAQ software framework. EUTelescope is based on the ILCSoft framework [30] which provides the basic building blocks for offline analysis such as a generic data model (Linear Collider I/O, LCIO), a geometry description language (GEAR) and the central event processor (Marlin) [31].

The CMSPixel converter plugin provides a built-in function to convert PSI46 data to LCIO events. This function is used by the EUDAQ native reader library which is invoked by the EUTelescope NativeReader ("Converter") processor.

8.7.3. Other Data Formats

EUDAQ provides a set of data formats to which data can be converted using the *Converter* tool. This currently includes ROOT trees and text files, among other options. The options available can be checked by calling

```
./Converter.exe -h
```

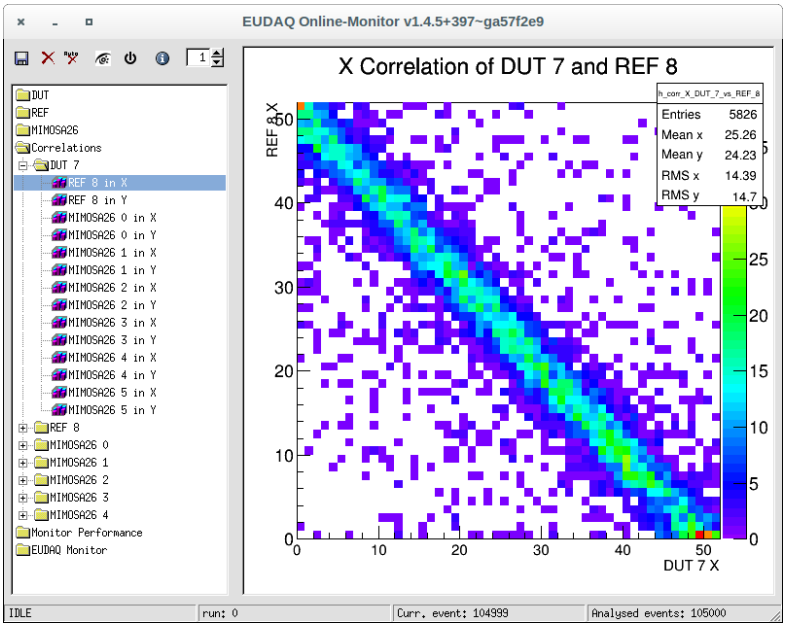


Figure 16: Correlation plot for the X coordinate of the DUT and reference plane. A distinct correlation pattern is visible indicating that the events read from the two devices are synchronized and the detectors are geometrically well aligned.

The selection of available output types depends on the build configuration of the EUDAQ library and some output formats might require additional libraries.

8.8. Online Monitoring

Online data quality monitoring is a key feature of data acquisition frameworks since it allows to immediately check the quality of the data recorded and check for possible glitches such as geometrical misalignment or missing time correlation.

EUDAQ provides the Online Monitor tool which displays a set of ROOT histograms. This includes simple hit maps and basic clustering plots for every detector plane included in the data acquisition as well as correlation plots in X and Y coordinates between all detector planes as shown in Figure 16. This allows to check for both time and geometry correlations and possible corrections directly during data taking.

The Online Monitor runs as separate process, connecting to the Data Collector of EUDAQ via the Run Control. By default it requests every 100th event for decoding in order to keep the load on the network and the CPU low. The tool can also be used to quickly check the content of an already stored file by executing it with the file argument:

```
./OnlineMonitor.exe -f path/to/raw/file.raw
```

The raw detector data is decoded using the StandardEvent conversion method of the CMSPixelConverterPlugin.

9. Summary

The pxarCore library provides a flexible and powerful interface to the digital test board (DTB) and allows data acquisition (DAQ) and testing of various PSI46-type detectors. The library has been developed as the primary DAQ software to be used in the module qualification of the Phase I Upgrade of the Compact Muon Solenoid (CMS) Pixel Detector and is used by all collaborating institutes for the quality control and functionality tests of produced detector modules.

This document provides a detailed description of the software architecture, the readout electronics, and the intended usage scenarios. Code examples for several common tasks as well as an exhaustive reference of possible configuration parameters and their influence on the detector tests are provided.

The software library has been developed with an emphasis on flexibility, maintainability and cross-platform operating system support in order to provide a reliable test bench for the detector modules over the full lifetime of the Phase I Pixel Detector of the CMS experiment.

10. Acknowledgments

We would like to acknowledge the contribution of Pirmin Berger, Martino Dall'Osso, Philipp Eller, Wolfram Erdmann, Andrew Hart, Satoshi Hasegawa, Jan Hoss, Urs Langenegger, Niklas Mohr, Daniel Pitzl, Robert Stringer and Marco Verzocchi.

Thank you for the fruitful discussions, code contributions, and long testing and debugging sessions.

A. Dictionary Reference

The following pages contain tables referencing the pxarCore dictionaries. All parameters are provided together with some explanation on the scope or the expected effect of the setting. Tables 1 and 2 list the available devices for Token Bit Managers (TBMs) and readout chips (ROCs) respectively. Tables 3, 4 and 5 list the available registers and digital-to-analog converters (DACs) for the DTB, the TBMs and ROCs.

A list of available trigger sources is given in Table 6. Examples on how to activate and use these sources has been provided in Section 7.1. The possible signal outputs of the DTB LEMO connectors are listed in Table 7 and the available signals for the pattern generator are summarized in Table 8.

Table 1: TBM device types: All names and synonyms understood by pxarCore and a short description.

Device Type	Description
tbmemulator	TBM emulator of the DTB FPGA, to be used in test beams with single ROCs. Header and trailer contain additional information such as the trigger arrival phase.
tbm08 = tbm08a	Initial prototype version of the Layer 3 and 4 TBM.
tbm08b	Intermediate prototype TBM version.
tbm08c	Production version of the TBM for modules in layer 3-4.
tbm09	Initial prototype of the dual-datalink TBM for layer 2.
tbm09c	Production version of the dual-datalink TBM for layer 2.
tbm10	Production version of the dual-datalink TBM for layer 1.

Table 2: ROC device types: All names and synonyms understood by pxarCore and a short description. All ROC versions except for psi46v2, psi46digv21respin and psi46digl1 are obsolete prototypes.

Device Type	Description
psi46v2	Chip with analog levels readout, as used in the Phase 0 pixel detector.
psi46xdb	As above, but with extended number of DCol buffer cells (time stamps, data).
psi46dig	First chip with digital 160 MHz readout.
psi46dig_trig	Interim version of the digital chip.
psi46digv2_b	Interim version of the digital chip.
psi46digv2	Version 2 of the PSI46 chip with digital 160 MHz readout. Problems with Double Column freezing.
psi46digv21 = psi46digv2.1	Interim version of the digital chip with reworked Double Column logic, but minor problem on Metal layer 1.
psi46digv21respin	Final production version of the PSI46 digital chip for the Phase I pixel detector. Wafer print is still `psi46digv21`.
proc600 = psi46digl1 = psi46digplus	Layer 1 version of the PSI46 digital chip for the Phase I pixel detector.

Table 3: DTB Registers: All names and synonyms understood by pxarCore, their register equivalent in the DTB (if applicable) and a short description.

Register Name	Register	Description
clk	0x0	Phase of the 40 MHz clock.
ctr	0x1	Phase of the Calibrate Trigger Reset (CTR) signal.
sda	0x2	Relative phase of the I2C SData signal.
tin	0x3	Relative phase of the Token In signal to the detector.
tout = rda	0x4	Same setting, two possible terminations depending on the DUT (TBM present or not).
level		Signal gain of the DTB output signals to the DUT (clk, ctr etc.). Range from 0 (off) to 15 (max. gain).
deser160phase		Relative phase of the 160 MHz deserializer module, see Section 3.2.
deser400rate		Sampling rate of the DESER400 phase detector, see Section 3.2.
triggerdelay		Additional delay between two pattern generator iterations in the trigger loops (see Section 3.3.2). Given in units of 10 bunch crossing (BC).
trimdelay		Additional delay after trimming of a pixel in the trigger loops (see Section 3.3.2). Given in units of 10 BC. Useful for heavily irradiated ROCs which require more time for programming operations.
triggerlatency		Additional trigger latency. After receiving an external trigger, this delay in units of 25 ns will be obeyed by the DTB before the trigger is sent out to the DUT. This can be used to compensate for different WBC settings.
triggertimeout		Timeout after sending an external trigger without receiving the Token Out signal back, after which the readout is terminated.
tindelay		Delay before the ADC starts sampling after the Token In signal has been sent (analog ROCs only).
toutdelay		Delay before the ADC stops sampling after the Token Out is received back from the DUT (analog ROCs only).
adctimeout		Timeout after which the ADC stops sampling if no Token Out signal is received (analog ROCs only). Given in units of 10 BC

Table 4: TBM Registers: All names and synonyms understood by pxarCore, their register equivalent (if applicable) and a short description. In the register address, X is either e or f for the two TBM cores. A complete overview of the registers can be found in [17].

Register Name	Register	Description
base0 = counters	0xX0	Enable/disable PKAM, Auto reset, triggers etc.
base2 = mode	0xX2	TBM mode setting.
base4 = clear = inject	0xX4	Clear stack/trigger counter, inject signals.
base8 = pkam_set	0xX8	Set the PKAM counter.
basea = delays	0xXa	Set token and ROC data delays.
basec = autoreset	0xXc	Configure the auto reset setting.
basee = cores	0xXe	TBM α: Adjust the 160 MHz and 400 MHz phases.
nrocs = nrocs1		Token chain length of TBM channel A.
nrocs2		Token chain length of TBM channel B.

Table 5: ROC DACs: All names and synonyms understood by pxarCore, their register equivalent and a short description. Information compiled from [19, 20].

DAC Name	Register	Description
vdig = vdd	0x01	Regulated digital supply voltage 1600 mV - 2400 mV, depends on the unregulated digital supply voltage.
vana = iana	0x02	Analog current 6 mA - 65 mA.
vsf = vsh	0x03	Current regulator for sample+hold circuit.
vcomp	0x04	Regulated comparator supply voltage.
vwllpr = fbpre	0x07	Preamplifier feedback.
vwllsh = fbsh	0x09	Shaper feedback.
vhlddel = holddel	0x0a	Sample+hold delay, 255 is shortest.
vtrim = trimscale	0x0b	Global trim scale.
vthrcomp = globalthr	0x0c	Global threshold (inverse).
vibias_bus	0x0d	Digital bus receiver.
voffsetro = voffsetr0 = phoffset	0x11	Pulse-height offset.
vcomp_adc = vibias_ph = adcpower	0x13	ADC comparator voltage (for digital ROCs).
viref_adc = vibias_dac = ibias_dac = phscale	0x14	Pulse-height scale.
vicolor	0x16	Column-Or current limit.
vcal	0x19	Calibration pulse height.
caldel	0x1a	Calibration pulse delay, 0.4 ns per DAC.
ctrlreg = ccr	0xfd	Bits 0: unused, 1: disable ROC, 2: Vcal Range.
wbc	0xfe	Trigger latency, requires ROC reset to take effect.
readback = rbreg	0xff	Readback mode/register.
vbias_sf	0x0e	up to psi46digv2 only.
voffsetop	0x0f	up to psi46digv2 only.
vion	0x12	up to psi46digv2 only.
vleak_comp = vleak	0x05	up to psi46dig only.
vrgpr	0x06	up to psi46dig only.
vrgsh	0x08	up to psi46dig only.
vibiasop = vibias_op	0x10	up to psi46dig only.
vibias_roc	0x15	up to psi46dig only.
vnpix	0x17	up to psi46xdb only, Pixel sensitivity for multiplicity trigger.
vsumcol	0x18	up to psi46xdb only, DCol sensitivity for multiplicity trigger.
vrangetemp	0x1b	up to psi46xdb only, temperature range for on-chip sensor. To be set last in order to get temperature readings via lastDAC (see Section 5.9).

Table 6: Possible DTB trigger sources and a short description. All trigger signals can either be sent to the detector directly or via the internal TBM emulator.

Trigger name	Description
none	Turns off all trigger sources.
Asynchronous external triggers	
async = extern	External trigger received via the "Trig" LEMO port of the DTB. Will be passed to the TBM emulator.
async_dir = extern_dir	External trigger received via the "Trig" LEMO port of the DTB. Will be passed directly to the detector.
async_pg = extern_pg	External trigger received via the "Trig" LEMO port of the DTB. This trigger will start one pattern generator cycle. The input pulse has to be shorter that the total cycle length of the configured pattern generator. Will be passed directly to the detector.
Synchronous external triggers	
sync	Synchronous external trigger received via the "Trig" LEMO port of the DTB. Will be passed to the TBM emulator.
sync_dir	Synchronous external trigger received via the "Trig" LEMO port of the DTB. Will be passed directly to the detector.
Single event injection	
single	Single signal sent to the TBM emulator.
single_dir = single_direct	Single signal sent directly to the detector.
Internal Trigger Generator	
random	Generates an internal random trigger with configurable frequency.
random_dir	Generates an internal random trigger with configurable frequency, sent directly to the detector.
periodic	Generates an internal periodic trigger with configurable frequency.
periodic_dir	Generates an internal periodic trigger with configurable frequency, sent directly to the detector.
Pattern Generator	
pg = patterngenerator	Loops through the configured patterns of the pattern generator. All produced signals are sent to the TBM emulator.
pg_dir = pg_direct	Loops through the configured patterns of the pattern generator. All produced signals are sent directly to the detector. This is the default setting at start-up and for all tests.
chain	Not supported currently.
sync_out	Activate the synchronous trigger output for daisy-chaining several DTBs. Not supported currently.

Table 7: Signal Probes: possible signals for the DTB LEMO scope outputs. "A" marks the availability at the differential analog scope outputs, while "D" states the possibility to route the signal to the digital scope outputs. Signals marked with [X] allow selection of the TBM data channel to be probed (0-7), default is channel 0.

Signal name	A	D		Description
tin	✓	✓		Token In signal to the detector.
ctr	✓	✓		CalTrigReset signal to the detector.
clk	✓	✓		Clock sent to the detector.
sda	✓	✓		SDA signal for I2C programming.
tout	✓	✓		Token Out received from the detector.
rda	✓	✓		RDA I2C return signal.
off	✓	✓		no signal.
sdasend		✓		Send signal for I2C SDA.
pgtok		✓		Pattern Generator Token is generated.
pgtrg		✓		Pattern Generator Trigger is generated.
pgcal		✓		Pattern Generator Calibrate is generated.
pgresr = pgresroc		✓		Pattern Generator ROC Reset is generated.
pgrest = pgrestbm		✓		Pattern Generator TBM Reset is generated.
pgsync		✓		Pattern Generator Sync signal is generated.
clkp = clkpresent		✓		External clock is present (high) or not (low).
clkg = clkgood		✓		External clock is good (high) or not (low).
daq0wr		✓		Write signal of DAQ channel 0 (ADC/DESER160).
crc		✓		Internal FPGA CRC check of the firmware (inactive).
adcrunning		✓		ADC is recording data (in DAQ).
adcrun		✓		ADC is enabled and can be started.
adcpgate		✓		Stop signal for ADC (e.g. token out signal received)
adcstart		✓		Same as adcrun but delayed by the start delay.
adcsgate		✓		Start signal for ADC, gated with adcrun.
adcs		✓		ADC start signal (e.g. token in signal received).
ds_gate		✓		DESER400 gate signal.
deser_frameerror		✓	[X]	DESER400 error: failed synchronization.
deser_codeerror		✓	[X]	DESER400 error: invalid 5 bit word received.
deser_error		✓	[X]	DESER400 errors encountered.
deser_header		✓	[X]	DESER400 event header detected.
deser_packet		✓	[X]	DESER400 data packet received.
deser_tbmhdr		✓	[X]	DESER400: TBM header detected.
deser_rochdr		✓	[X]	DESER400: ROC header detected.
deser_tbmtrl		✓	[X]	DESER400: TBM trailer detected.
deser_idle_error		✓	[X]	DESER400 error: no TBM trailer received.
deser_header_error		✓	[X]	DESER400 error: only TBM header received.
sdata1	✓			Differential input signal, line 1.
sdata2	✓			Differential input signal, line 2.

Table 8: Possible Pattern Generator signals and a short description.

Signal name	Description
none = empty = delay	No signal. This can be used to add additional delays in the generator pattern.
pg_tok = tok = token	Sends a readout token out. This should only be used when sending the pattern generator signals directly to the device (see Table 6).
trigger = pg_trg = trg	Sends a L1 trigger signal out.
calibrate = pg_cal = cal	Send a calibrate pulse to the DUT.
resetroc = pg_resr = resr	Sends a Reset signal to all attached ROCs.
resettbm = pg_rest = rest	Sends a Reset signal to all TBMs.
sync = pg_sync	Synchronization signal for DTB daisy chaining, currently unused.

References

[1] CMS Collaboration, "CMS Technical Design Report for the Pixel Detector Upgrade", Technical Report CERN-LHCC-2012-016. CMS-TDR-11, CERN, Geneva, Sep, 2012.

[2] M. Minano Moya, "CMS Pixel Upgrade for the Phase I: Module Production and Qualification", in *Proceedings of the 10th International "Hiroshima" Symposium.* Xi'an, China, 2015. CMS-CR-2015-263.

[3] J. Hoß, N. Mohr, P. Eller, and S. Spannagel, "pyXar - Python Test Routines for PSI46 Pixel Chips using the pxarCore Python Bindings", 2014. Retrieved March 31, 2015, from `https://github.com/simonspa/pyxar`.

[4] U. Langenegger and S. Spannagel, "pxar/pxarCore Releases", 2015. Retrieved March 31, 2015, from `https://github.com/psi46/pxar/releases`.

[5] Kitware, "CMake". Retrieved March 31, 2015, from `http://www.cmake.org/cmake/resources/software.html`.

[6] FTDI Ltd., "FTDI D2XX Drivers", 2014. Retrieved March 31, 2015, from `http://www.ftdichip.com/Drivers/D2XX.htm`.

[7] Cython Developers, "Cython – C-Extensions for Python". Retrieved January 11, 2016, from `http://cython.org/`.

[8] NumPy Developers, "NumPy - Scientific Computing with Python". Retrieved January 11, 2016, from `http://www.numpy.org/`.

[9] R. Brun and F. Rademakers, "ROOT – An object oriented data analysis framework", *Nucl. Instr. Meth. Phys. A* **389** (1997), no. 1–2, 81 – 86, `doi:10.1016/S0168-9002(97)00048-X`. New Computing Techniques in Physics Research V.

[10] Git Developers, "Git – free and open source distributed version control system". Retrieved January 15, 2016, from `https://git-scm.com/`.

[11] U. Langenegger and S. Spannagel, "pxar/pxarCore Issue Tracker", 2014. Retrieved March 31, 2015, from `https://github.com/psi46/pxar/issues`.

[12] B. Meier, "Firmware Flash Files for the DTB", 2014. Retrieved March 31, 2015, from `https://github.com/psi46/pixel-dtb-firmware/tree/master/FLASH`.

[13] Altera Corp., "Nios II Classic Processor Reference Guide", 2015. `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/nios2/n2cpu_nii5v1.pdf`. Accessed: 18.11.2015.

[14] B. Meier and S. Spannagel, "RPC Interface Generator Tool", 2013. Retrieved March 31, 2015, from `https://github.com/psi46/tools/rpcgen`.

[15] S. Spannagel and U. Langenegger, "pxar API Doxygen Documentation", 2015. Retrieved July 20, 2015, from https://psi46.github.io/annotated.html.

[16] H. Perrey and S. Spannagel, "pxar API Header", 2015. Retrieved July 20, 2015, from https://github.com/psi46/pxar/blob/master/core/api/api.h.

[17] E. Bartz, "TBM08c Documentation", 2015, https://cms-docdb.cern.ch/cgi-bin/DocDB/ShowDocument?docid=12626. Retrieved July 20, 2015, from https://cms-docdb.cern.ch/cgi-bin/DocDB/ShowDocument?docid=12626.

[18] J. Hoss (ETH Zurich), "Private communication", 2015.

[19] W. Erdmann, "PSI46digV2.1 Reference (Dec 2013 submission)", 2014. Retrieved March 30, 2015, from http://cms.web.psi.ch/phase1/psi46dig/index.html.

[20] W. Erdmann, "PSI46digV2 Reference", 2014. Retrieved March 30, 2015, from http://cms.web.psi.ch/phase1/psi46dig/psi46digv2.html.

[21] S. Spannagel, "A High-Rate Beam Test for the CMS Pixel Detector Phase I Upgrade", Master's thesis, Karlsruhe Institute of Technology, 2012.

[22] M. Reichmann (ETH Zurich), "Private communication", 2015.

[23] The EUDAQ Development Team, "EUDAQ Software User Manual", 2014. Retrieved April 11, 2015, from http://eudaq.github.io/manual/EUDAQUserManual.pdf.

[24] The EUDET Collaboration, "EUDET: Detector R&D towards the International Linear Collider Final Report", Technical Report EUDET, 2011. Retreived April 14, 2015, from http://www.eudet.org/content/e116147/Final_Report_final_vers.pdf.

[25] A. Bulgheroni et al., "EUTelescope: Tracking Software", *EUDET-Memo-2007-20* (2007).

[26] S. Spannagel, "CMSPixelProducer Development Repository", 2014. Retrieved April 16, 2015, from https://github.com/simonspa/eudaq.

[27] S. Spannagel, "Results from Testbeams",. CMS Pixel Workshop 2015, Visegrad, Hungary. Unpublished.

[28] I. Rubinskiy, "EUTelescope. Offline track reconstruction and DUT analysis software", Technical Report EUDET-Memo-2010-12, 2010.

[29] EUTelescope Software Developers, "EUTelescope Website". http://eutelescope.desy.de. Accessed: 03.09.2014.

[30] S. Aplin, J. Engels, and F. Gaede, "A production system for massive data processing in ILCSoft", Technical Report EUDET-Memo-2009-12, 2009.

[31] F. Gaede and J. Engels, “Marlin et al – A Software Framework for ILC detector R&D”, Technical Report EUDET-Report-2007-11, 2007.