Readout system for the ALPIDE pixel sensor

G.G. et. al.

May 12, 2021

Contents

Chapter 1

Introduction

This document aims to be a complete description of the firmware released in this repository, covering both the roles of "user manual" and developer's reference for further development.

The main objective of this work is the development of a simple and adaptable FPGA-based readout system for the ALPIDE pixel sensors[3] intended mainly for two experiments:

- The particle tracker that will be part of the HEPD2 detector.
- A small particle telescope for sensors characterization.

Nevertheless the firmware will be developed with a strong emphasis on modularity, to cope with sudden changes of the detector design and also ease its reuse in others potential applications of comparable complexity.

The firmware is targeting the Xilinx 7 series of FPGAs, trying to use the smallest amount of resources, requirements imposed by the nature of the HEPD project, and incidentally maintain compatibility a wide range of devices from the same series. On the other way, while it should be possible to use most Xilinx development boards with this firmware, both the ALPIDE carrier boards and the FPGA-Carrier interface electronics used in the development phase are based on custom-made PCBs. Again, for the sake of simplicity, most of such designs contain the bare minimum of components required to do the work, so they should be relatively easy to reproduce.

The following sections will concentrate on the more complex case of the HEPD detector, since the case of the ALPIDE telescope can be considered as a simplified version of the smallest unit that compose the full detector. Consequently any consideration discussed for HEPD can be trivially applied to that case as well.

1.1 The High Energy Particle Detector

The High Energy Particle Detector (HEPD) is a compact, high energy particle detector that will be used to detect particle trapped in the earth ionosphere. The system will include a particle tracker based on the ALPIDE pixel sensors developed for the ALICE experiment at LHC. A detailed discussion of HEPD is beyond the scope of this document and so the information presented in this section will be limited at the description of the detector topology and its consequences and challenges to the realization of a readout system.

The ALPDIE chip is a CMOS pixel particle detector with binary output: the digitalized output of the sensor consist of a list of "hits" reporting the coordinates of the pixels that registered an over-threshold event. The sensors can work in a self-triggered mode for lower event rates, but in this work it is always used with an external trigger. Again, a detailed description of the ALPIDE sensor is omitted for brevity and only some basic informations relevant to this document will be reported. A detailed description of the chip and its functions can be found in the ALPIDE operation manual[1].

In a simplified view of the device, the communication with the sensor requires the use of 3 differential lines:

- DCLK: M-LVDS clock used by the digital part of the chip, the maximum design frequency is 40 MHz. This clock line can be shared between several chips leveraging the multi-drop architecture of the standard.
- DCTRL: M-LVDS bidirectional serial line synchronous to DCLK used for trigger, commands and monitoring of the chip. Several chips can share the same line according to the master-slave scheme and protocol described in the manual[1].
- HSDATA: LVDS high speed serial line with programmable speeds of 400/600/1200 MB/s used for data readout.

The DCTRL line also offers a side-channel to perform the readout of the sensors data, allowing the operation with only 2 differential lines albeit with a severe (and, for most applications, utterly crippling) reduction of performances; the effective data rate of the CTRL line available for pixel data readout is around 8 Mbit/s when running at 40 MHz, calculated without considering the bandwidth required for trigger and possibly commands to the chip. Given the tight power budget available for HEPD and the large power consumption of the PLL required for the high speed lines, this work is aiming to demonstrate an architecture capable of performing the readout

using the DCTRL line only, taking advantage of the low event rate expected by the experiment and a modular parallel readout architecture. While a great development effort is dedicated to this goal, the firmware will also include the possibility to use the high speed serial line mainly for use in the ALPIDE telescope.

As schematized in fig. 1.1, the basic block of the HEPD detector is a stave similar to the modules used to build the Outer Barrel layer of the ALICE inner tracker. As shown in figure, a single stave is composed by two master, sharing the same DCTLR and DCLK lines, each connected to 4 slaves. In this configuration the master chip forwards commands to the slaves on a local bus. Chips sharing the same bus are addressed by a *chip id*, which also determine the role of the chip in the bus, as described in depth in the ALPIDE manual[1]. To allow particle tracking, three staves are then piled to form a *tower*. Finally the complete detector is composed by five tower side-by-side.

In general, to cope with the low speed of the DCTRL line, each of the fifteen staves is read in parallel, but the five towers are treated as "indepen-

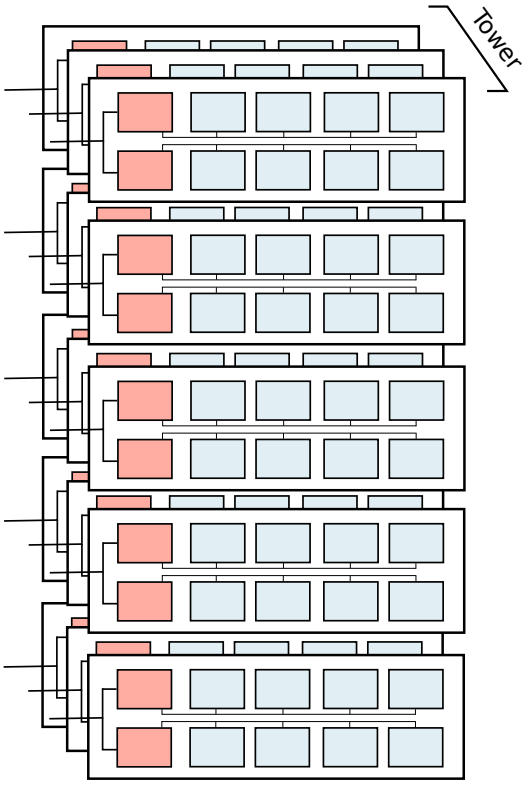


Figure 1.1: The complete detector tracker.

dent" elements with a dedicated trigger signal, so an event can be composed by data from a subset of towers. During time of inactivity, a tower is kept in a "standby" mode by disabling the clock line in order to save power.

Chapter 2

TDAQ firmware architecture

This section describes the firmware architecture necessary for the readout of the HEPD tracker. The discussion in divided in three parts, that provides a black-box functional model of the main firmware macro-blocks:

- The design of the core implementing a readout state machine interfacing with an ALPIDE master pair (a stave) over the control line.
- The dataflow architecture necessary to manage, pre-process and serialize the data stream from the entire set of 15 staves.
- The "command and control" system necessary for configuration, calibration, testing and communication with the rest of the DAQ.

A general overview of the system is illustrated in the block diagram in fig. 2.1. Each stave in is controlled by a "*CTRL*" font-end module implementing the CTRL line protocol and containing the readout finite state machine. Each module works independently from the others and has its own trigger input. The same trigger line is shared between staves part of the same turret. Readout data is inserted in a dedicated FIFO buffer, with data from different events (i.e. triggers) delimited by a special end-of-event flag. The data present in the FIFO buffers is then read by an "event builder" for serialization and packetization of each event to be sent to the main DAQ board.

The system also includes a Microcontroller Unit (MCU) module based on a soft-CPU core (Microblaze). This acts as "brain" of the system, performing calibration routines, general bookkeeping, and acting as general interface with the external word. This block will generally be employed during the configuration and diagnostic (telemetry gathering) of the system, with the readout data being served to the off-chip communication module directly without the need of intervention from the CPU.

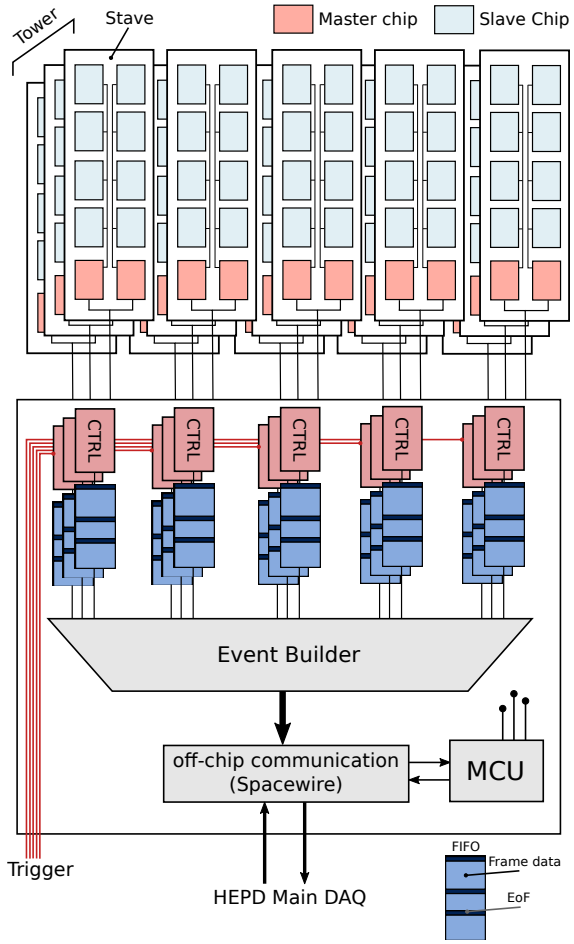


Figure 2.1: The readout pipeline of the HEPD tracker.

2.1 Data Readout

2.1.1 DCTRL line portocol

As already said, the DCTRL line is used both for data readout and sending commands to the ALPIDE chips, and implements a 3-state logic to allow bidirectional communication. According to the ALPIDE operation manual[1], the DCTRL line support three types of transactions, illustrated in fig. 2.2. Each transaction is composed by 10-bit words composed by a "start bit" (logic 0), 8-bits of data and a stop bit (logic 1). Multiple words part of the same transaction are separated by a "gap" (logic 1) of possible length between 0 (no gap) and 42 clock cycles. The 3 possible type of transactions are:

- A 1-word *command broadcast* to all the chip on the bus. This is mostly used for the trigger and reset commands.
- A 6-words *write register* command that can either target a single chip (unicast write) or be broadcasted to all chips (multicast write).
- A 4-words *read register* command, targeting a single chip, which is followed by 3-words answer containing the register data from the targeted chip. To avoid issues from transients when the control of the bus is changed, a 15 clock cycles bus turnaround "rest" period is defined where the bus master release the control of the bus and the slave acquire it.

In this implementation the FPGA does not generate any gap (gap size is 0) in order to maximize the transmission speed. The response of the chip is never longer than 65 clock cycles (including the two turnaround periods) and was observed to contain gaps of size between 0 and 2, but never longer.

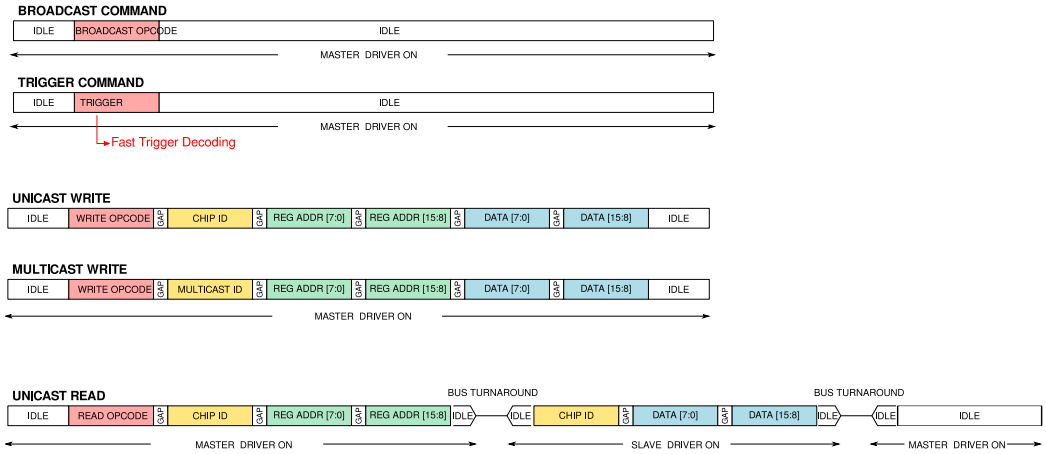


Figure 2.2: Data format over the DCTRL line

An important note regarding the reliability of this communication channel is that the protocol does not offer any form of error detection, with the only exception of the commands codes which were chosen by the chip designers to have a Hamming distance of 2. As a consequence there isn't a safe general solution to validate the received data against possible transmission errors. This is particularly troublesome in this application where, as will be discussed shortly, data framing informations are relevant for the operation of the readout state machine.

The diagram in fig. 2.3 describes the state machine implemented in the FPGA driving the DCTRL line. Transmission of the packet is executed by

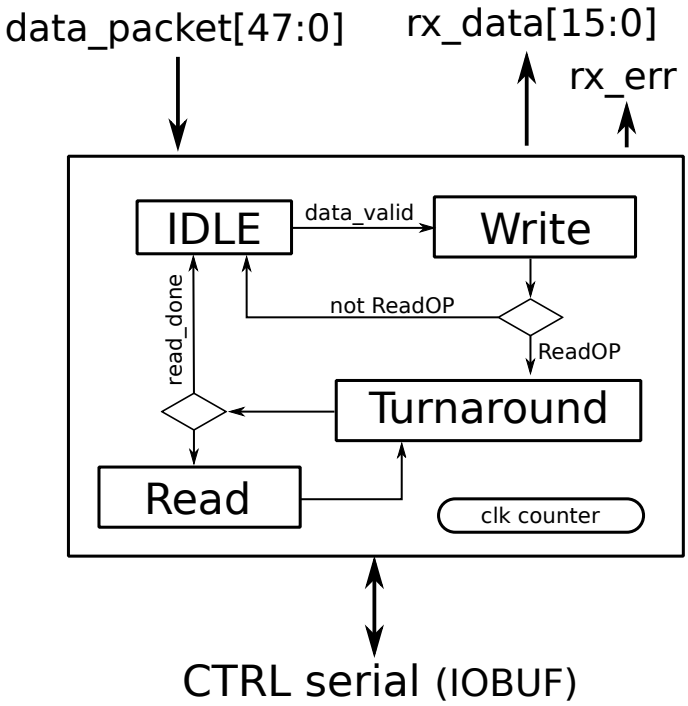


Figure 2.3: State machine implementing the DCTRL line master

means of a simple serializer that packs the data packet in a register (together with the stop and start bit for every 8-bit section of the packet) and pushes out a single bit at very clock cycle starting from the MSB.

The input accept a single packet to transmit over the control line, which according to fig. 2.2 is maximum 48 bits long. The state machine decodes the OPcode (first 8 bit of the input) to detect how many bits of the input are valid and must be transmitted (48 for a write operation, 4 for a read, 8 otherwise) and whether or not a read from the bus must be attempted (OPcode is RDOP). ALPIDE can recognize invalid OPcodes and will keep a counter with the errors, so a finer validation of the OPcode by this state machine is not necessary and does not improve the stability of the system.

The read operation will verify if the 3 8-bit words are received (by counting the stop/start bit pairs) and if the received chip id correspond with the transmitted id. The output of the operation are the received 16 data bits and a rx error signal in case of failure of the two previous conditions.

The entire state machine is governed by a clock counter that determine the maximum length of each state, so a deadlock condition is not possible (the counter always reach the maximum pushing the state machine in the next state until it back to idle).

2.1.2 Readout State Machine

The core managing the communication with the chips is based on a state machine designed to offer the required flexibility but also be as simple and lightweight as possible. A diagram of the state machine is shown in fig. 2.4; After power-on the system is in the IDLE state. In the IDLE state it is possible to send any ALPIDE packet to the DCTRL line using the *ALPIDE command input* interface composed by the following signals:

- `ALP_cmd [47:0]`: ALPIDE packet to send.
- `cmd_DV`: data on ALP_cmd is valid (command send request).
- `cmd_ack`: command acknowledge from state machine.
- `cmd_res [15:0]`: command response.
- `cmd_res_err`: read error.
- `cmd_res_DV`: data on cmd_res and cmd_res_err is valid.

To send a packet the master sets the packet data and asserts the signal `cmd_DV` to request the transmission of the packet. The master shall keep `cmd_DV` asserted until the state machine acknowledge the received command by asserting `cmd_ack`. After the packet send request is acknowledged, the master shall wait for the assertion of the `cmd_res_DV` signal to confirm that the command was successfully sent to the chip; any response to the command (i.e. command opcode was RDOP) is then available on the `cmd_res` bus. A `cmd_res_err` indicates if there was an error reading the command response from the chip, and when asserted any data on `cmd_res` should be considered invalid. If a command is acknowledged by the state machine, the `cmd_res_DV` will be always asserted once it completes the packet transmission, so for packets not generating a chip answer (i.e. anything except WROP) ignoring the `cmd_res_DV` signal is tolerated.

This interface is mainly intended to transmit the configuration data to the chips and other external commands (e.g. from the CPU) during test and calibration procedures. In fact the internal implementation is in some cases mostly a pass-thought of the input command to the core implementing the CTRL line protocol (described in the previous section). The actual readout procedure is executed internally by the state machine by consecutive read of the two registers associated to the chip DMU data FIFO.

The start of the readout procedure is initiated by the assertion of the dedicated `trigger` signal, or by the transmission of a `TRIGGER` or `PULSE` broadcast

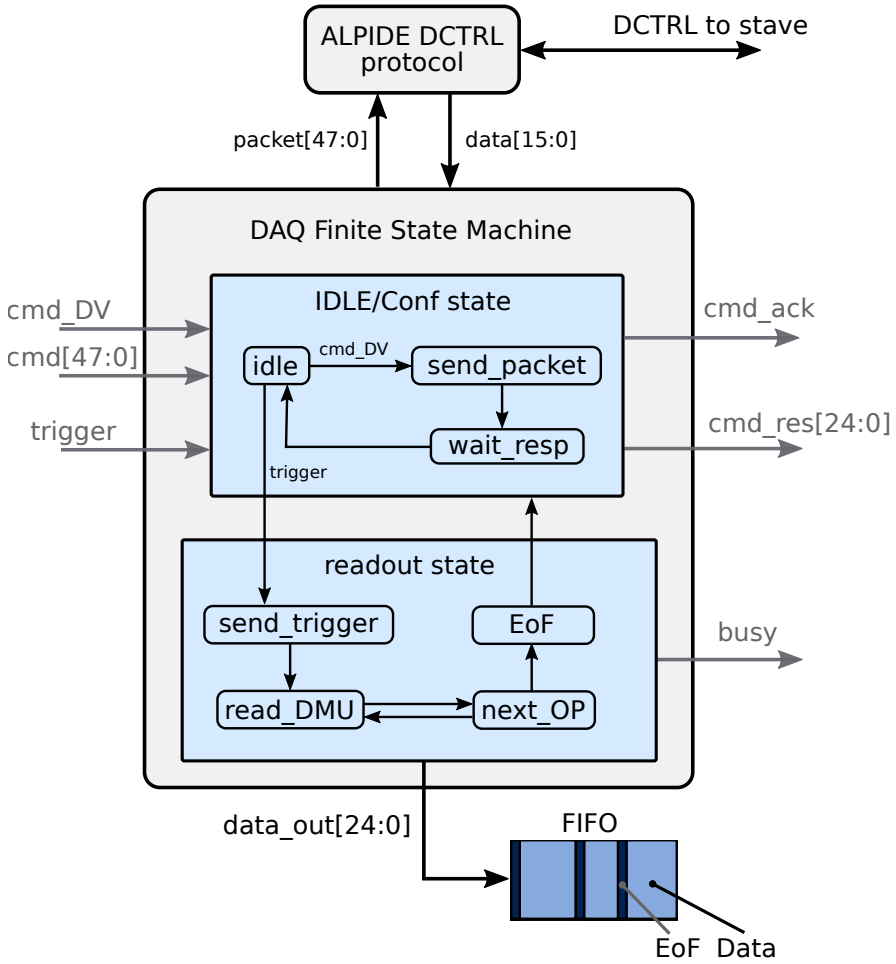


Figure 2.4: Finite state machine controlling the DCTRL line.

command over the command interface. As soon as the state machine detect the assertion of the trigger signal, the ALPIDE `TRIGGER` command is broadcasted over the DCTRL line and then the core starts continuously reading the two DMU data registers of each chip in the chain in sequential order; read requests are continuously sent to a chip until a trailer word is detected, then the system moves to the next chip until the entire chain is scanned, or in others words when the *chip_trailer* word of the last chip id is found.

The DMU data received is parsed to decode framing informations and status events, such as the *busy on/off* word, and any pixel data received is stored in a dedicated data FIFO buffer. A read operation from the control line always returns a 32-bit word, but depending on the word type only 1 to 3 octets contains useful information. To optimize the data stream, words are then serialized in a 8-bits stream before being saved in the data buffer.

In the serialization process invalid octets are discarded and the remaining are inserted in the buffer in big-endian order, which allows to easily decode the stream (an ALPIDE data word can be recognized by reading the most significant byte). Once the readout sequence is completed a event trailer is inserted in the buffer together with an *end-of-event* flag.

Any data received from the chips is stored in the FIFO with the exception of the *busy on/off* and *IDLE* words. The *chip_empty* word is also suppressed; the typical event is expected to produce few data words from a single chip per stave, and then a chip_empty word from the other 9 chips. So removing this word from the stream leads to a noticeable reduction in the event size. In it's place a chip empty bitmask is inserted in the event trailer (a detailed description of the data format is presented in the next section).

If the FIFO becomes full during the readout process, the FSM pauses its execution until the FIFO full flag is deasserted.

During the event readout operation the state machine will not accept another trigger until finished, and the busy signal is kept asserted as long it does not enter in the IDLE state. While this was not a necessary choice (ALPIDE can store up to 3 data frames) it makes the development much easier by avoiding event alignment issues, which are made especially dangerous by the lack of error detection (and correction) coding in the ALPIDE protocol; for example, during the readout of a chip storing 2 data frames from 2 different trigger, a SEU might corrupt a data framing word, such a chip trailer word, causing the state machine to start reading the successive frame immediately. The result would be an event packet with data from multiple event interleaved together.

In any case it is necessary to develop different safety measures to detect and manage anomalous conditions such as simple read errors and more general erroneous behaviour from the chips. There are two main failure conditions that might happen during readout: loss of important data (framing informations) and permanent chip damage. The state machine generally tries to recognize and validate every word received from the chip; an error counter is incremented every time an unexpected word or unrecognizable word is received and reseted when switching to another chip in the chain. If the error counter reach a certain threshold (5 consecutive errors by default), the chip is skipped and masked. The resulting chip mask is stored in a status register which is readable and writable from the CPU, which may then take some measure to restore the masked chip. This mask is also inserted in the event trailer together with the chip_empty mask. This procedure avoid deadlock conditions of the state machine, that can then continue the data acquisition process regardless of temporary or permanent issues on multiple chips.

Another safety measure keeps track of the event size (in term of number

of words received for a certain event) of each chip, truncating the event when a certain threshold, programmable from a control register, is trespassed.

2.2 Data Packaging

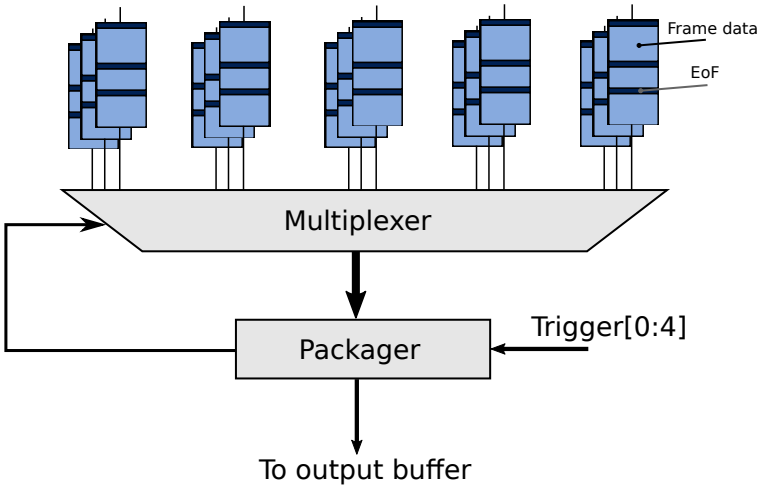


Figure 2.5: Data Multiplexer

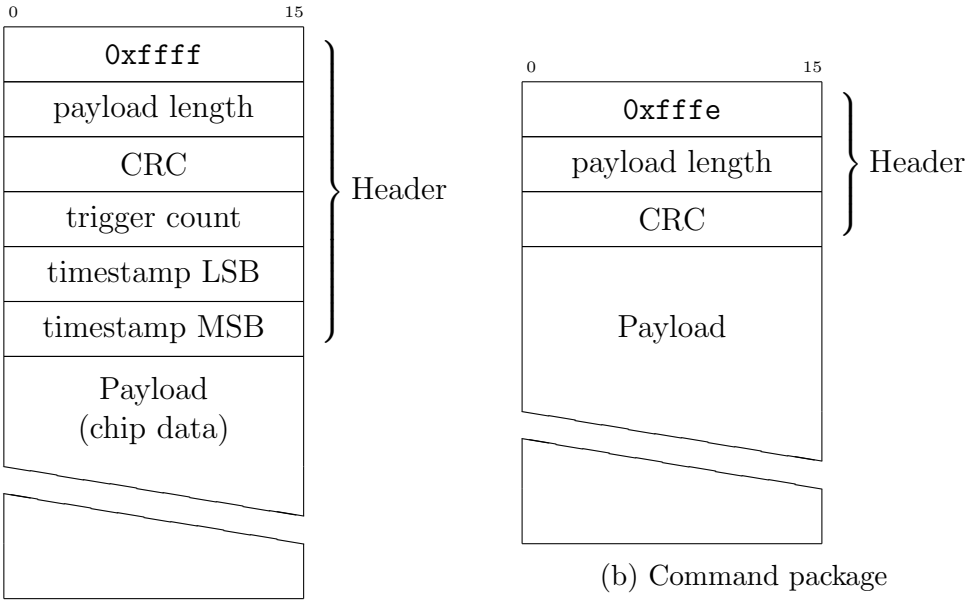
As said in the previous section, readout data is stored as a 8-bit big-endian stream in the dedicated stave buffers. Data is then packaged by the *packager* block in fig. 2.1, which takes care of building and transmitting data packages containing the entire event. Furthermore, any other information from the MCU (i.e. answer to commands) must be transmitted over the same output link but kept separate from event data. A more detailed view of the blocks composing this module is presented in fig. 2.5.

A *packager* finite state machine keeps track of the triggers received to determine which group of staves should be read for every event (in general it is expected that the stave buffers might contain data from multiple events).

The read start from the first selected buffer and continues until an *end-of-frame* flag is found and so the state machine moves to the next valid buffer. To distinguish between data from different staves a stave header word is inserted in the output data before each stave data.

When the system in not transmitting an event data package, the output multiplexer in figure is used to bypass the event FIFO and allow the transmission of data stored in the MCU output buffer.

A second multiplexer can be used to divert the readout data to a data decoder, which in turn write in a dual-ported RAM shared with the MCU,



(a) Data package

(b) Command package

Figure 2.6: output event package.

thus allowing the process to directly read the data form the chip; this function that is necessary for the calibration procedures, but could also be useful to preform an online analysis of real data. The implementation of a ALPIDE data decoder in the FPGA was not strictly necessary, but it greatly improves the performances on the system, especially during the calibration procedures.

2.2.1 Data Format

The event packet generated by the packager module is schematized in fig. 2.6; the package is composed by 16-bit words and has a well defined maximum length (determined by the maximum allowed event size on each stave). The packet header contains a first word to indicate the packet type (event data or data from MCU), payload length (byte count), and the CRC16-IBM (polynomial `0x8005`, final XOR `0xffff`) of the payload. In the case of an event data package other information useful for the event identification, such as the trigger counter, are inserted in the header as shown in the fig. 2.6. The byte order of the header is little-endian.

Readout data is packaged in an event packet by concatenation of the 8-bit words, and if necessary is padded with a single `0x80` byte for alignment with the packet boundary. If present, the padding byte is included in the packet length counter and CRC generation. In the concatenation process the

big-endian order of the data stream in maintained to allow the decoding of the variable-length stream of ALPIDE data words: ALPIDE words can have length between 1 and 3 bytes, with the word type (and so its length) fully determined by the most significant byte. The possible words that can appear in the data stream are listed in fig. 2.7. The chip header, region header, data (long and short) and chip header are directly produced by the chip and are the only alpide words that can be present in the data buffers. In addition the following words are generated and inserted to format the data stream:

- Stave header: precede data from the stave number stave_id.
- Stave trailer: mark the end of data from a stave, and contains the chip_mask (masked chip) and empty_map (bitmap of the chip that answered with a chip_empty word) registers.
- Truncated event: inserted when a chip is skipped during readout following and anomalous condition. This word might be inserted at any moment after a stave header.

An important note is that the data package contains only the last 4 bits of the chip id (stored in the CHIP HEADER) plus the stave number (in the STAVE HEADER) This is not a problem in the HEPD-02 tracker since the first 4 bits of the chip id are identical for all chips (`0x7`) so this information is redundant.

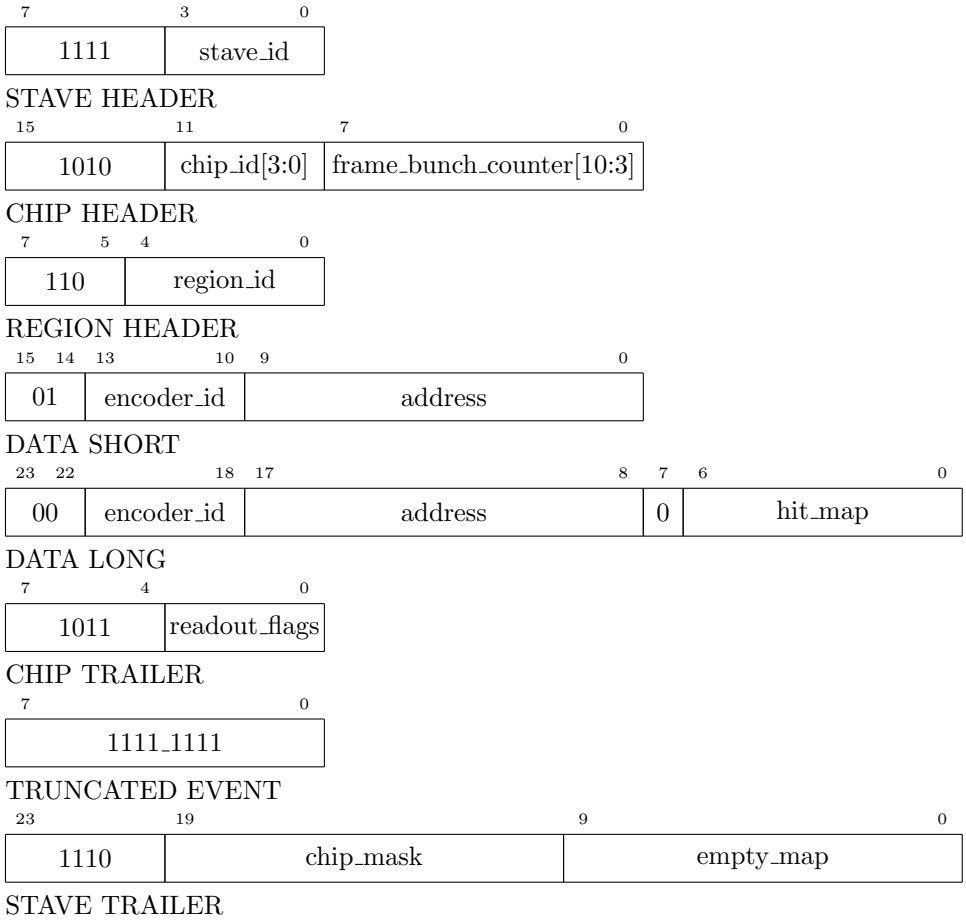


Figure 2.7: Event data words, listed in the same order they might appear in the data stream.

2.3 Micro-Controller Unit

The Micro-Controller Unit (MCU), based on the Microblaze soft-CPU IP core, is intended to perform calibration, diagnostic procedures and also gather data for telemetry (such as reading the many temperature sensors). The MCU communicates with the rest of the system by means of two channels: a two-way AXI-Stream FIFO used to receive and transmit data to the spacewire-based transmission module for communication with the DCPU, and a firmware control registers file mapped directly on the memory space of the CPU by an AXI-Lite bus.

The AXI control register firmware block, other that managing the write and read operation on the control registers, also produces the proper signals and handshake operations to control the others firmware blocks, such as

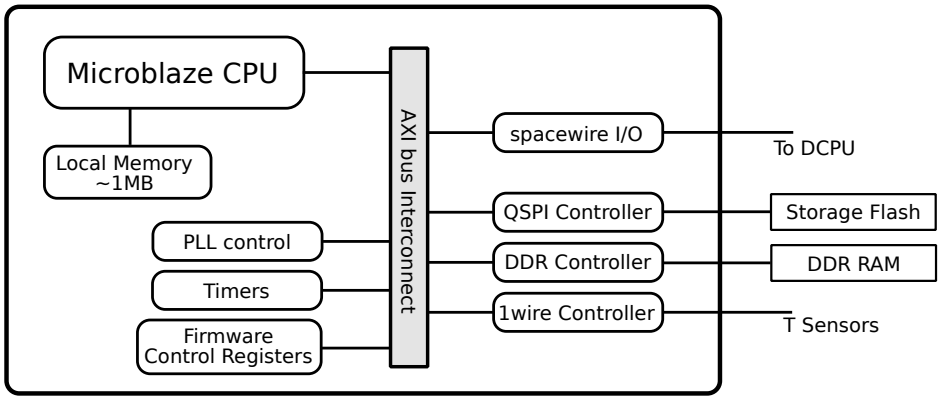


Figure 2.8: Block diagram of the MCU peripherals

example the control interface of the ALPIDE readout state machine.

The MCU has also access to a "storage" flash memory that is used to store the configuration parameters for the tracker along with any configuration data. As described in the TDAQ section, this is a separate flash chip from the one that contains the FPGA bitstream and the firmware for the MCU.

Finally the MCU has also control on the PLL used to generate the clock for the ALPIDE subsystem, this allows to set the clock speed for the sensor and implement a clock scaling algorithm.

2.3.1 Commands

Command for the MCU are received from the spacewire link and are packaged according to the "command package" format illustrated in fig. 2.6. Each packet contains in its payload a single command that is read and executed by the MCU in a FIFO order. A command is composed by a 8-bit "OP code", followed by any additional argument required for the execution of that command. A list of supported command, and their associated required arguments, is summarized in table 2.1.

The command interface is designed to always generate an answer to a command; In case of error decoding or executing a command, the MCU answer with the word `0xffffffff`. When a command is executed successfully the MCU answers with a package containing the command output or, for commands without an output, with the generic success word `0x00000000`.

2.3.2 Register Space

Todo

Command	OP code	Args	Answer
Echo	0x01	Any sequence of 32-bit words	Same sequence of 32-bit words
Chip Scan	0x02		Array of chip id found
Run Configuration	0x03		
Run Thresh Scan	0x04	[ChipId] [Mode]	
Run Digital Scan	0x14	[ChipId] [Mode]	
ALPIDE Command	0x05	[OPcode] [ChipId] [register] [data]	ALPIDE register data if OPcode = RDOP
Write FPGA Register	0x06	[Register] [Data]	
Read FPGA Register	0x15	[Register]	Register data
Apply pixmask	0x07		
Scan hot pixels	0x08	[Stave id] [threshold] [num of triggers]	
Write file	0x09	[File Index] [File len] [Mode] [Data]	
Dump file	0x10	[File Index]	
Format Flash	0x11		
List Files	0x12		null-terminated list of strings
FIFO self-test	0x13	[ChipId]	Number of errors

Table 2.1: Supported commands

2.4 SEU and Error Tolerance

Since the electronics will have to function in a radioactive environment, firmware and software need to be hardened against errors induced by the interaction of the radiation with the device. A charged particle depositing a certain amount of charge in a digital circuit might induces a state change (such as a bit-flip in a memory cell), that impacts the proper function of the device. Such error is called single-event upset (SEU). In a FPGA it is possible to distinguish between two different classes of errors caused by SEUs: errors in the configuration memory itself, and errors in the logic elements (flip-flops, memories, ...) part of the design implemented in the FPGA.

An SRAM FPGA, such as the Artix-7 used in the TDAQ board, implements the design logic mostly using SRAM LUTs and an interconnect structure based on switching matrices. A SEU striking any of those structure will case a change of the configuration of the device braking the underling logic. As consequence the design will malfunction or stop working properly until error in the FPGA configuration is fixed. There are several approaches of different complexity to mitigate this problem, and the proper solution depends or the required level of reliability requested (given a certain expected rate of SEU). The definition of reliability also depends on the particular case; some application can deal with a temporary incorrect output and a relatively long error recovery time, other might require a short recovery time, and others strictly require a correct (or at least "safe") output regardless of the recovery time.

The problem of protecting the FPGA configuration memory has been already studied in depth, and for this case Xilinx offers a commercial solution that can be implemented in any 7-series device. The configuration data is dived in frames, each protected with an Error Correcting Code (ECC). In addition groups of frames are covered by a CRC code for error detection. The Internal Configuration Access Port (ICAP) allows logic and hard blocks part of the FPGA logic to read and modify the FPGA configuration memory, thus allowing to execute a integrity check and take corrective actions. For this task Xilinx offers the Soft Error Mitigation Controller (SEM) IP core, which is a fully integrated solution than continuously read the configuration memory to detect errors and can be used to implement different strategies depending on the requirements.

Fig. 2.9 shows a flowchart with the possible actions that can be implemented when the controller detects an error. While the SEM is running it produces an heartbeat signal that can be used to verify that the controller is actually running. In the implementation used in the project, once the controller finds an error in the configuration memory, the error is automatically

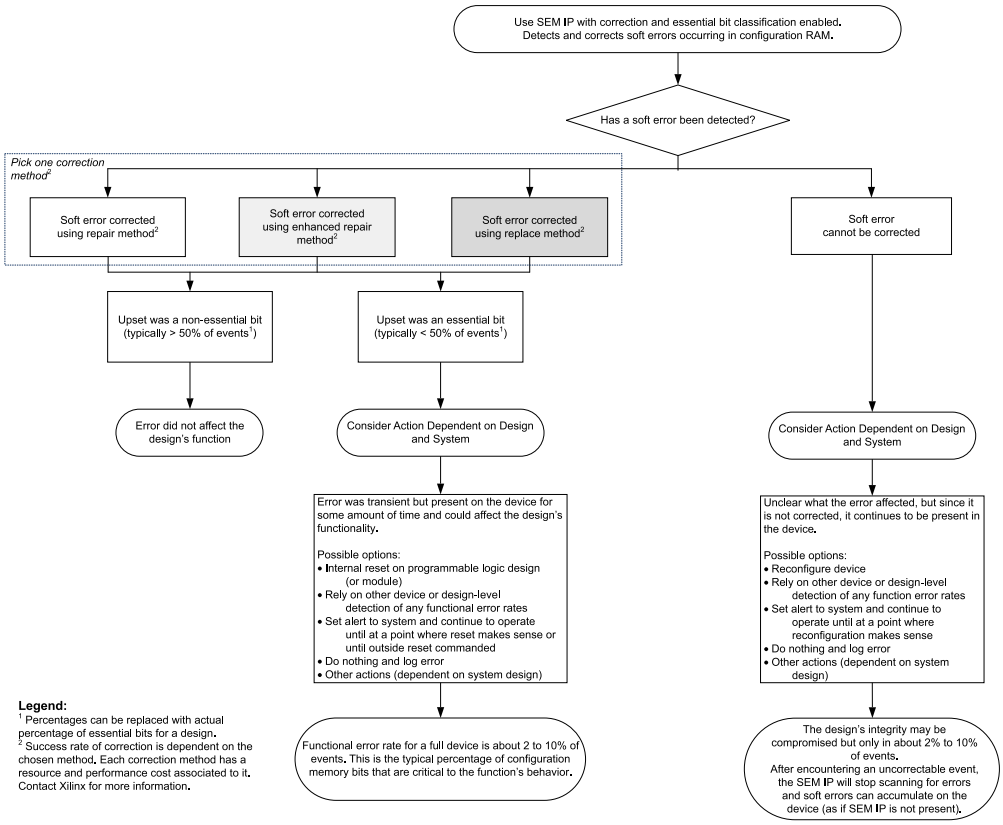


Figure 2.9: SEM decision tree in case an error is detected [2].

corrected (thanks to the ECC). After this it is still possible that underling logic implemented in the FPGA is in an invalid state caused by the broken configuration itself. For this reason, after correction, the firmware tries to raise the alarm signal to the DCPU, which in turn can verify the status of the TDAQ and if necessary apply a reset or even scrub the configuration (by triggering a reload of the FPGA bitstream).

Typically a good part of the FPGA logic is not actually used for a certain design, so to improve the reliability is would be preferable to not trigger any operation for errors in areas that not in use. Using the SEM is possible to enable an "error classification" system, that classify errors in "essential" and "non-essential" bits. This feature require that the SEM have access to the configuration flash of the FPGA, where a "essential bits mask" need to be stored in addition to the bitstream. This feature is not implemented in this project so every error is considered as essential and will trigger the alarm to the DCPU.

Chapter 3

Testing and Calibration Procedures

Test and calibration procedures are both necessary for the qualification of the hardware produced and for the in-flight testing and calibration. Ideally the same procedures could be executed at any time, but the limited bandwidth and computation power available in the TDAQ does not allow the execution of the most complex testing procedures. In this case, a simplified in-flight variant is developed. Switching between on variant or the other will be governed by a compile-time flag in the software.

The following section describes all the procedures developed and their intended purpose. All procedures are executed by the MCU when the corresponding command is received.

3.1 FIFO Test

The FIFO test is a read-back test executed on a given ALPIDE chip internal DPRAM memory used to store data. The selected chip is set in a configuration mode setting the appropriate value in a control register, this allow to used the control line to write arbitrary data into the chip DPRAM. In this test the entire memory is filled and read back with 4 different test patterns (`0x0000, 0xffff, 0xaaaa, 0x55555`).

The purpose of the test is to verify the integrity of the chip internal memory and also more in general to test the communication with the chips and the basic functionality of the chip digital periphery. Any persistent corruption of the data is considered a critical failure of the system. Occasional errors are most likely to be attributed to transmission errors.

3.2 Digital Scan

The digital scan is a test of the functionality of the digital part of the pixel matrix readout circuit. It is divided in two procedures: "digital scan" and "white frame scan".

Digital scan: In the digital scan procedure, a digital pulse is applied to the whole pixel matrix. This is done in practice by broadcasting a PULSE command to all chip in a stave after configuring the relevant pulse configuration register. A digital pulse command simply insert a '1' bit in one of the three in-pixel hit storage latches, so a functioning readout circuit should always register an hit. During the scan, the procedure pulses only one double column at time, and each double columns is pulsed 10 times. The final hit-map read from the chip should then count 10 hits on every pixel; each pixel with a different number of hits is considered faulty.

White frame scan: The white frame scan procedure is identical but the entire pixel matrix is masked before sending the pulses. The purpose of this procedure is to test the functionality of the pixel masking circuit; any pixel reporting a number of hits different from 0 is labelled as "unmaskable". Unmaskable pixels can potentially be very detrimental to the detector performances, since a single (or a group) of unmaskable noisy or stuck-on pixel can generate a large amount of extra data, increasing the readout time and so the final total data rate and detector dead time.

Staves with many unmaskable pixels are then generally excluded from the production. In case a large number of unmaskable pixel is observed after the deployment of the instrument it is possible to disable a single priority encoder or the readout of an entire region. In case this is insufficient, if the number of noisy pixel is too high the entire chip can be excluded from readout altogether.

3.3 Hot Pixels Scan

Noisy pixels are detrimental to the system since increase the readout time and average event size, and so might case a large increase in the detector dead time or data bandwidth occupied. The purpose of the hot pixels scan is to detect "stuck-on" pixels or in general pixels producing excessive noise, so that they can be masked to be excluded from the readout procedure. The discovery procedure is simple:

- The data multiplexer output in the packager firmware block is configure to divert the readout data to the dedicated buffer accessible by the CPU
- For a given stave, the MCU broadcast N triggers commands.
- The MCU read the data and build a list of pixel hit, counting how many times each pixel fires.
- Any pixel firing more than a certain threshold (% of N) is considered noisy and marked.

The resulting list of pixels marked as noisy is stored in a dedicated file in the TDAQ board storage flash memory. This file is read by the processor to generate a pixel mask for the chips after the reception of the command starting the tracker configuration procedure.

For simplicity the file format consist of a simple list of 32-bit binary values, where value[9:0] is the x coordinate of the pixel, value[19:10] the y coordinate, value[27:20] the chip id and value[31:28] the stave id.

During the pixel scan, new values are appended to the file. Removal of an entry from the file is done by overwriting the value with 0. The append procedure will overwrite such zeroed values before appending the new data to the end of the file.

3.4 Threshold Scan

The threshold scan measures the charge threshold of each pixel by injecting a controlled amount of charge directly in the analog front-end. This is done by using the injection capacitor shown in fig.. The amount of charge injected expressed in coulombs can be calculated by:

$$Q = C_{\text{inj}} \cdot (\texttt{VPULSEH} - \texttt{VPULSEL}) \tag{3.1}$$

where `VPULSEH` and `VPULSEL` are two voltage between 0 and AVDD (1.8 V nominal) set by two of the chip internal programmable 8-bit DACs. The injection capacitor has a nominal capacitance value C_{inj} of 160 aF. So every DAC voltage step ($\frac{\text{AVDD}}{255} \simeq 7\,\text{mV}$) is equivalent to around 10 electrons.

The threshold is measured by performing a sweep over the injected charge, and then building a charge versus hits histogram. For every charge step 20 analog pulses (injections) are executed.

This procedure takes a relatively long time to execute and generates a large amount of data. So differently from the other procedures this in the only one that can't be executed in-flight in its complete form. Two variants

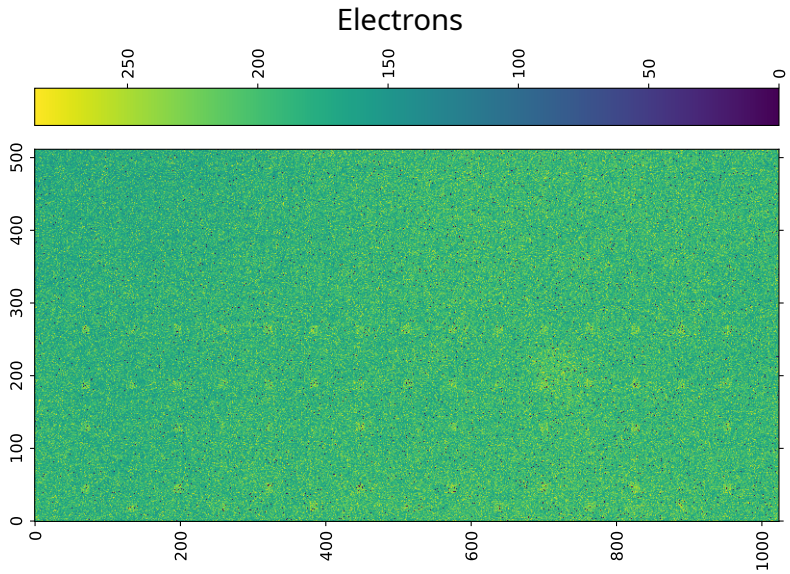


Figure 3.1: Full threshold scan map

of the threshold scan are then defined for the ground qualification of the sensors and in-flight calibration:

Full threshold scan: the full pixel matrix is scanned and event data from each pixel is stored for analysis. In the analysis a hits versus charge histogram is built for each pixel of the sensor and the threshold is determined by fitting with an error function. Data from the fit is used to build a threshold map similar to the one in fig. 3.1. From this the average threshold, threshold RMS, noise and dead/hot pixel count is determined.

Simple threshold scan: the full pixel matrix is scanned but only the global number of hits measured at each charge step over the entire chip area is saved. This measure still allow to gain a measure of the average chip threshold, but it's characterized by a much higher noise (larger sigma from the error function fit). An example of data is shown in fig. 3.2. In practice, the value of the threshold is determined not form a fit, but by seeking the bin where the curve converges (e.g. number of hits $> 90\,\%$ number of pixels). This procedure was defined to cope with the small amount of memory and processing power available on the flight hardware.

Sub-sampled scan: this scan is similar to the simplified scan, but instead of being executed on the entire pixel matrix it is performed on a subsample of pixel. In particular only pixels in a 16 by 8 matrix of 4×4 pixels dots

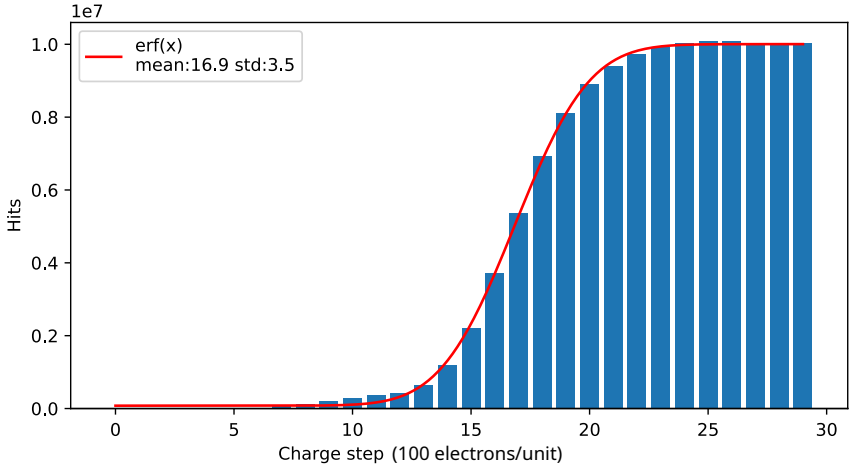


Figure 3.2: Histogram from a simplified threshold scan

are scanned. An example is shown in fig. 3.3. This test offer a even faster alternative to the simple scan that can be executed over the entire detector in a few seconds. Fig. 3.4 shows the threshold histogram for 5 different datasets scanning an area of different size. It can be observed that the estimation of the average threshold don't differ with a precision on the second decimal point.

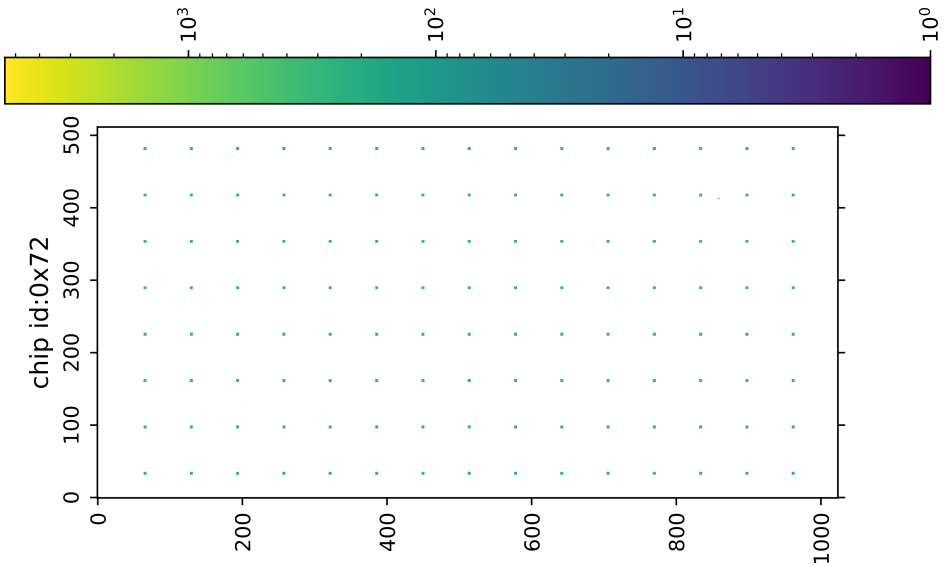


Figure 3.3: Threshold scan performed on a 4×4 pixel dots matrix.

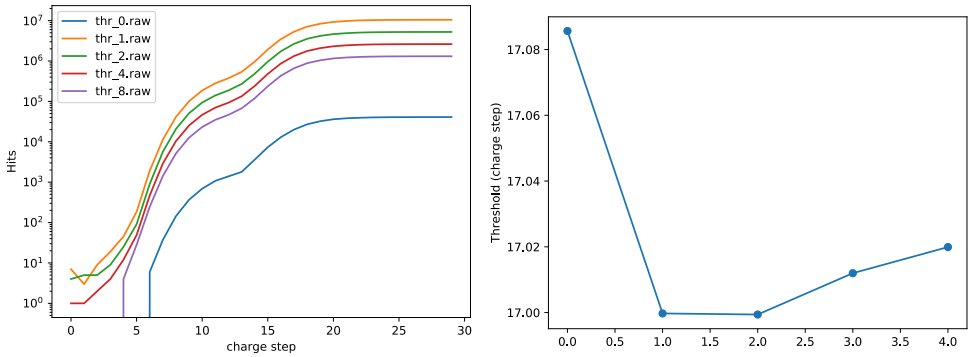


Figure 3.4: Comparison of the threshold estimation with scan covering a different chip area.

Bibliography

[1] ALICE ITS ALPIDE development team. *ALPIDE Operations Manual*, June 2016. ↑3 , ↑4 , ↑7

[2] Xilinx. *PG036 - Soft Error Mitigation Controller*. ↑20

[3] M Šuljić. ALPIDE: the Monolithic Active Pixel Sensor for the ALICE ITS upgrade. *Nuovo Cimento C*, 2018. ↑2