CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering

Department of Telecommunication Engineering

Software for Radiation Detectors Medipix

DIPLOMA THESIS

2011

Student: Bc. Daniel Tureček
Supervisor: prof. Ing. Pavel Zahradník, CSc.

Čestné prohlášení

Prohlašuji, že jsem zadanou diplomovou práci zpracoval sám s přispěním vedoucího práce a konzultanta a používal jsem pouze literaturu v práci uvedenou. Dále prohlašuji, že nemám námitek proti půjčování nebo zveřejňování mé diplomové práce nebo její části se souhlasem katedry.

V Praze dne 12.5.2011

.....
Bc. Daniel Tureček

Acknowledgements

I would like to express my extensive gratitude to Ing. Jan Jakůbek, PhD. who proposed the topic of this work and have given me valuable advices during development of the software. My special thanks go to Ing. Stanislav Pospíšil, DrSc. and Dr. Michael Campbell for giving me the opportunity to spend one year at CERN working on this interesting project. I would like to thank also my supervisor prof. Ing. Pavel Zahradník, Csc. for supervising this thesis. My thanks go also to my colleagues at CERN Rafael Ballabriga, Winnie Wong and Xavier Llopart for their support and for creating a very friendly working atmosphere. Last but not least I would like to thank my family and friends, without their support this work would not be possible.

Daniel Tureček

Zadání diplomové práce

(Originál v originálu diplomové práce, oboustranná kopie v kopii diplomové práce)

Summary

This diploma thesis is devoted to the upgrade and extension of the software package Pixelman for control, power and data acquisition of pixel detectors of the Medipix type such as Medipix2, MXR and Timepix. The package is designed to operate the detectors with devoted interfaces - MUROS2 and the USB-based integrated interfaces. These interfaces with connected detectors are controlled on the PC side by Pixelman software package. Pixelman is now newly extended to support the new Medipix3 detector and the new read-out interface FITPix. Further, the software has been extended so that it is possible to operate it on 3 different operating systems: Microsoft Windows, Mac OS X and Linux. For this purpose the Java programming language and its Swing framework are used. A new graphical user interface was created in Java, which features device window docking and enhanced monitoring. Also, a bridge between the software core written in C++ and Java programming language was developed as well as a framework allowing creation of Java extension modules (plugins). The operation of the package is evaluated and demonstrated by real measurements of a network of neutron detectors in the ATLAS experiment at CERN and tests for dosimetry studies in space with NASA.

Anotace

Cílem této diplomové práce je rozšíření stávajícího softwaru Pixelman sloužícím k ovládání a vyčítání dat z pixelových detektorů typu Medipix (Medipix2, MXR a Timepix). Software je navrženy tak, aby pracoval s vyčítacími systémy MUROS2 a USB rozhraním. Tyto systémy s připojenými detektory jsou ovládány na straně počítače softwarem Pixelman. Pixelman je v této práci rozšířen o podporu nového detektoru Medipix3 a nového vyčítacího rozhraní FITPix. Dále byl software rozšířen tak, aby bylo možné ho použít na 3 různých operačních systémech: Microsoft Windows, Mac OS X a Linux. K tomuto účelu je využito programovacího jazyku Java a jeho knihovny Swing. Bylo také vytvořeno nové grafické uživatelské rozhraní, které umožňuje dokování oken a poskytuje vylepšené monitorování měření. Dále byl vytvořen most mezi jádrem programu napsaném v jazyku C++ a jazykem Java a rozhraní pro rozšiřující moduly (pluginy) napsané v jazyce Java. Funkce programu byla ověřena a demonstrována měřeními v síti neutronových detektorů v ATLAS experimentu v CERNu. Ve spolupráci s NASA byly provedeny testy využití softwaru a detektoru Timepix pro dosimetrická měření ve vesmíru.

Contents

List of Figures

List of Tables

Introduction

Significant progress in the last years in instrumentation for particle detection and radiation imaging has opened new areas of study in high energy nuclear physics as well as novel applications in fields such as medicine and material analysis. In the mid-1990s, particle physicists working on detection methods for the new LHC particle accelerator at CERN discovered that hybrid semiconductors detectors are a promising technology for tracking applications in high energy physics. A collaboration of 4 institutes was created for the purpose of developing a Photo Counting Chip (PCC). The Medipix collaboration [1] was formed. In 1997 Medipix1 - a hybrid pixel single photon counting chip was presented.

The semiconductor quantum counting pixel detectors of the Medipix type (256x256 square pixels, 55x55 μm each) such as Medipix2 [2], MXR [3] and Timepix [4] are superior imaging devices in terms of sensitivity (detect single particles), signal-to-noise ratio, spatial resolution, linearity and dynamic range. This makes them suitable for various applications such as radiography and neutronography.

The control and data acquisition of the Medipix detectors are currently realized by dedicated read-out systems – MUROS2 [5] and integrated USB-based interfaces [6], [7]. The USB-based interfaces are highly portable read-out systems developed at the IEAP[1] in Prague. They provide integrated control, power, and DAQ while featuring small dimensions, ease of use, portability and no need for any external power supply.

The control of read-out interfaces and data acquisition as well as data processing of pixel detectors of the Medipix type on standard PC has been realized by the Pixelman [8] software package. Pixelman is a flexible and extendable software platform for Medipix2 detectors running under Microsoft Windows OS. It is compatible with the MUROS2 and USB-based interfaces. The functionality of the software can be extended by custom-made modules (plugins). However the software could be used only on the Microsoft Windows platform. With increasing popularity of Linux and Mac OS X operating systems, especially the Linux platform in the scientific community, it was decided to extend the operability of Pixelman to other operating systems.

The detectors from the Medipix family have proven to be an important tool for real time imaging with high sensitivity and wide dynamic range. In 2009 a new

[1]Institute of Experimental and Applied Physics, Czech Technical University in Prague (www.utef.cvut.cz)

addition to Medipix family was presented - Medipix3 chip. It is a 256x256 channel hybrid pixel detector chip working in single photon counting mode with a novel architecture aimed at eliminating the spectral distortion produced by the charge sharing process. The chip also permits color imaging, and dead-time-free operation.

In 2010 a new read-out interface FITPix [9] has been developed at IEAP in Prague. This new interface addresses a slower read-out speed of the USB-based interfaces, while preserving the same physical dimensions and improving the read-out speed up to 90 frames per second. FITPix brings also new features such as adjustable clock (for the Timepix detectors), external triggering and hardware timer with precision 20 ns. This makes FITPix the fastest and most portable interface for Medipix detectors available.

In a view of increasing requirements of measurement methods and user demands, as well as the arrival of new hardware (Medipix3 detector and new read-out interface FITPix) and requirements for multi-platform software a need has been arisen for a major upgrade of the Pixelman software package. The goal of this work is to perform such an upgrade and extension of the software to support the new Medipix3 detector and the new read-out interface FITPix. Further, the software is extended in a way that it is possible to operate it on 3 different operating systems: Microsoft Windows, Mac OS X and Linux. For this purpose the Java programming language and its Swing framework are chosen. A new graphical user interface is created in Java, which features device window docking and enhanced monitoring. Also, a bridge between the software core written in C++ and Java programming language was developed as well as a framework allowing creation of Java extension modules (plugins). The operation of the package is evaluated and demonstrated by real measurements of a network of neutron detectors in the ATLAS experiment at CERN and tests for dosimetry studies in space with NASA.

A description of the content of this thesis follows:

Chapter 1: An overview of all the detectors in the Medipix family is given and parameters of Medipix detectors significant for a development of software are described.

Chapter 2: Three available read-out interfaces for Medipix detectors (MUROS2, the USB-based interface and FITPix) are reviewed. Their architectures, parameters and communication protocols with the host PC are described.

Chapter 3: A description of the original software package, whose upgrade and extension is subject of this work, is given. The software architecture and extension possibilities are discussed.

Chapter 4: A brief summary of objectives and reasons for this work are given. An overview of implemented extensions to the software including support for Medipix3 detector, FITPix read-out interface is given.

Chapter 5: A development and architecture of two multi-platform hardware libraries for USB and FITPix interface compatible with Medipix2 MXR, Timepix and Medipix3 detectors is discussed.

Chapter 6: Principles of interfacing C++ programming language with Java programming language are described An application of these principles in a form of a bridge between Pixelman core written in C++ and Java interface is shown. A description of a Java framework for Java plugin development is given together with description of several Java plugins implementing the new Java graphical user interface.

Chapter 7: An overview of real-life application of the software in large projects is given. A distributed acquisition system using Pixelman software controlling network of 16 Medipix2 devices (ATLAS-MPX Network) in ATLAS detector at CERN is described. Brief overview of Pixelman remote network interface, central controlling application and data visualization application is shown. Moreover, a project with NASA and University of Houston to demonstrate dosimetry application of Timepix detector at the International Space Station (ISS) is described. Also a special dosimetric plugin developed for this project is shown.

1 Medipix Detectors

This chapter describes briefly basic information about radiation detectors Medipix. It is a summary of chip manuals and articles devoted to Medipix detectors: [3], [10], [11], [12], [13], [14].

1.1 Medipix Family

Medipix is a family of photon counting pixel detectors developed in a frame of *Medipix Collaboration* [1] at CERN[1]. It consists of 5 detectors: *Medipix1, Medipix2, Medipix2 MXR, Timepix and Medipix3*. Medipix detectors are hybrid detectors. This means that the read-out chip and the sensor are manufactured independently. The sensor is usually a semiconductor such as GaAs or CdTe in which the incident radiation interacts with sensor in that way that it creates electrons/hole pairs. This created charge then goes into the read-out chip that counts the number of events in each pixel. The advantage of hybrid detector is that the read-out chip can be designed in standard CMOS technology independently of the sensor.

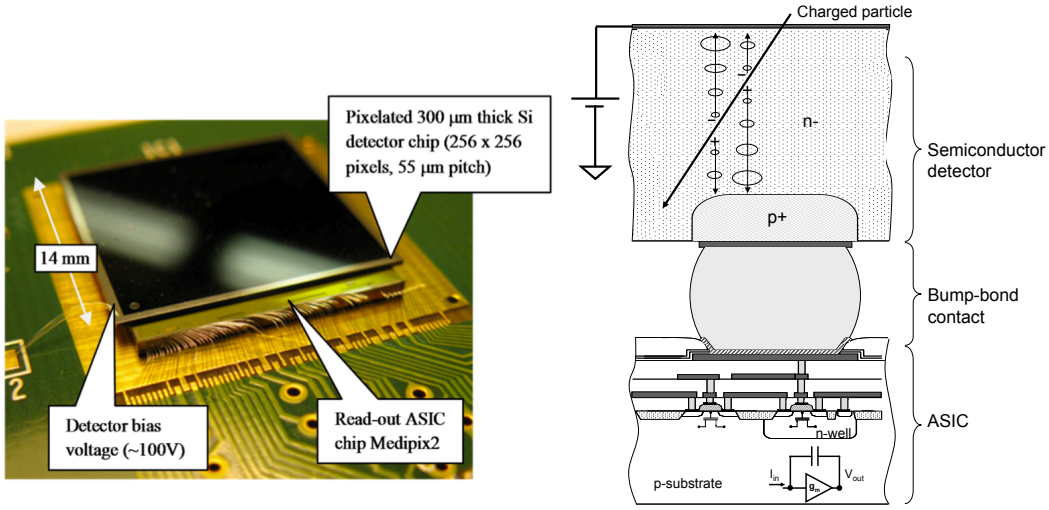


Figure 1.1: A photograph of Medipix2 detector with a sensor [17] and principal schema of hybrid pixel detectors.

[1]European Organization for Nuclear Research.

Medipix detectors can be used in many different applications. Figure 1.2 demonstrates one of those application - high contrast X-ray transmission radiography. The figure shows an image of pelvis and backbone of a living mouse. The small circular inset (bottom right) taken at higher magnification demonstrates that the contrast of X-ray images taken with Medipix detector can be high enough to distinguish individual hair fibers on the mouse skin through the full thickness of its body.[17]

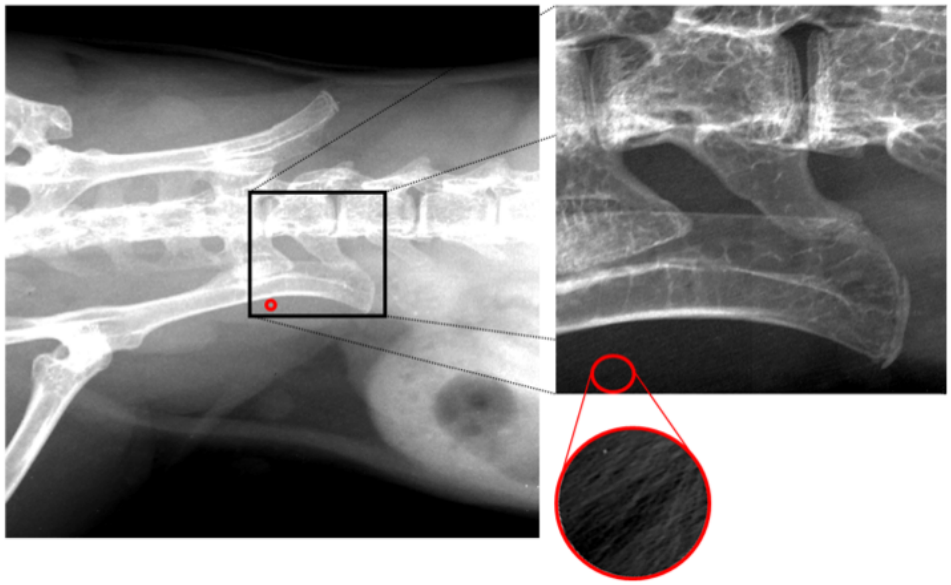


Figure 1.2: Images of pelvis and backbone of living mouse taken with Medipix2 QUAD system (2x2 Medipix2 chips tilled together).[17]

1.2 Medipix1

Medipix1 was a prototype photon counting chip with 64x64 square pixels of 170μm side-length, sensitivity to positive charge, 15-bit counter per pixel and an active area about 1.2 cm. This detector was later replaced by MedipixMXR and Timepix detectors that are the most frequently used Medipix detectors nowadays. The last addition to the Medipix family is Medipix3 detector.

1.3 Medipix2

Medipix2 chip is a successor of Medipix1 chip. It is a pixel detector read-out chip consisting of 256x256 pixels, each working in single photon counting mode for positive or negative input charges. Each pixel has a surface area of 55μm x 55μm and contains two discriminators, two 3-bit thresholds adjustments and 13-bit pseudo-random counter. The detector offers very good spatial resolution and unlimited dynamic range which makes it suitable for radiographic and tomographic measurements.

1.3.1 Physical Layout

The Medipix2 chip surface (see Figure 1.3) is divided into sensitive and non-sensitive area. The sensitive area (on the top) is matrix of 256x256 pixels representing total detection area of 1.98 cm (87 % of entire area). The non-sensitive area (on the bottom) represents periphery containing 13 8-bit DACs[2] and the input/output control logic. The periphery was placed at the bottom of the chip in order to minimize the dead area between chips when butting several chips together.

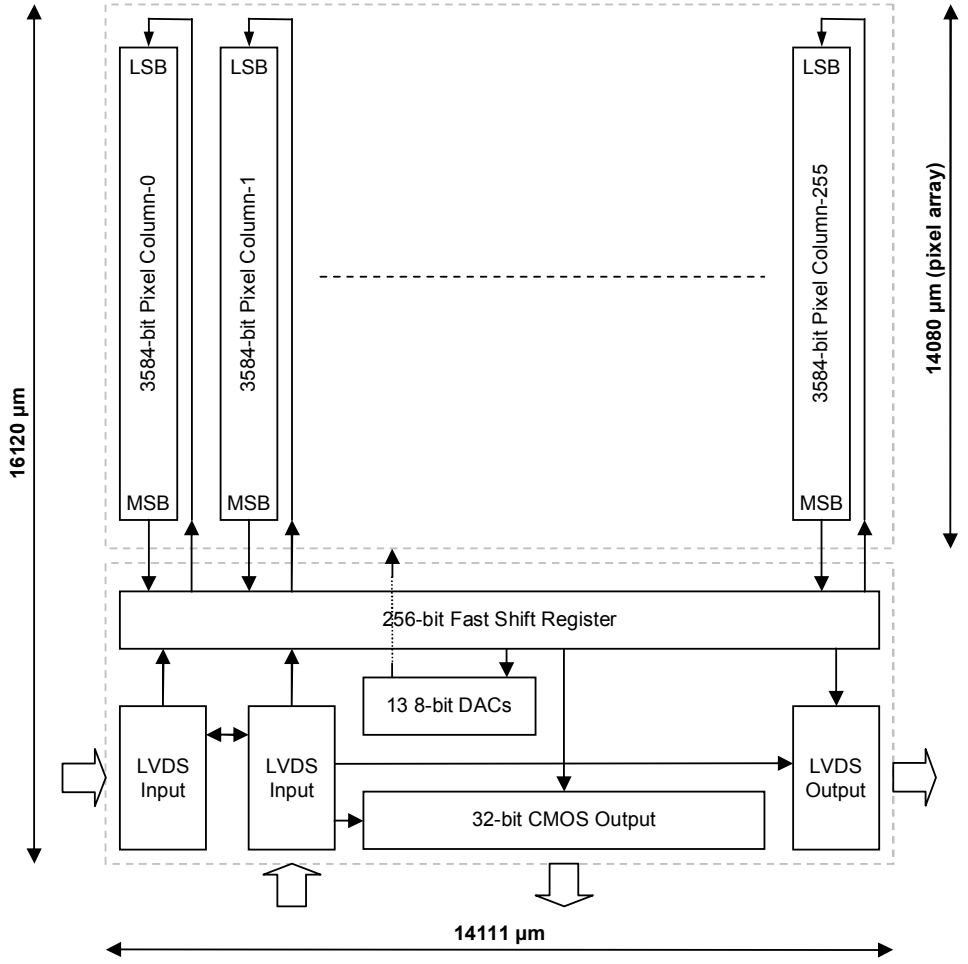


Figure 1.3: Medipix2 surface plan and block diagram [6]

1.3.2 Pixel Cell

The Medipix2 pixel cell contains analog and digital part. The analog part consists of a charge preamplifier DC leakage current compensation, a test capacitance and two discriminators. The digital side contains Double Discriminator Logic (DDL) and 13-bit shift register with overflow bit.

[2]Digital-to-Analog Converter

When charged particles interact in the detector material they deposit a charge that is drifted towards the collection electrodes under influence of detector bias voltage. This charge is amplified and compared with two different thresholds forming an energy window. If charge is in the range of this window the digital pseudo-random counter is incremented.

Each pixel can be configured by eight independent configuration bits. Six bits are used for fine threshold adjustment (3 bits for each discriminator), one for masking noisy pixels and one to enable the input charge test. Pixel configuration bits are set through 14-bit shift register. A placement of 8 configuration bits in the shift register is shown in Figure 1.4. Their functionality is shown in Table 1.1 where THL is 3-bit low threshold adjustment bit (bit 0_{LSB}, bit 1, bit 2) and THH is 3-Bit high threshold adjustment bit (bit 0_{LSB}, bit 1, bit 2).

0	1	2	3	4	5	6	7	8	9	10	11	12	13
X	X	X	X	Bit 2 High	Bit 1 High	X	Bit 2 Low	Bit 1 Low	Bit 0 Low	Test Bit	Mask Bit	Bit 0 High	X

Figure 1.4: Medipix2 pixel configuration bits positions

Bit	High (1)	Low (0)
THL	Not Active	Active
THH	Not Active	Active
Test Bit	Not Active	Active
Mask Bit	Not Masked	Masked

Table 1.1: Medipix2 pixel configuration bits functionality

Each pixel works in two modes: data acquisition mode and readout/setting mode. In the data acquisition mode the shift register works as a pseudo-random counter of 13-bit with a dynamic range of 8001 counts. The output of the double discriminator logic is used as clock for this counter. Every pulse coming from the discriminators increments the counter value by one unit. After the maximal value of the counter is reached, an overflow bit is set to zero. The read-out of the data is performed after the exposure.

In readout/setting mode the 14-bit shift registers of each pixel in each column are connected together so that they form a 3584-bit shift register as shown in Figure 1.4. Transfer of data between pixel matrix and IO logic is done through 256-bit FSR[3]. The FSR is used for reading out of measured data as well as setting the 8-bit configuration.

[3]Fast Shift Register

1.3.3 IO Operations

Medipix2 chip operates in several IO operation modes:

General Reset During the reset the chip FSR bits are set high, all IO counters are reset to default values and all DACs are set to their mid-range values.

Counting Mode Each pixel in the matrix starts to count independently. Noisy pixels can be disabled by setting the mask bit in pixel configuration bits.

Matrix Read-out Two read-out modes are possible: serial and parallel. Parallel readout is not yet implemented in read-out interfaces. In the case of serial read-out received data contain 8 preload register bits followed by MSB[4] of the pixel from right-bottom corner (column 255) followed by MSB of the neighboring pixel from column 254 and so forth. Last bit in the data is LSB[5] of the pixel from left-upper corner of the chip (see Figure 1.3).

Setting the Matrix This is the opposite operation to matrix read-out. The FSR is used to write 8 configuration bits for each pixel into the matrix. Bits are placed in the FSR as shown in Figure 1.4. Due to a preload register, eight dummy bits has to be sent before matrix data.

Setting the DACs This mode is used to set values of the on-chip DACs. DACs are described in section 1.3.4

Resetting the Matrix This operation resets all the counter bits to high without affecting pixel configuration registers.

Testing the FSR This operation is used only for testing of 256-bit Fast Shift Register functionality.

Test Pulse This testing structure is used to test functionality of the pixel without an X-ray source and also to calibrate the pixel matrix. Test pulse is individually selectable for each pixel through Test Bit in pixel configuration register (see Figure 1.4).

[4]Most Significant Bit
[5]Least Significant Bit

1.3.4 DACs

There are 13 8-bit DACs in the chip. Their role is to provide bias (voltage and current) to analogue and digital circuitry within pixel cells. DACs values are set through 256-bit FSR. 5 bits of the FSR register have a special purpose: Sensing of each DAC value and bypassing one of the DACs with an external DAC. Position of each DAC in the FSR register is shown in Table 1.2. Table 1.3 shows position of special control bits.

Dac name	Code	Bits in FSR <LSB:MSB>
VbiasDelayN	0001	<3:10>
VbiasDisc	0010	<11:18>
VbiasPreamp	0011	<19:26>
VbiasSetDisc	0100	<45:48><55:58>
VbiasThs	0101	<59:66>
VbiasIkrum	0111	<99:106>
VbiasABuffer	1110	<107:114>
VthH	1100	<115:122>
VthL	1011	<123:130>
Vfbk	1010	<131:138>
Vgnd	1101	<180:187>
VbiasLVDStx	0110	<226:233>
VrefLVDStx	1001	<234:241>

Table 1.2: Medipix2 DACs

Control bits	FSR position
DAC Code (B0:B3)	B37,B38,B40,B41
Sense DAC (Active High)	B42
External DAC_SEL (Active High)	B43

Table 1.3: Medipix2 special DACs control bits

1.4 Medipix2 MXR

Medipix2 MXR chip is a redesign of Medipix2 chip. Its pixel pitch, floor plan and dimensions are the same as Medipix2 but pixel cell and periphery were fully redesigned. The main reasons for the redesign were:

- DACs showed a relatively high temperature dependence and a non-optimal linearity.
- Analog buffers did not have a unary gain response, i.e. it was not possible to do an absolute calibration using injection test pulse.

- The pixel counter was reinitialized when the maximum number of counts was reached.
- A measured pixel radiation hardness was lower than expected.

The redesign of the chip solved all of the problems mentioned above and also added several new features as well as improved functionality and robustness of the chip.

1.4.1 Pixel Cell

The pixel architecture of Medipix2 MXR is very similar to Medipix2. The main functional difference is an increased pixel counter depth and an overflow control logic. The chip has now 14-bit pseudo-random counter that counts up to 11810. The new overflow control logic stops the counting in the counter if the number of pulses coming to the counter is higher than 11810. There is no overflow bit, instead all the pixels with a counter value set to the maximum (11810 counts) are considered as overflown.

The Medipix2 MXR pixel configuration register has the same number of bits and the functionality as Medipix2. Only positions of the bits are different as shown in Figure 1.5.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
Mask Bit	X	X	X	X	X	Bit 1 Low	Bit 0 Low	Bit 2 Low	Test Bit	Bit 1 High	Bit 2 High	Bit 0 High	X

Figure 1.5: Medipix2 MXR pixel configuration bits positions

Bit	High (1)	Low (0)
THL	Not Active	Active
THH	Not Active	Active
Test Bit	Active	Not Active
Mask Bit	Not Masked	Masked

Table 1.4: Medipix2 MXR pixel configuration bits functionality

1.4.2 IO Operations

The IO operation modes in Medipix2 MXR are similar as in Medipix2 except of these differences:

Matrix Reset This operation mode has been eliminated in Medipix2 MXR. Matrix can be reseted by a dummy serial or parallel read-out.

Matrix Read-out Reading-out of the matrix uses same procedure as in Medipix2. The difference is that 256 post-load bits for synchronization purposes are appended to the data. Therefore the structure of data is: *8 bits preload + matrix data + 256-bit post-load.*

Setting the Matrix the only difference here is also 256-bit long post-load.

Setting the DACs The 13 on-chip integrated DACs are set through FSR. The FSR is also used to set 32-bit Column Test Pulse Register (CTPR) described below. Unlike FSR in Medipix2 chip, the FSR in Medipix2 MXR has one more function: reading the content of the fuses (see below) that represent a chip ID.

1.4.3 DACs

A stability of DACs output values to temperature and DC power supply variations was improved in Medipix2 MXR. The chip contains two 14-bit voltage DACs for precise setting of threshold and 11 8-bit current and voltage DACs. The 14-bit DACs are made of hight linearity 10-bit DAC (fine) and a 4-bit DAC (coarse) whose current outputs are summed. A list of the DACs and their positions in the FSR is shown in Table 1.5. Position of the special bits is the same as for Medipix2 (see Table 1.3).

DAC name	Code	Bits in FSR <LSB:MSB>
IKrum	1111	<3:10>
Disc	1011	<11:18>
Preamp	0111	<19:26>
BuffAnalogA	0011	<45:48><55:58>
BuffAnalogB	0100	<59:66>
DelayN	1001	<85:92>
THL	0110	<99:112>
THH	1100	<113:126>
FBK	1010	<127:134>
GND	1101	<135:142>
THS	0001	<180:187>
BiasLVDS	0010	<226:233>
RefLVDS	1110	<234:241>

Table 1.5: Medipix2 MXR DACs

1.4.4 The Fuses

The fuses are one of the new features in Medipix2 MXR chip. They are used to identify uniquely chip wafer number and all chips in the wafer by means of a 24-bit register. This identification is done by laser blowing an array of 24 fuses at

the foundry. Every time the chip is reseted the contents of the 24-bit are loaded into the FSR register at bit positions <195:218>. The wafer number, x and y position on the wafer are encoded in these bits as shown in Figure 1.6.

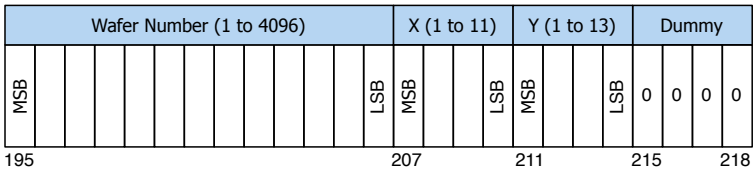


Figure 1.6: Medipix2 MXR fuses bits in FSR

1.4.5 Test Pulse Architecture

The Medipix2 test pulses injection input charge was not properly defined because of gain variation of analog buffers and a coupling between neighbor circuitry. The Medipix2 MXR includes unitary gain buffers and a 32-bit Control Test Pulse Register (CTPR) that selects individually the pixel columns to test.

In Medipix2 the calibration test pulse voltage was sent to all the pixels in the matrix simultaneously regardless of number of pixels selected to be tested (Test Bit active). The resulting calibration measurement was imprecise as a result of crosstalk between global metal lines in the chip. The 32-bit CTPR address this issue by selecting which columns should be tested simultaneously. Every bit in CTPR select 8 columns distributed uniformly every 32 columns (i.e. for bit 0 columns 0, 32, 64, 96, 128, 160, 192 and 224 are selected).

1.5 Timepix

Timepix chip is an evolution of Medipix2 MXR chip which proved to be a big success in single photon counting. Although Medipix2 MXR demonstrated that single primary electrons could be detected, the chip did not provide any information on the arrival time of the electron nor the quantity of charge deposited. The motivation for Timepix design was to provide this kind of information. The Timepix chip allows a measurement of arrival time, "time-over-threshold" (TOT) and/or event counting independently in each pixel. An external reference clock was added to generate a clock in each pixel that increments the counter depending on the selected pixel operating mode. The chip has the same dimensions and read-out architecture as Medipix2 MXR chip. This requires just a minor modification in the read-out hardware to support new features.

Figure 1.7 demonstrates one of the possible applications of Timepix detector - X-ray fluorescence imaging (XRF). This method is based on spectrometry of characteristic X-ray radiation emitted by excited atoms of the sample. The sample excitation is usually performed by its irradiation with X-ray tube. Standard XRF

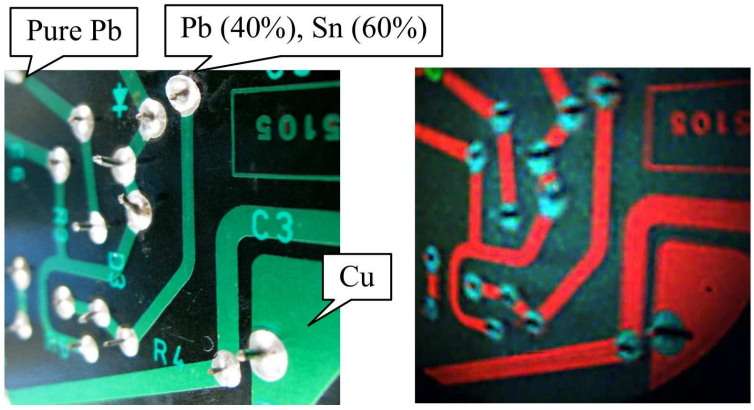


Figure 1.7: A piece of PCB (printed circuit board) contains regions made of copper and solder (alloy of tin and lead). The point in upper left corner was soldered with pure lead (left). The measured content of copper, lead and tin are RGB color encoded in the XRF image (right).[17]

method serves for elemental analysis at certain area of the sample surface. When also element distribution is needed then an energy sensitive imaging method has to be used. In this case the energy sensitive imaging detector Timepix in combination with camera-obscura (pin-hole collimator) is used as depicted in Figure 1.8 and 1.7.[17]

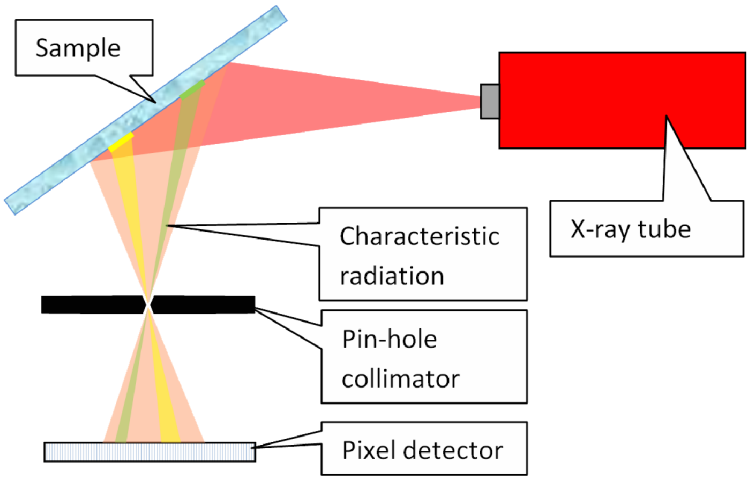


Figure 1.8: Experimental setup used for X-ray fluorescence imaging. The material of the sample is excited by radiation from an X-ray tube. The characteristic radiation emitted by sample is projected onto detector surface by pin-hole collimator (with diameter of 200 μm).[17]

1.5.1 Pixel Cell

The pixel cell of Timepix chip resembles Medipix2 MXR pixel cell except of three main differences:

- There is a single threshold with one 4-bit threshold adjustment DAC.
- Each pixel can be configured independently in 4 different operation modes configured in P0 and P1 bits: arrival time, TOT and event counting.
- During acquisition mode a counting clock is distributed to each pixel.

The Timepix pixel configuration register as opposed to Medipix2 MXR does not have 2 threshold adjustments but only one, and has two special bits P0 and P1 that set pixel operation mode. The position of the bits in the register is shown in Figure 1.9 and their functionality is described in Table 1.6.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
X	X	X	X	X	X	P1	Mask Bit	Thr Bit 0	P0	Thr Bit 2	Thr Bit 3	Thr Bit 1	Test Bit

Figure 1.9: Timepix pixel configuration bits positions

Bit	High (1)	Low (0)
Threshold	Not Active	Active
Test Bit	Active	Not Active
Mask Bit	Not Masked	Masked
Pixel Mode	see below	see below

Table 1.6: Timepix pixel configuration bits functionality

1.5.2 Pixel Modes

Each Timepix pixel can be configured independently in 4 different modes via P0 and P1 configuration bits. These modes are:

Medipix This is an event counting mode. Each event above the threshold increments the counter by 1. This corresponds to Medipix2 functionality.

Time over Threshold (TOT) The counter is incremented continuously as long as the signal is over threshold. It is used to measure particle energy.

Timepix 1-hit The counter is incremented to value 1 if there is at least 1 hit during the data acquisition.

Timepix The counter is incremented from the moment first hit arrives until the end of the data acquisition. The time that can be measured in this mode is limited by depth of the 14-bit counter. For 50 MHz clock the maximum time is about $23\mu s$.

All possible pixel modes including masking of the pixel are shown in Table 1.7

Pixel Mode	Mask Bit	P0 Bit	P1 Bit
Masked	0	X	X
Medipix	1	0	0
TOT	1	1	0
Timepix 1-hit	1	0	1
Timepix	1	1	1

Table 1.7: Timepix pixel modes

1.5.3 DACs

Timepix includes 8 current and 5 voltage DACs. Most of the DACs are the same as in Medipix2 MXR chip. Only the THH DAC was replaced by Vcas and DelayN was replaced by Hist. Summary of all the DACs is shown in Table 1.8.

DAC name	Code	Bits in FSR <LSB:MSB>
IKrum	1111	<3:10>
Disc	1011	<11:18>
Preamp	0111	<19:26>
BuffAnalogA	0011	<45:48><55:58>
BuffAnalogB	0100	<59:66>
Hist	1001	<85:92>
THL	0110	<99:112>
Vcas	1100	<119:126>
FBK	1010	<127:134>
GND	1101	<135:142>
THS	0001	<180:187>
BiasLVDS	0010	<226:233>
RefLVDS	1110	<234:241>

Table 1.8: Timepix DACs

1.6 Medipix3

Medipix3 is the most recent addition to the Medipix family. It is a 256x256 channel hybrid pixel detector chip working in single photon counting mode with a novel architecture aimed at eliminating the spectral distortion produced by the charge

sharing process. The chip was developed in an 8-metal 0.13μm CMOS technology and has dimensions of 17.3 x 14.1 mm^2. The chip implements a charge summing architecture. Medipix3 pixel counter has been redesigned to allocate 2 depth programable binary counters and continuous count-read mode. It is the first high resolution X-ray spectral imager hybrid pixel detector.

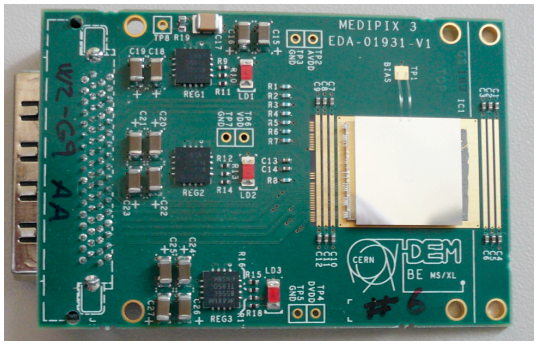


Figure 1.10: Medipix3 detector on the chip-board

Medipix3 chip shares some similar features as previous versions of Medipix chips such as matrix size of 256x256 pixels or configurable DACs. However in many aspects it uses an entirely different approach. The chip can be used in several acquisition modes: Fine pitch mode, Spectroscopic mode, Charge summing mode. The pixel matrix can be programmed in high or low gain mode. Each pixel of the matrix includes two counters with configurable bit depth. Data read-out can be performed either sequentially or continuously. Either the entire matrix can be read-out or only part of the matrix (region of interest). Lastly, all the chip operation (reading of matrix, setting the DACs, setting pixel configuration, ...) is controlled through special 48-bit register: Operation Mode Register (OMR).

1.6.1 Physical Layout

Medipix3 chip dimensions are 17.3 x 14.1 mm^2. The surface is divided into 3 parts: the sensitive area, top periphery and bottom periphery (see Figure 1.11). The sensitive area (on top in the figure) contains 256x256 matrix of 55 x 55 μm square pixels. The regular structure in the matrix is implemented as a cluster composed of 4 pixels with different layout. This allows an implementation of Colour Mode functionality, where only one pixel in the 4-pixel cluster is bump-bonded.

The top periphery (not shown in the figure) is a row of power pads which may be used to feed additional power from the top into the chip in the case this is needed.

The bottom periphery (bottom part of the figure) is the interface between the active area and the Medipix3 read-out hardware. It contains IO and power pads, Medipix3 IO logic, 32 e-fuse bits containing chip ID, 25 DACs and for each pixel column one End Of Column circuit and 2 test pulse circuits.

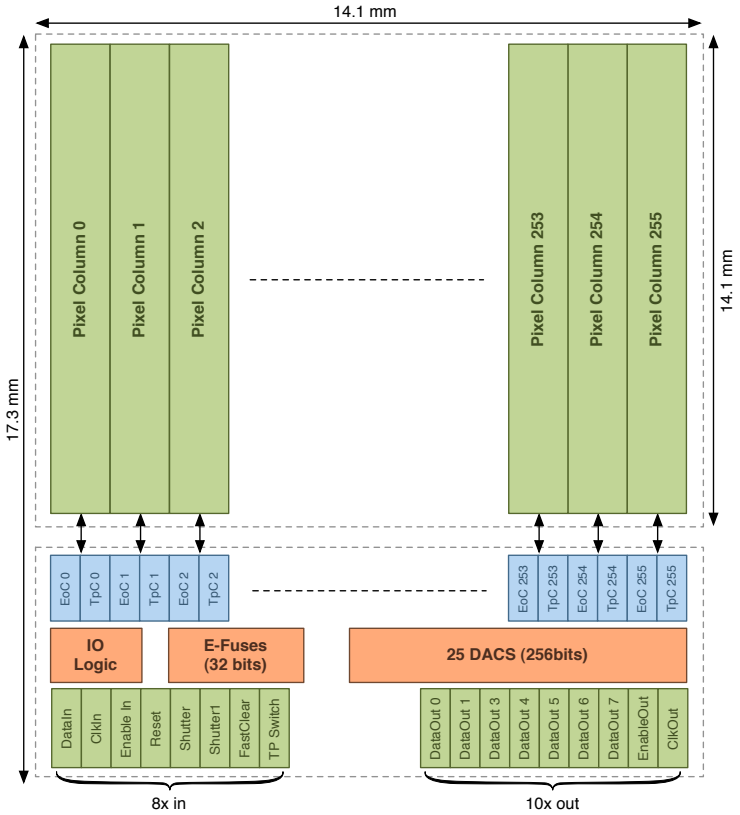


Figure 1.11: Medipix3 schematic - pixel matrix and bottom periphery

1.6.2 Pixel cell

Medipix3 pixel cell is more complex compared to previous version of Medipix detectors. It contains two independent 12-bit counters with configurable depth (1-bit, 4-bit, 12-bit). The counters can be used either separately or combined into one counter of double depth (24-bit). Each counter is associated with one configurable threshold. This allows simultaneous counting and readout when the pixel read-out is configured in Continuous Read-Write mode (see 1.6.3).

In order to fully test chip functionality, a testing capacitance is provided in each pixel. Two different test pulses are available to allow injection of different charges in adjacent pixels.

The mode of operation of each pixel is configured using 13 local configuration bits and global metal lines. These configuration bits are placed in the 24-bit counter as shown in Figure 1.12. Their functionality is described in Table 1.9

1.6.3 Operation Modes

The pixel front-end can be configured in three modes of operation: Single Pixel Mode (SPM), Charge Summing Mode (CSM) and Colour Mode:

0	1	2	3	4	5	6	7	8	9	10	11
X	X	X	X	Cfg THB 4	Cfg THA 1	Cfg THA 2	Cfg THA 4	Gain Mode	X	X	X
12	13	14	15	16	17	18	19	20	21	22	23
X	X	X	Test Bit	Cfg THB 3	Cfg THA 0	Cfg THB 0	Cfg THA 3	Cfg THB 2	Mask Bit	Cfg THB 1	X

Figure 1.12: Medipix3 pixel configuration bits positions

Bit	High (1)	Low (0)
Test Bit	Active	Not Active
Gain Mode	High Gain	Low Gain
Mask Bit	Masked	Not Masked
ThresholdA	Active	Not Active
ThresholdB	Active	Not Active

Table 1.9: Medipix3 pixel configuration bits functionality

1. **Charge Summing Mode** Charge in every cluster of 4 pixels is added and assigned to the pixel with the largest charge deposition. This is a way of eliminating the spectral distortion due to charge diffusion in the sensor.

2. **Single Pixel Mode** each pixel works independently. Similar to the operation of previous Medipix detectors.

3. **Colour Mode** The pixels are grouped in the clusters of 4 and creates a single detection unit. This mode is used when pixel pitch is $110\mu m$x$110\mu m$ and only one of 4 pixels is bump-bonded. The chip can be programmed for SPM or CSM operation:

   - **Single Pixel operation** - the pixel pitch is $110\mu m$x$110\mu m$ and 8 thresholds are available for each pixel.
   - **Charge Summing operation** - all the charge falling in an area of $220\mu m$x$220\mu m$ is assigned to only one "virtual pixel" with spatial resolution $110\mu m$x$110\mu m$. 8 thresholds per pixel are available for colour imaging.

1.6.4 DACs

Medipix3 chip has 25 on-chip DACs. Their names, DAC code and position in the DAC register are summarized in Table 1.10. First 8 DACs are used for threshold adjustments. In the Colour Mode all of them can be used to specify 8 energy levels for

one super-pixel (group of 4 pixels with only one pixel bump-bonded). In other modes only first two (*Threshold0* and *Threshold1*) are used for each counter respectively. In the 24-bit mode only *Threshold0* is used.

DAC name	Code	Position
Threshold[0]	00001	<46:54>
Threshold[1]	00010	<55:63>
Threshold[2]	00011	<64:72>
Threshold[3]	00100	<73:81>
Threshold[4]	00101	<82:90>
Threshold[5]	00110	<91:99>
Threshold[6]	00111	<100:108>
Threshold[7]	01000	<100:108>
Preamp	01001	<118:125>
Ikrum	01010	<126:133>
Shaper	01011	<134:141>
Disc	01100	<142:149>
Disc_LS	01101	<150:157>
ThresholdN	01110	<158:165>
DAC_pixel	01111	<166:173>
Delay	10000	<174:181>
TP_BufferIn	10001	<182:189>
TP_BufferOut	10010	<190:197>
RPZ	10011	<198:205>
GND	10100	<206:213>
TP_REF	10101	<214:221>
FBK	10110	<222:229>
Cas	10111	<230:237>
TP_REFA	11000	<238:246>
TP_REFB	11001	<247:255>

Table 1.10: Medipix3 DACs

1.6.5 Operation Mode Register (OMR)

Operation Mode Register (OMR) is a 48-bit register that controls the chip operation mode. This register must be loaded serially before executing any new operation. The OMR bits positions are shown in Figure 1.13. A description of OMR bits is listed in Table 1.11.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
M0	M1	M2	CRW	Polarity	PS0	PS1	Enable_PT	Enable_TP	CountL0	CountL1	ColumnBlock0	ColumnBlock1	ColumnBlock2	ColumnBlockSel	RowBlock0
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
RowBlock1	RowBlock2	RowBlockSel	EqualizeTHH	ColourMode	EnablePixCom	ShutterCtr	FuseSel0	FuseSel1	FuseSel2	FuseSel3	FuseSel4	FusePulseWidth0	FusePulseWidth1	FusePulseWidth2	FusePulseWidth3
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
FusePulseWidth4	FusePulseWidth5	FusePulseWidth6	FusePulseWidth7	FusePulseWidth8	SenseDAC0	SenseDAC1	SenseDAC2	SenseDAC3	SenseDAC4	ExtDAC0	ExtDAC1	ExtDAC2	ExtDAC3	ExtDAC4	ExtBGSel

Figure 1.13: Medipix3 OMR bits

Name	Description
M0,M1,M2	Operation mode selection
CRW	0: Sequential read-write, 1: Continuous read-write
Polarity	0: Holes collection mode, 1: Electron collection mode
PS	Number of parallel lines used for data
Enable_PT	Not used
CountL	Select the counter depth 0: 2x1-bit, 1: 2x4-bit, 2: 2x12-bit, 3: 1x24-bit
ColumnBlock	Select the matrix column to be read (see 1.6.7)
ColumnBlockSel	0: Whole matrix read-out, 1: Columns selected by ColumnBlock
RowBlock	Select the matrix row to be readout (see 1.6.7)
RowBlockSel	0: Whole matrix read-out, 1: Rows selected by RowBlock
EqualizeTHH	1: Equalize high threshold
ColourMode	1: Colour mode, 0: CSM/SPM
EnablePixCom	1: Enables arbiter operation for matrix equalization in CSM
ShutterCtr	Select shutter signals when CRW
FuseSel	Select Fuse to be burned
FusePulseWidth	Set the pulse width to burn selected e-fuse
SenseDAC	Select DAC to be sensed
ExtDAC	Select DAC to be externally imposed
ExtBGSel	0: Internal band-gap used, 1: External band-bap used

Table 1.11: Medipix3 OMR bits description.

1.6.6 Matrix Load and Read-out

Both operation (reading-out of data and loading pixel matrix configuration) are performed for each counter separately. In the case of loading matrix data into to chip, firstly, one counter in each pixel is loaded with 12 configuration bits, then, the second counter is loaded. Reading-out of matrix is performed in the same way. Even when the counters are set in 24-bit mode the matrix data are read from each counter separately.

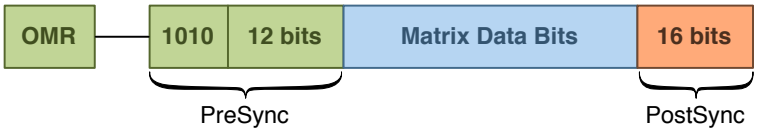


Figure 1.14: Medipix3 matrix load and read-out data format

Every matrix operation is preceded by OMR register with corresponding settings. The OMR is followed by PreSync bits, actual matrix data and PostSync bits (see Figure 1.14). A Number of bits in the matrix data may differ depending on the configured counter depth. A format of the bits in the matrix data is equal to previous generation of Medipix chips. The shift registers in each column are joined together and the data is shifted bit by bit starting with the most bottom row. This is true for loading or reading of the entire matrix. However Medipix3 also allows only read-out of parts of the matrix (Region of Interest).

1.6.7 Region of Interest (ROI)

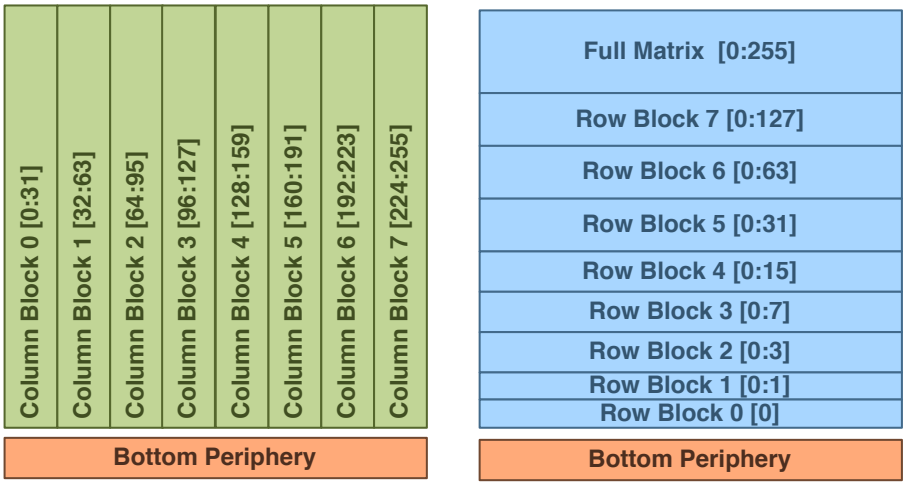


Figure 1.15: Medipix3 Matrix Region of Interest

Only a part of the matrix can be read-out. This is achieved by selecting a column block and/or a row block through OMR bits before the read-out of the data. Figure 1.15 shows the matrix split into column blocks (on the left) and into row blocks (on

the right). The column blocks have minimal width of 32 pixel and can be selected individually while the rows are always read-out from the bottom and have variable row block width.

Column block can be selected with a width of 32, 64, 128 or 256 columns. This selection is done through the *PS*, *ColumnBlockSel* and *ColumnBlock* OMR bits. The number of read-out rows can be set in *RowBlockSel* and *RowBlock* OMR bits.

Both selection modes can be combined. When ROI is selected only pixel data contained in the region are transferred during read-out operation. This has an advantage of less amount of data transfered and smaller transfer time.

1.6.8 Acquisition Modes

Medipix3 implements 3 different acquisition modes. A control of these modes is achieved through *ShutterCtr* and *CRW* OMR bits and LVDS inputs *Shutter* and *Shutter1_CounterSelCRW*.

Full Sequential Mode Acquisition is started (shutter opened) and finished (shutter closed) for both counters simultaneously. The counters measure individually. They must be read-out after shutter is closed and before the next acquisition is started. This produces a certain delay in between (dead-time).

Semi Sequential Mode *Shutter* and *Shutter1_CounterSelCRW* control the acquisition time of both counters. Each counter can be read out independently only when the acquisition for that counter is not-active.

Continuous Read-Write Mode *Shutter* sets the total acquisition time while *Shutter1_CounterSelCRW* controls which counter is read out.

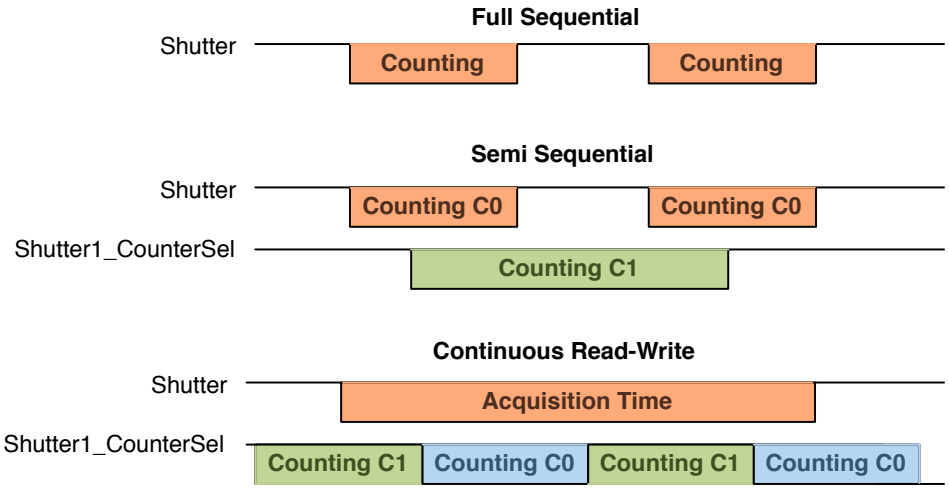


Figure 1.16: Medipix3 Acquisition modes

2 Read-out Interfaces

A read-out Interface is a special dedicated hardware device that reads data and control acquisition of the detector. In this chapter the most frequently used read-out interfaces for Medipix chips will be described. The emphasis will be put on USB Interface and FITPix interface because the development of hardware libraries for these interfaces is a part of this work.

2.1 MUROS

MUROS[5] Interface is the first and original read-out interface for Medipix chips developed by NIKHEF3 institute in Netherlands. Two versions of this interface exist: MUROS-1 (Medipix1 re-Usable Read Out System) and MUROS-2 (Medipix2 re-Usable Read Out System version 2).

2.1.1 MUROS-1

The MUROS-1 interface was developed for the Medipix1 detector. For its proper functionality it needed several special hardware components. The PC controlling the MUROS-1 interface required to have National Instruments (NI) PCI-DIO-32HS board ("NI digital board") plugged in the PCI[1] bus and NI AT-AO-10 board ("NI analog board") plugged in the EISA[2] bus. The software used to control the Medipix1 was Medisoft3[16] developed in the LabWindows also from NI.

The MUROS-1 interface had several disadvantages. Firstly it was depended on several hardware components that had to be bought and installed in the controlling PC. Secondly the configuration and installation of the entire setup was fairly complicated. Lastly the device was not very portable – had to be installed near the PC and 220 V power supply was required.

[1] Personal Computer Interconnected
[2] Extended Industry Standard Architecture

2.1.2 MUROS-2

The MUROS-2 interface is successor of MUROS-1. It is a FPGA[3] based interface between Medipix2 chipboard (that can contain up to 8 Medipix2 chips) and universal data acquisition card (NI DIO-653X) for PCI computer bus.

Figure 2.1: MUROS-2 Read-out interface[5]

In comparison to MUROS-1, MUROS-2 requires only one dedicated PCI card in the controlling PC. But still needs external power supply for the board and for the detector bias. Also due to large dimensions of the device and limited cable lengths the applicability of the detector is restricted.

2.2 USB Interface

The USB interface[6] (see Figure 2.2) is an alternative interface to MUROS2 for Medipix2 data readout and acquisition control. Unlike MUROS, USB interface is connected with PC via widespread USB[4] port. All the necessary detector power sources are integrated in the interface including the detector bias source (up to 100 V). No need for external power supply or any additional hardware is required. All the power supply is taken from the USB connection. To connect the interface to a PC, only a standard USB A-B cable is needed. This as well as small physical dimensions (80 x 50 x 20 mm) allows to achieve maximum portability of the measurement setup.

Another advantage of this interface is a support for back-side pulse processing. The charge generated by an ionizing particle is measured in the bias circuit and can be used for spectroscopic purposes or for triggering. The USB interface is also fully compatible with current chip-boards carrying one or more Medipix2 chips.

[3]Field Programmable Gate Array
[4]Universal Serial BUS

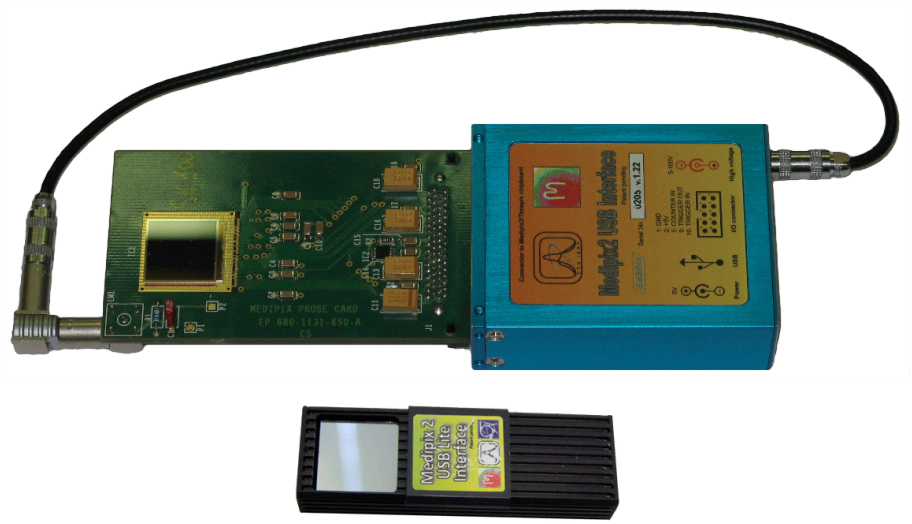


Figure 2.2: On the top USB interface with connected chip-board. On the bottom a smaller version of the interface USB Lite

2.2.1 Hardware

A block diagram of the USB interface is depicted in Figure 2.3). The power supply for the interface is taken from the USB host controller and adjusted in the Power Supplies block. This block contains 2.2 V power source for Medipix2 chip and variable voltage source for Medipix2 detector bias. It powers also the microcontroller (MCU) and other components on the board.

All the communication between the PC and MCU is maintained through FTDI chip (FT245BM). The advantage of this chip is that it handles USB protocol completely. No USB-specific firmware programming is necessary. The other advantage is that FTDI chip drivers for PCs are provided royalty-free by the manufacturer.

The central block of the USB interface is a microcontroller ADuC841 from Analog Devices. The MCU handles all the communication from the PC and controls Medipix2 chip-board and power supply circuits.

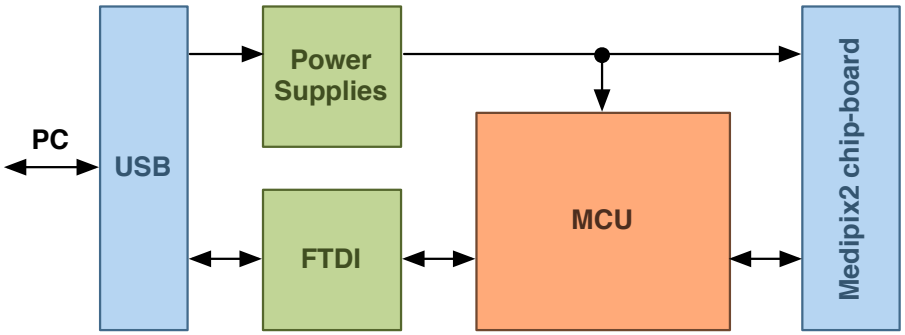


Figure 2.3: USB Interface block diagram

The interface also contains an interface DAC that can impose an external voltage to one of the Medipix/Timepix internal DACs or can set test pulse voltage. The on-board ADCs[5] can read values of detector DACs as well as interface parameters (voltage levels, temperature, etc.).

2.2.2 Software

A firmware for the MCU has a simple structure. After initialization the program jumps into an endless loop and waits for one-byte command received from USB interface. When the command is received it is compared with the command table and an appropriated operation is performed. The communication protocol is described in the next section.

Several versions of the firmware for the USB Interface exist:

- Firmware for Medipix2 1.1.9a
- Firmware for Medipix2 2.0.8a
- Firmware for Medipix3 2.1.1

The original firmware 1.1.9a was used together with the original hardware library that was not multi-platform. The firmware itself was not compatible with Medipix3 detector. Also the communication protocol was highly asynchronous and not very robust against communication failures between Medipix and USB interface or USB interface and PC. Therefore a new robust protocol with better error handling was designed. This protocol is described in the following section.

Since the operating Medipix3 detector differs greatly from operating a previous generation of Medipix detectors, two separate firmwares had to be developed – one for Medipix3 and one for Medipix2 and Timepix detectors. Also a new hardware library with multi-platform support in mind was created for these two firmwares.

Firmware upgrade

To upgrade the firmware in the interface two approaches are possible - using the special serial cable or uploading the new firmware through USB interface. The later takes advantage of one of the ADuC841 MCU features - an ability to modify parts of the program code by the code itself. To perform such a operation, a special firmware (Bootloader) must be present in the device. Then a new firmware can be uploaded from the PC using a special Medipix USB Flash Utility or a Firmware Flasher plugin. This allows users to change the firmware remotely, even when the device is not accessible physically.

[5] Analog Digital Convertors

2.2.3 Communication protocol

The communication protocol is textual and except a few cases synchronous. The PC always initiates the communication by sending a command and the device responds with corresponding reply. The commands are one byte long with optional set of parameters (see Figure 2.4).

Figure 2.4: USB Interface command

When a command is recognized and successfully processed by the device two types of responses can be sent back to PC. If the command requires a binary data (DAC values, matrix data) the response has format as in Figure 2.5a,

Figure 2.5: USB Interface response

where C is the command response is replying to, T is type of the response: 0 – no output data, 1 – synchronous string, 2 – synchronous data, 3 – asynchronous string 4 – asynchronous data and L1, L2, L3 are bytes describing length of the data stream. Response with a text has format as shown in Figure 2.5b, where L1 byte is length of the text.All the responses are terminated with 2 bytes containing text *"OK"* and *CR*[6] byte (0x0D).

When the command execution finishes with an error, the response has a format as shown in Figure 2.6:

Figure 2.6: USB Interface error response

where the first part of the response is similar to successful response, only the *"OK"* bytes are replaced with *"!1"* followed by length of error text and the error text itself. The response is again terminated with *CR* byte.

Most of the command responses are synchronous, i.e. are send only as a reaction to a specific command. The device can send also several asynchronous responses

[6]Carriage Return

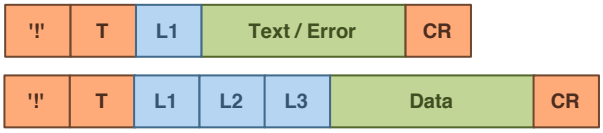


Figure 2.7: USB Interface asynchronous response

reporting error or status of measurement. Their format is shown in Figure 2.7. Instead of the command, first byte contains "!" character and it is followed by a response type (either asynchronous text or data), length of the data and terminated by CR byte. Table 2.1 contains list of main commands supported by the device.

Cmd.	Bytes In	Description	Res. Type	Data Out
_	0	CPU Reset	0	-
i	0	version information	1	version text
?	0	list of all commands	1	commands list
!	0	async. message	3	message/error
<	0	status of measurement	4	1 byte status
x	0	switch Medipix power ON	0	-
X	0	switch Medipix power OFF	0	-
b	1	set Bias	0	-
t	1	set Timepix clock	0	-
o	0	open shutter	0	-
c	0	close shutter	0	-
O	3	open shutter for defin. time	0	-
Q	3	open shutter started bytrigger	0	-
D	1	set detector CMOS Lines	0	-
C	33	set the DACs	0	-
R	0	read the DACs	2	33 bytes
#	0	get number of chips	2	1 byte
T	3	send test pulses	0	-
E	114 721	load matrix	0	-
d	0	clear matrix	0	-
m	0	read matrix	2	64 kB
r	0	finish read matrix	2	rest of data
%	0	get last acq. time	2	time
M	0	read ADC	2	value of ADC
b>	3	set interface DAC	0	-
S	8	set Medipix3 OMR	0	-

Table 2.1: Medipix2 USB Interface commands

2.3 FITPix

FITPix[9] (Fast Interface for TimePix, see Figure 2.8) is a successor of USB Interface. It was created as a response to a need for faster, smaller, flexible and better interface for Timepix. The USB Interface is a flexible device with advanced features. However, the acquisition speed is very slow (up to 5 fps). The MUROS interface is faster but not very portable and requires external power supply. The aim of FITPix was to provide fast, flexible and portable interface for Medipix and Timepix detectors which does not need any additional equipment for its operation.

Figure 2.8: FITPix read-out interface

The FITPix also supports Medipix3 detector. However, a special firmware and minor hardware modifications are necessary. A special dedicated version of the interface is planned for Medipix3 in the future. It will be optimized to exploit fully all the functionality of Medipix3 detectors.

2.3.1 Hardware

The interface is based on FPGA[7]. Simplified block diagram of the device is shown in Figure 2.9. Thanks to the FPGA circuit the device can achieve high data frame-rate. The digital system placed in the FPGA chip is clocked at 50 MHz (system clock). The read-out frequency is two times higher (100 MHz).

The device is connected to PC via USB 2.0 interface. The USB protocol is handled by FTDI FT2232H chip that supports synchronous FIFO mode at frequencies up to 60 MHz, enabling high data-rates.

All the power supply is taken form the USB bus. Nevertheless, it is possible to connect external power supply when needed. The power supply circuits provide several voltages for Timepix (2.2V for analog and digital part separately), FPGA

[7]Field-programmable gate array

(1.2V, 2.2V, 2.5V and 3.3V), FTDI (3.3V) and integrated circuit that generates bias for the sensor (5V).

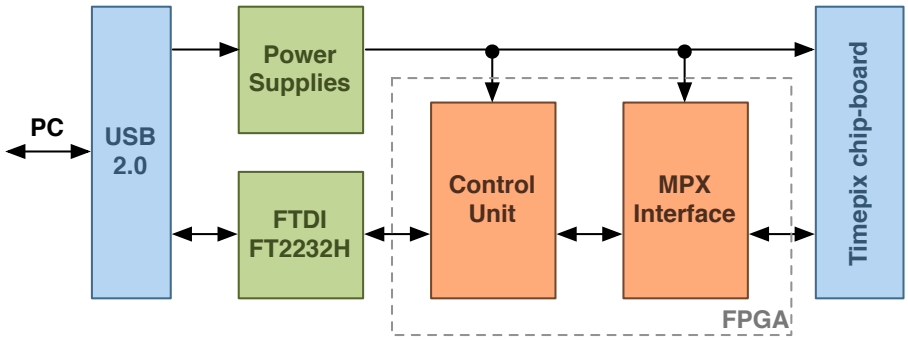


Figure 2.9: Simplified block diagram of FITPIx interface

The maximum read-out speed that device can achieve is 90 frames per second with a single chip (serial read-out frequency of 100 MHz). The FITPix can currently control up to 16 detectors in daisy chain. The interface also features a hardware triggering system.

2.3.2 Firmware

The firmware for the FPGA is divided into two parts: Control Unit and MPX Interface. The Control Units process all the communication with PC via USB 2.0 bus, decodes commands and communicates the information to MPX Interface and other on-board integrated circuits.

The MPX Interface communicates directly with the connected Medipix detectors. It controls all the Medipix signal lines and transfers data streams from and into Medipix. On the start-up of the device it also automatically detects a number of chips that are connected to the device.

2.3.3 Communication Protocol

All the commands and responses are transferred in 6 bytes long frames as shown in Figure 2.10:

Figure 2.10: FITPix protocol frame

where the first byte (start byte) of the frame is always $0x55$ followed by the command code and 3 parameters. The frame is terminated by control check byte for

integrity check. The command code of a response corresponds to received command. The check byte is calculated by applying XOR operation on all 4 bytes of the frame:

$$Check = Cmd \textbf{ xor } Par1 \textbf{ xor } Par2 \textbf{ xor } Par3 \tag{2.1}$$

When the frame is received and recognized by FITPix an acknowledgement frame is sent back to PC with either all 3 parameter bytes set to $0x00$ or corresponding data. When an error occurs (frame corrupted or detector reported error) all the 3 parameter bytes are set to $0xFF$.

All the data streams longer than 3 bytes are transferred between frames and are acknowledged by a dedicated frame. A example of such a communication for *WriteFSR* command is shown in Figure 2.11.

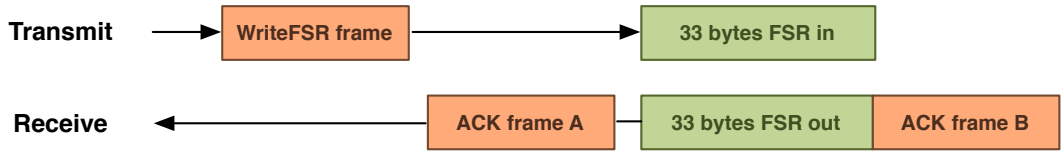


Figure 2.11: FITPix protocol – WriteFSR command

First, PC sends a *WriteFSR* command frame and waits for its acknowledgement: *ACQ frame A*. When it is received the FSR stream from PC is sent and written into Medipix. The validity of the data and their successful writing into the detector are acknowledged by *ACK frame B*.

List of all the commands supported by the FITPix is listed in Table 2.2.

Command	Code	Param.	Command	Code	Param.
Test	0x00	–	Measurement start	0x22	yes
Medipix status	0x02	–	Measurement break	0X23	–
Timer H preload	0x03	yes	Write FSR	0x24	–
Timer L preload	0x04	yes	Reset medipix	0x26	yes
Set DAC	0x05	yes	Test pulse period	0x31	yes
Measurement status	0x06	–	Test pulse start	0x28	–
Set bias	0x07	yes	Test pulse break	0x29	–
Timer pre-divider	0x08	yes	Test pulse status	0x32	–
Signal delay	0x09	yes	Erase matrix	0x27	–
Detector type	0x13	–	Test pulse	0x28	yes
Measurement settings	0x14	yes	System frequency	0x35	–
Counter H	0x16	yes	Measurement frequency	0x36	–
Counter L	0x17	yes	Power	0x41	yes
Trigger delay	0x1A	yes	Read write IO	0x42	yes
Read matrix	0x20	–	Read ADCs	0x40	yes
Write matrix	0x21	–			

Table 2.2: FITPix commands

3 Original Software

This chapter describes an original software package called **Pixelman** that was used to control measurements with Medipix and Timepix detectors. The software architecture[18] is presented together with all the main components. Further, possibilities of extension of the software (adding support for new hardware, extending software functionality) are presented.

3.1 Software Architecture

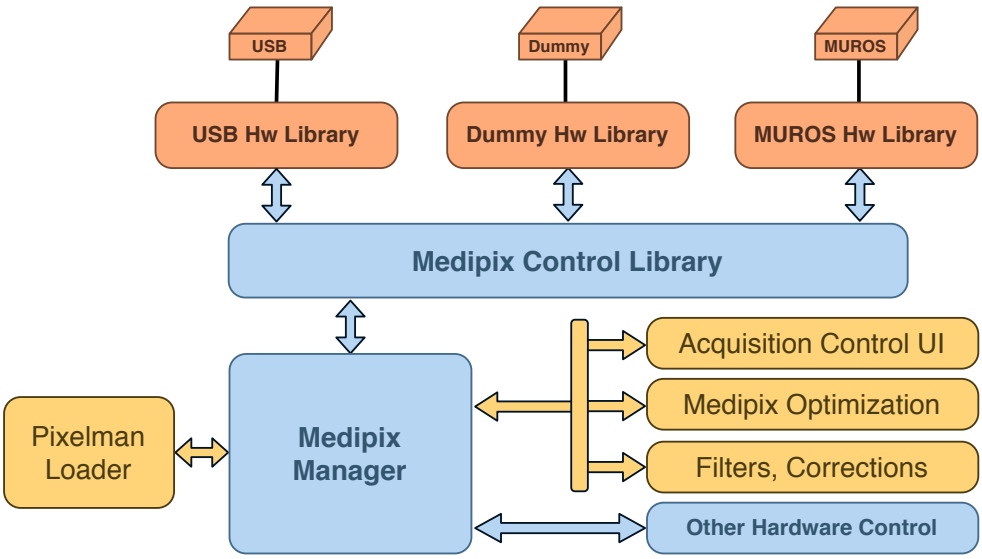


Figure 3.1: Pixelman architecture[18]

Pixelman software package was created as a replacement of Medisoft4[19] software package that was used to control measurements with Medipix1 and Medipix2 detectors. Medisoft4 was written in LabWindows and did not allow any simple extension of its functionality. Only MUROS read-out interface was supported. Moreover, since the software source code was available, many different incompatible versions existed.

Pixelman is designed for maximal flexibility and interoperability with other devices to control complex measurements. This is achieved by a modular architecture

(see Figure 3.1) that can be extended by custom-made modules (plugins). New types of read-out interfaces and hardware can be added by creating an additional hardware library.

3.1.1 Hardware Libraries

Hardware libraries are dynamic linked libraries (dll) that communicates through read-out interfaces with Medipix/Timepix devices. Each hardware library creates a common interface which allows Control Library to access devices in the readout interface independent way.

The original software included 3 hardware libraries: USB hardware library (for the USB Interface), MUROS hardware library (for MUROS2 read-out interface) and Dummy hardware Library. The dummy library is an emulator of Medipix chips and is used for testing and debugging purposes.

3.1.2 Control Library

The Control Library manages all the connected Medipix devices. It provides high level access for different operations. It also manages complete configuration of the devices, provides access synchronization, frame buffer allocation and callbacks for events related to Medipix detectors.

The library is automatically loaded and initialized by the Manager. After initialization it searches for hardwares libraries in a dedicated directory and initializes all the connected Medipix devices that they manage.

The access to each Medipix device through the Control Library is synchronized. This means that when an operation such as writing pixel matrix configuration is running, the device is in a "locked" state to prevent interference from other operations running in other threads. This ensures that each operation on Medipix device is performed uninterrupted.

3.1.3 Manager Library

Manager Library is a core part of Pixelman. It handles and manages the Control Library and all installed plugins. Also, it is responsible for intermediating communication between plugins and the Control Library. The Manager maintains list of frames, registered functions (offered by plugins), events triggered by Medipix devices and plugins, filters and filter chains.

The manager is loaded usually by pixelman loader (pixelman.exe). However it is possible to use it directly in another custom-made application by loading the library and calling the function *mgrInit()*. The library is dependent on the control library and *mgrUtils.dll* library that contains common graphical user interface.

While the Manager is initialized, it loads the Control library, initialize connected

devices and then loads custom-made user plugins. After initialization a icon in the system tray panel is added that contains a popup menu with manager menu items (exit, about, show log dialog) and items registered by plugins. All the main program functionality can be accessed through this menu.

Frames

The manager maintains a list of frames. These are objects that contains matrix data read-out from Medipix devices or created by plugins. Each frame has a unique identification (frameID) that is used by the manager and plugins to manipulate with it. Except from matrix data the frame object contains also frame attributes that describe frame properties such as time of acquisition, used read-out interface, frame format, etc. These attributes are extendable and any plugin can append its own attributes. Access to the frames is synchronized to prevent concurrent updates in a frame.

Frames can be saved to a hard-drive. The matrix data is saved into a file in several different formats:

Ascii matrix the entire matrix is saved as a matrix of ascii numbers with number of rows corresponding to the height of matrix and columns corresponding to the width.

Sparse ascii (X,C) or (X,Y,C) is a format where only pixels that are greater the 0 are saved. It is a very disk space efficient format when a lot of frames with low occupancy is produced. Each line in a file contains position of pixel X (either index position in matrix or x-y coordinate) and the value of the pixel C.

Binary matrix the entire matrix is saved as 16 bits integer binary numbers (depending on the frame format it can be also double, or unsigned 32 bits integer).

Sparse binary (X,C) or (X,Y,C) is similar as sparse ascii, only pixels counting more then zero are saved as binary numbers. X,Y are unsigned 32-bit integer values and the count value C depending on frame format (16-bit integer, 32-bit unsigned integer, double). Numbers are saved in the file one after each other.

The frame attributes are saved into separate file called description file with extension .dsc. Each frame can be saved either into single file (plus description file) or several frames can be saved into a multi-frame file that is more disk space efficient. When more then one frame is saved into single file an index file is created that contains positions of frames in frame file and frame attributes in multi-frame description file.

Functions and filters

Each plugin can export a function that other plugins may use. Each function can have multiple input and output parameters. They are referenced in the manager

library by plugin name and function name.

Plugin can also export one of two special functions - non-adjustable filter function and adjustable filter function. These functions take as a parameter frameID of a frame they modify (filter). Filter functions can perform different operations on the frame. As an example they can change the frame type, or frame size. Example of non-adjustable filter would be filter that converts any frame type to double type. An adjustable filter would be a crop filter, where user can specify crop dimensions.

Users can use filters offered by plugins by creating a filter chain with desired filters. Filter chains are ordered lists of filters that are applied on a frame consequently starting with first filter in the list and ending with the last one. Each filter is applied on a result frame of preceding filter.

3.1.4 Plugins

Plugins are dynamic loading libraries that extend Pixelman functionality. They can contain user interface, control experiment specific hardware, process data or even control the entire experiment. Every plugin has access to Control Library functions and Manager Library functions. Each plugin can add its own functions, call functions of other plugins, create new events or register to be notified when an event occurs.

Every library that is to be recognized as a Pixelman plugin has to export function:

u32 InitMpxPlugin(const FuncTableType* funcTable),

where *u32* is 32-bit unsigned integer (defined in file common.h) and *funcTable* is pointer to a table containing all available functions of Control and Manager Library. Functions from Control Library are prefixed by *mpxCtrl* and functions from Manager Library by prefix *mgr*.

All the constant and type definitions as well as declaration of functions are contained in two header files: *mpxpluginmgrapi.h* and *common.h*. The first header file also contains a macro *PLUGIN_INIT* that simplifies initialization of plugins. The macro declares a global pointer to functions table: *const FuncTableType *mgr* that can be used throughout the implementation (cpp) files. It checks also the version of API of the plugin with the version of Pixelman API to avoid instabilities and crashes.

4 Extension of the Software

The first 3 chapters have been an introduction into Medipix technology and related hardware as well as original software that was used to operate the detectors. The following chapters will describe the upgrade of the software carried out in the frame of this work. A purpose of this chapter is to summarize the reasons that led to the creation of this work and to describe extension of the original software.

4.1 Objectives

Up until recently the latest detectors in Medipix family were Medipix2 MXR and Timepix. The detectors were operated by 2 dedicated read-out interfaces MUROS2 and USB interface and managed from the PC side by Pixelman. Then in 2009 a development of Medipix3 chip was finished and first chips were fabricated. Medipix3 brings an entirely new architecture with new operation modes and features quite different from previous versions of Medipix chips. In the same year, a development of a new fast read-out system FITPix began. Further, operating systems Linux and Mac OS X have been gaining popularity and especially Linux platform is nowadays widely used among scientific community.

All these facts have brought a need for a new software that would fulfill these requirements:

- Medipix3 support
- FITPix support
- Multi-platform support
- New graphical user interface

It was decided that the current software Pixelman will be extended to support these four requirements. The upgrade and extension of the software is the subject of this work.

4.2 Medipix3 support

Medipix3 chip is in many ways quite different from previous chips. It shares the same matrix dimensions (256x256 pixels), but it has two independent counters in each pixel, new operation modes and entirely different way of changing chip configuration (through OMR[1]). The original software supported only Medipix2 chips (Medipix2, Medipix2 MXR and Timepix). The differences between Medipix2 chips were quite minimal - number and positions of DACs and pixel configuration bits. However, For Medipix3 several changes were required in Pixelman. Firstly, representation of matrix data has been changed from *signed integer 16 bit* to *unsigned integer 32 bit* in order to support depth of Medipix3 counter (up to 24 bits). Secondly, the hardware interface was split into *Medipix2* and *Medipix3* as well as many inner objects of Control Library. Lastly, new API functions were added to set Medipix3-only properties. Most importantly though, a new hardware library supporting Medipix3 chip was created. Moreover, Medipix3 has required also a new user interface that would be able to visualize all the new operation modes.

4.3 Hardware libraries

In order to support Medipix3 chip and the new read-out interface FITPix, two hardware libraries were created. Another motivation for a new hardware library was a requirement to use the software also in Linux and Mac OS X operating systems. For the USB interface a hardware library (see 5.2) was created that, according to firmware in the device and connected chip, can control both Medipix2 and Medipix3 chips. For the FITPix interface a hardware library optimized for fast read-out speeds and with auto error recovery features was created (see 5.3). Both the libraries are multi-platform.

4.4 Multi-platform support

The original software was depended on Windows. It was using Windows API for accessing system resources. Also graphical user interface (GUI) was written based on Windows-only framework (MFC[2]). In order to be able to use the software on different operating systems, all the system API calls had to be wrapped into a common interface. This interface wrapped calls to file system functions, thread functions, dynamic library management functions and time functions. Moreover several modification had to be made to compile the software in different C++ compilers. For this purposes a conditional preprocessor directives were used.

However, porting the GUI to other platforms was not possible. It was decided to write a new GUI that would be multi-platform and that would address several

[1] Operation Mode Register
[2] Microsoft Foundation Classes

drawbacks of current GUI. Java programming language and its Swing framework has been chosen as a tool for this purpose. Java has an advantage that only one code has to be written for multiple platforms. Java Swing framework then offers high flexibility and extensibility in designing of GUI. Thanks to this it was possible to create a multi-platform GUI that improved the usability of the software (docking of windows, hiding of panels, etc.). The Java interface is described in more detail in chapter 6.

5 Hardware Libraries

This chapter describes an interface that every hardware library has to implement for Medipix2 or Medipix3 chips in order to be recognized by Pixelman. Moreover implementation and development of two hardware libraries, for USB Interface and for new FITPix Interface, is described.

5.1 Hardware Libraries Interface

The original software supported only Medipix2 and Timepix chips. Therefore every hardware library had to implement an interface for Medipix2/Timepix chips that consisted of exporting several defined functions. With the arrival of Medipix3 chip, which has a different architecture then Medipix2, the hardware library interface was split into two: one for Medipix2 chips and one for Medipix3 chip. One hardware library can export both of these interfaces.

5.1.1 Medipix2 Interface

In order to register Medipix2 hardware interface a library must export function **Mpx2Interface* GetMpx2Interface()**. This function returns a structure that contains pointers to the functions shown in listing 5.2. These functions are defined in header file *mpxhw.h*.

After the library is loaded first function that Pixelman calls is *findDevices()*. This function has to return number of connected devices and fill out the array with device IDs assigned by HW library that are used later to distinguish each device. Afterwards the function *init()* is called for each device to initialize it.

Every device has to fill out a special structure *DevInfo* that contains basic information about the device such as number of chips, number of pixels, number of rows, chipboardID, interface name, etc. This structure is read by Pixelman via *getDevInfo()*.

Besides the DeviceInfo structure each device can provide interface or device specific settings and information through *hardware info items*. These items are read and set by functions: *getHwInfoCount(), getHwInfoFlags(), getHwInfo()* and *setHwInfo()*. Each hardware item is a structure that contains a complete description of

a custom data attribute (data type, name, size, etc.) as shown in listing 5.1.

Listing 5.1: Hardware Info Item

```
typedef struct{
    Data_Types type;
    u32 count;
    u32 flags;
    const char *name;
    const char *descr;
    void *data;
} HwInfoItem;
```

The flag parameter specifies whether the information is configurable by user (MPX_HWINFO_CHANGE), saved into config files (MPX_HWINFO_CFGSAVE) or saved into frames (MPX_HWINFO_MPXFRAME).

Listing 5.2: Medipix2 Interface functions

```
int findDevices(int ids[], int *count);
int init(int hwID);
int closeDevice(int hwID);
int setCallback(HwCallback cb);
int setCallbackData(int hwID, INTPTR data);
int getHwInfoCount();
int getHwInfoFlags(int hwID, int index, u32 *flags);
int getHwInfo(int hwID, int index, HwInfoItem *hwInfo,int *dataSize);
int setHwInfo(int hwID, int index, void *data, int dataSize);
int getDevInfo(int hwID, DevInfo *devInfo);
int reset(int hwID);
int setDACs(int hwID, DACTYPE dacVals[], int size, byte senseChip,
   byte extDacChip, int codes[], u32 tpReg);
int getMpxDacVal(int hwID, int chipNumber, double *value);
int setExtDacVal(int hwID, double value);
int setPixelsCfg(int hwID, byte cfgs[], u32 size);
int setAcqPars(int hwID, AcqParams *pars);
int startAcquisition(int hwID);
int stopAcquisition(int hwID);
int getAcqTime(int hwID, double *time);
int resetMatrix(int hwID);
int readMatrix(int hwID, i16 *buff, u32 buffSize);
int writeMatrix(int hwID, i16 *buff, u32 buffSize);
int sendTestPulses(int hwID, double pulseHeight, double period, u32
   pulseCount);
int isBusy(int hwID, BOOL *busy);
const char* getLastError();
const char* getLastDevError(int hwID);
```

The DAC values are set to device by function *setDACs()*. All DACs are set at once. Value of single DAC can be sensed and read by *getMpxDacVal()*. The pixel configuration matrix is set by function *setPixelsCfg()*. The acquisition is controlled by *setAcqPars()*, *startAcquisition()* and *stopAcquisition()*.

5.1.2 Medipix3 Interface

A hardware library implementing Medipix3 Interface has to export a function **Mpx3Interface* GetMpx3Interface()**. Similarly as for Medipix2 interface this function returns a table of function pointers. They are listed in listing 5.3. There are a few differences compared to Medipix2 interface: acquisition parameters and pixel configurations structures are different and *setDACs()* function does not contains parameters for sensing DAC values. This is done directly in function *senseSignal()*. Medipix3 has also different matrix data type - 32-bit unsigned integer as opposed to 16-bit signed integer in Medipix2.

Listing 5.3: Medipix3 Interface functions

```
int findDevices(int ids[], int *count);
int init(int id);
int closeDevice(int hwID);
int setCallback(HwCallback cb);
int setCallbackData(int id, INTPTR data);
int getHwInfoCount();
int getHwInfoFlags(int id, int index, u32 *flags);
int getHwInfo(int id, int index, HwInfoItem *hwInfo, int *dataSize);
int setHwInfo(int id, int index, void *data, int dataSize);
int getDevInfo(int id, DevInfo *devInfo);
int reset(int id);
int setDACs(int id, DACTYPE dacVals[], int size);
int senseSignal(int id, int chipNumber, int code, double *value);
int setExtDAC(int id, int code, double value);
int setPixelsCfg(int id, Mpx3PixCfg cfgs[], u32 size);
int setAcqPars(int id, Mpx3AcqParams *pars);
int startAcquisition(int id);
int stopAcquisition(int id);
int getAcqTime(int id, double *time);
int resetMatrix(int id);
int readMatrix(int id, u32 *buff, u32 buffSize);
int writeMatrix(int id, u32 *buff, u32 buffSize);
int sendTestPulses(int id, double charge[2], double period, u32
   pulseCount, u32 ctpr[8]);
int isBusy(int id, BOOL *busy);
const char* getLastError();
const char* getLastDevError(int id);
```

5.2 USB 1.0 Hardware Library

The USB 1.0 Hardware library is a dynamic link library that manages communication with USB Interface and connected detector. It supports both Medipix2 detectors (Medipix2, MXR and Timepix) as well as Medipix3 detector. Figure 5.1 shows main building blocks of the library that are described in the following text.

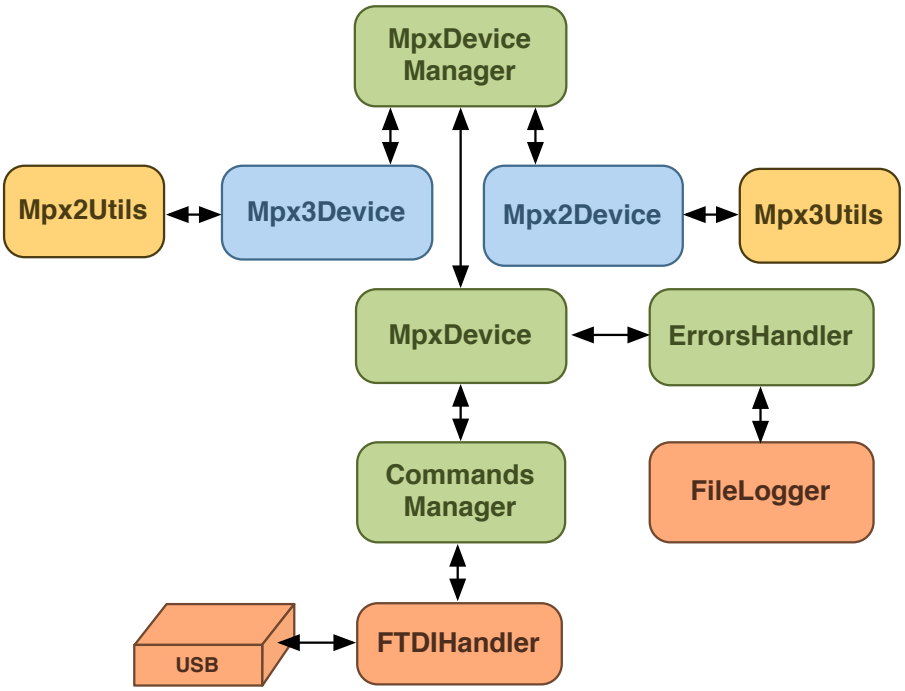


Figure 5.1: USB hardware library architecture

5.2.1 FTDI Handler

The USB device is connected to the PC via USB interface. The USB communication is handled by FTDI chip. In order to facilitate communication on the PC side the manufacturer of the chip (Future Technology Devices International Limited) supplies system drivers for several operating systems: Microsoft Windows, Linux and Mac OS X. There are two alternative versions of FTDI drivers: Virtual COM Port (VCP) and Direct drivers (D2XX). The VCP interface provides a virtual COM port which appears to the system as a legacy COM port. The second interface (D2XX) is provided via proprietary dll (FTD2XX.dll) and allows more direct access to the device that standard system COM port API functions would not allow. For example setting the device into a different mode or writing into device EEPROM. The hardware library for USB interface uses the FTD2XX.dll.

The ftdi library contains several functions for transmitting, receiving data and controlling parameters of USB communication. Each function is prefixed with *FT_*. To get a list of all FTDI devices connected to the PC functions *FT_GetDeviceInfoList()* and *FT_ListDevices()* are used. They return either indexes of devices or their names which are used afterwards for opening the chosen device via *FT_Open()*. This functions returns a device handle *FT_HANDLE* which is used in all the other functions to distinguish different FTDI devices.

The FTDIHandler class represents a wrapper around the functions available in ftdi library. It provides simpler interface than raw FTDI functions and adds new functionality. Listing 5.4 shows methods of FTDIHandler class. The method *listDevices()* enumerates all the connected FTDI devices (including only Medipix2 USB

devices) and returns std::vector of strings containing their names. The device name string is used to open the device through *openDevice()* which returns *FT_HANDLE*.

After opening the device with *initDevice()* method has to be called to set up communication (purge buffers, set timeouts, etc.). This method returns device description and serial number. To check if device is opened *isDeviceConnected()* can be called. The device is closed via *closeDevice()*.

Listing 5.4: FTDIHandler functions

```
vector<string> listDevices();
FT_HANDLE openDevice(string deviceName);
bool initDevice(FT_HANDLE ftHandle, string &deviceSerNo, string &
    deviceDescription);
bool closeDevice(FT_HANDLE ftHandle, bool noMessage);
bool isDeviceConnected(FT_HANDLE ftHandle);

int numBytesToReceive(FT_HANDLE ftHandle);
int receive(FT_HANDLE ftHandle, char* buffer, uint bufferSize, uint
    bytesToReceive);
int receiveUntil(FT_HANDLE ftHandle, char* buffer, uint bufferSize,
    char stopChar);
int receiveSkipUntil(FT_HANDLE ftHandle, char stopChar, double
    recTimeout);
int receiveSkipUntil(FT_HANDLE ftHandle, string stopStr);

int transmit(FT_HANDLE ftHandle, char* buffer, uint bytesToTransmit);
```

The methods *numBytesToReceive(), receive(), receiveUntil()* and *receiveSkipUntil* handle reading of data from device. The first method returns number of bytes that are available to read in receive queue. The *receive()* method is a wrapper around *FT_READ* function. It reads data of fixed length in small bulks (4 kB). Data are read this way in order to avoid filling up of receive queue in FTDI drivers. It was discovered while experimenting with different *FT_READ* function settings that sometimes when the receive queue filled up several data bytes were lost. Therefore it is crucial to read the data as quick as possible to avoid losing any bytes. The *receive* method also adds possibility to specify maximal time to wait for data (timeout) and can log received data or errors into a log file.

The *receiveUntil()* method reads data from the device until a stopChar character is received. The *receiveSkipUntil()* method is similar, but skips all the data until *stopChar* or *stopString* is read. Sending data to the device is handled by *transmit()* method which uses internally *FT_Write()* function.

5.2.2 Medipix Commands Manager

The Medipix Commands Manager class is responsible for parsing of the communication protocol between hardware library and USB interface. It formats output data, sends commands to the device and receives and parses data from the device via FTDIHandler. The communication protocol is described in 2.2.3. The listing 5.5 shows main methods of the class.

Listing 5.5: MpxCommandsManager functions

```
int sendCommand(CommandInfoType commandInfo,char* data,uint dataSize);
int sendCommandAndWaitForResponse(CommandInfoType& command,char* data,
    uint dataLength, ResponseInfoType &response, double waitTime,
   uint retries = 0);
int sendStringAndWaitForString(const char* text, const char*
   textToWaitFor, double delayBetweenTxRx);

int fetchSingleResponse(ResponseInfoType &response);
int waitForResponse(ResponseInfoType &response, double waitTime);
int waitForSpecificResponse(ResponseInfoType &response, const char
   command, double waitTime);

int fetchCommandData(ResponseInfoType &response, uint dataLength);
int fetchErrorMessage(ResponseInfoType &response);
int reportErrorAndResync(int returnCode, const char* errMessage);

int synchronization();
void backgroundMessagesMonitorThreadFunc();
```

Commands are sent via *sendCommand()* method. It accepts as a parameter CommandInfoType structure that contains the actual one-byte command character and its description. The description is used for logging purposes. However more useful method is *sendCommandAndWaitForResponse()* which sends command and then waits for its response. The device response is parsed and saved into *ResponseInfoType* class (see Listing 5.6).

Listing 5.6: ResponseInfoType

```
class ResponseInfoType{
    char command;                    // one-byte command
    const char* commandDescription; // text description
    char*  data;                     // command data
    uint dataSize;                   // size of data (0=no data)
    bool backResponse;               // it is background message
    bool errorOccured;               // device reported an error
}
```

To receive data (get response) from device method *fetchSingleResponse()* is used. It reads any response: command response, asynchronous background message, error message and fills ResponseInfoType class accordingly. It is possible to wait for a response *waitForResponse()* or wait for response to certain command *waitForSpecificResponse()*.

When an execution of a command fails and an error is returned from the device, response flag *errorOccured* is set and data contains error message. The error is also reported to MpxErrorsHandler (see 5.2.7) class and logged into a file. When an communication error occurs (data partially received, invalid response from device, ...), the error is reported to MpxErrorsHandler and the library tries to resynchronize communication (*synchronization()*). The synchronization means that the library sends a synchronization command. The device responds with defined synchronization string. The library skips all the data until the synchronization string

is detected. The synchronization is necessary to make sure that device and software are in defined state.

Responses from device can be both synchronous and asynchronous. The majority of responses are synchronous (responses to command). However there are a few asynchronous background responses informing about acquisition finish, trigger event, error message, etc. To fetch these responses a separate background thread (*backgroundMessagesMonitorThreadFunc()*) is running that is repeatedly checking (calling fetchSingleResponse()) for new responses. If the response is asynchronous it is processed as background message. If it is a response to command it is ignored by background thread and left for corresponding *waitForSpecificResponse()* method to process.

5.2.3 Medipix Device

The Medipix Device is an abstract class that represents common properties of Medipix2 and Medipix3 detectors and parameters of the USB interface. It is the base class for Medipix2 Device and Medipix3 Device. Medipix Device communicates with the device through MpxCommandsManager. It is separated from the actual communication protocol. This way this class could be also used with other interfaces than FTDI just by changing MpxCommandsManager and FTDIHandler. The Listing 5.7 shows main methods of Medipix Device class.

The majority of the methods are connected with the USB interface and are common for Medipix2 and Medipix3. The *initDevice()* method opens the device, sets up MpxCommandsManager, fills out device information structures, reads chipID (*readsChipInfo()*) and initializes USB interface. Methods *getADCs()* reads analog values of chip or usb interface DACs. *setInterfaceDAC()* together with *setExtDAC()* sets value of external voltage that can be applied to one of the Medipix/Timepix DACs. *openShutter()*, *openShutterForTime()* and *closeShutter()* control chip shutter and that way acquisition time.

Some methods are only virtual and are overridden in the subclass. An example of such methods are *startAcquisition()* and *stopAcquisition()*.

Listing 5.7: Medipix Device methods

```
int initDevice();
int closeDevice();
int globalReset();
int resetDevice();
string getVersionInfo();
int openShutter();
int openShutterForTime(double time, bool triggered);
int closeShutter();
string getChipID();
int readChipInfo(int &chipType, int &chipCount, string &chipID);
string getCommandsList();
int medipixPower(bool powerOn);
int setBias(double voltage);
int setExtDAC(char index, double value) = 0;
```

```
int watchdogTest();
int synchronization();
int getADCs(ushort ADC[13], uint chipNumber);
int getADCs(double ADC[13], uint chipNumber);
int getADCsForHwParams(uint chipNumber, double validTime);
int setInterfaceDAC(uchar index, ushort value);
int setInterfaceDAC(uchar index, double value);
int startAcquisition();
int stopAcquisition();
int clearMatrix();
```

5.2.4 Medipix2 Device

The Medipix2 Device class implements specific functionality for Medipix2 and Timepix chips. The most important methods are shown in Listing 5.8. The method *setDACs()* receives as input parameter array of DAC values. It encodes DAC value into a data stream that Medipix/Timepix chip understands and sends it to the device. *sendTestPulses()* is used for testing the chip functionality. It sends defined number of test pulses with defined width and height to the device and after the procedure is finished, matrix data is read-out. *setTimepixClock()* method sets frequency of USB interface clock to perform measurements with Timepix chip in ToT, Timepix or Timepix 1hit mode.

Listing 5.8: Medipix2 Device methods

```
int setDACs(ushort DACs[], uint size, byte senseChip, byte extDacChip,
    int codes[], uint tpReg);
int senseDAC(uchar index, uint chipNumber, double* value);
int setExtDAC(char index, double value);
int sendTestPulses(uint count, uchar width, double pulseHeight);
int readMatrix(short matrix[], uint size);
int writeMatrix(short matrix[], uint size, bool convertToPseudo);
int setPixelCfgs(byte cfgs[], uint size);
int setHwInfo(uint index, HwInfoItem* item);
int setTimepixClock(char clock);
```

The method *readMatrix()* reads matrix data from the device. The matrix is not read-out in one part, but is read in bulks of 64 kB. This size was chosen to optimize data transfer through older FTDI drivers that have internal 64 kB buffer. This way the buffer overflow is avoided and no bytes are lost. Since the USB Interface microprocessor is not very fast a few milliseconds of commands exchange does not have big influence on total read-out speed, but the stability of transfer is ensured.

The matrix is read-out from the device in raw format. That means it is deserialized and each pixel contains value of pseudo-random counter corresponding to number of hits. There for the data stream from device must be first deserialized and then pseudo-random counts are converted to proper values. Functions for stream deserialization and conversion are grouped in the file *Mpx2Utils.cpp*.

The *writeMatrix()* method writes matrix data into the device. It is possible to select whether the pixel values are to be converted to pseudo-random counts or not. The values are not converted when the function is used to write pixel configuration in method *setPixelCfgs()*. This method puts each configuration bit (mask bit, test bit, ...) into their proper position in data stream according to chip type.

5.2.5 Medipix3 Device

The Medipix3 Device class implements Medipix3 specific functions. Listing 5.9 shows some methods of this class. First 7 methods are similar to Medipix2Device. However these methods have different implementation inside, specific to Medipix3. All the settings of chip is handled through *Operation Mode Register (OMR)*, which is represented in the class as OMR struct (see Listing 5.10). OMR is send via *sendOMR()* method, which encodes OMR structure into data stream readable by Medipix3 chip.

Listing 5.9: Medipix3 Device methods

```
int setDACs(ushort DACs[], uint size);
int getDACs(ushort DACs[], uint size);
int setExtDAC(char index, double value);
int sendTestPulses(uint count, uchar width, double chargeTPA, double
    chargeTPB, uint CTPR[]);
int setPixelCfgs(Mpx3PixCfg cfgs[], uint size);
int startAcquisition();
int stopAcquisition();

int senseDAC(uchar index, uint chipNumber, double* value);
int setCTPR(uint CTPR[], uint size);
int sendPins();
int sendOMR();
int writeMatrix(uint matrix[], uint size, uchar counterSelect, bool
    doSendOMR = true);
int writeMatrix(uint matrix[], uint size);
int readMatrix(uint matrix[], uint size);
int readPeriphery(char data[], uint &size);
```

Listing 5.10: Medipix3 Device OMR

```
struct OMR {
  uchar m;
  bool  crw;
  bool  polarity;
  uchar ps;
  bool  enablePT;
  bool  enableTP;
  uchar count;
  uchar columnBlock;
  bool  columnBlockSel;
  uchar rowBlock;
  bool  rowBlockSel;
  bool  equalizeTHH;
```

```
  bool  colourMode;
  bool  enablePixelCom;
  bool  shutterCtr;
  uchar fuseSel;
  uint  fusePulseWidth;
  uchar senseDAC;
  uchar extDAC;
  bool  extBGSel;
};
```

Writing matrix data to the device is done by *writeMatrix()*. As opposed to Medipix2, Medipix3 has two separate counters and therefore writing matrix or pixel configuration has to be done for each counter individually. Reading of matrix is similar, each counter must be read separately. However, which counter will be read depends on acquisition mode. When counters work separately each counter can be read individually as 12-bit matrix. When Medipix3 is set in 24-bit mode, first counter1 is read then counter0 and they are combined into one matrix of 24 bits.

When matrix is read-out, a raw data stream is received from device. Data must be deserialized. Unlike Medipix2/Timepix, counter in Medipix3 is not pseudo-random but binary. Thus it is not necessary to convert pixel values. However, thanks to the possibility to read only part of the matrix (region of interest) deserialization of the data is more complex than for Medipix2. The helper functions for deserialization and stream conversions are in *Mpx3Utils.cpp* file.

The first version of Medipix3 chip shows a temperature instabilities in *FBK* and *CAS* DAC values. Therefore a special optimizing method is used to find optimal values of these DACs. How often the optimization should be performed is adjustable in hardware specific info items. This optimization senses DAC values from chip (reads their analog values - voltage) and adjust them until they equal desired voltage.

5.2.6 Medipix Device Manager

The Medipix Device Manager takes care of enumerating and managing all connected Medipix devices. Listing 5.11 shows main methods. When the hardware library is loaded Pixelman asks for a list of device IDs. The list of devices is created by *listDevices()* method and IDs are returned via *getDevicesIDs()*. The parameter *deviceType* specifies which device type (Medipix2 or Medipix3) should be enumerated into the IDs list.

Listing 5.11: Medipix Device Manager methods

```
int listDevices();
int getDevicesIDs(int ids[], int *count, MpxDeviceType deviceType);
MpxDevice* getDevice(uint id);
MpxDevice* getDeviceByName(string deviceName);
vector<string> getDevicesNames();
MpxDevice* getFirstDevice(MpxDeviceType deviceType);
int getFirst(int* index, MpxDeviceType deviceType);
int getNext(int* index, MpxDeviceType deviceType);
```

5.2.7 Medipix Errors Handler and File Logger

Medipix Errors Handler

The Medipix Errors Handler class is responsible for error handling and propagating. The *errorMessage()* method offers convenient way of reporting formatted messages with variable number of parameters:

```
int errorMessage(MPX_ErrType Error, const char* errortxt, ...);
```

First parameter is type of error message, the second is format string of message followed by optional parameters. It is similar to *printf* function. The last error message is always saved and can be obtain by calling *getErrorInfo()* method. Pixelman uses this method to get the last error message from HW library, when a command fails execution and returns an error code. Errors Handler allows also to register a callback function that is called when an error is reported. This is used by File Logger class to log errors into a log file.

File Logger

File Logger class logs data streams, device responses, commands, error and debug messages into a log file. All the logging methods are shown in the listing 5.12.

Listing 5.12: File Logger

```
void logMsg(string message, FileLoggerMsgType type);
void logMsg(const char* message, FileLoggerMsgType type);
void logStream(char* data, unsigned int datalen, bool transmit);
void logResponse(char command,const char* commandDescription, char*
    data, unsigned int datalen, bool transmit);
void logDataStream(char* data, unsigned int datalen, bool transmit);
void logError(string message, const char* errorTypeDescription, bool
    justMessage);
void logCommand(char command, const char* commandDescription);
```

Below is an excerpt from a real log file demonstrating capability of logging command and data stream:

```
COMMAND  (12:32:27.952): Command: "i" [69] (Version info)
RECEIEVED(12:32:27.953): Command: "i" <<<
   4d 65 64 69 70 69 78 32 20 55 53 42 20 49 74 65 |Medipix2.USB.Ite
   72 66 61 63 65 0a 56 65 72 73 69 6f 6e 3a 20 32 |rface.Version:.2
   2e 30 2e 38 61 0d                               |.0.8a.
   >>Bytes transfered: 38
MESSAGE  (12:32:27.953): [Message]: Command Version info performed OK.
```

5.3 FITPix Hardware Library

FITPix Hardware Library is a control library for FITPix read-out interface. FITPix is an alternative interface to USB Interface or MUROS2. Like the USB interface it is connected to the PC via USB using FTDI chip, but it has significantly faster frame rate - up to 90 frames per second compared to 5 frames per second in a case of USB Interface. However, the communication protocol is entirely different.

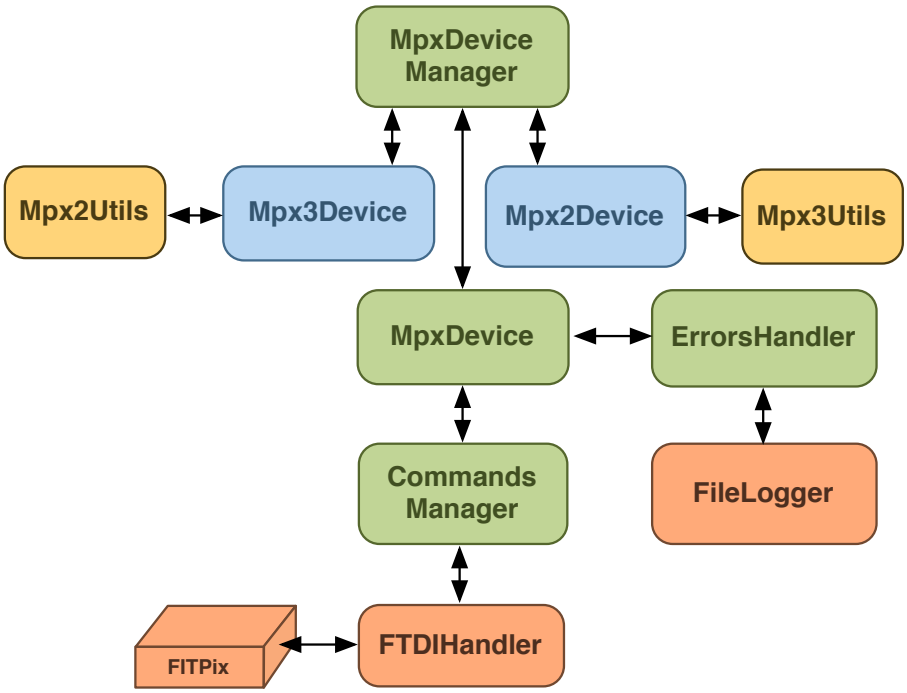


Figure 5.2: FITPix Library Architecture

The architecture of FITPix library (see Figure 5.2) is very similar to USB Hardware Library, which was taken as a basis when developing FITPix library. Since the two libraries are so similar only the differences and FITPix specific features will be described in this section

5.3.1 FTDIHandler

The majority of the methods are the same as for the FTDIHandler in USB hw library (see Listing 5.13). However methods *receiveUntil()* and *receiveSkipUntil()* are not needed for FITPix interface, because length of data streams is known beforehand. The other difference is the initialization of the device. The USB interface is operated by microcontroller that has build in watchdog. In an event that the interface is not getting any response from the Medipix chip and is stuck, the microcontroller is automatically restarted. Also when a communication error between the interface and PC occurs the hardware library can restart the microcontroller by sending an appropriate command. The FITPix interface is, however, FPGA based

and does not implement a command for restart. The only way to restart the device is to switch Bit Bang modes. FITPix is operating in the synchronous bit bang mode. When the mode is switched into asynchronous the device is restarted. Therefore in the initialization method *initDevice()* the device is restarted by switch to asynchronous and then back to synchronous mode.

Listing 5.13: FTDI Handler methods

```
std::vector<std::string> listDevices(int deviceType);
FT_HANDLE openDevice(std::string deviceName, bool checkDriverVer);
bool initDevice(FT_HANDLE ftHandle, std::string &deviceSerNo, std::
    string &deviceDescription);
bool closeDevice(FT_HANDLE ftHandle, bool noMessage = false);
bool isDeviceConnected(FT_HANDLE ftHandle);

int numBytesToReceive(FT_HANDLE ftHandle);
int receive(FT_HANDLE ftHandle, char* buffer, uint bufferSize, uint
    bytesToReceive, double recTimeout = -1, double fullTimeout = -1);
int transmit(FT_HANDLE ftHandle, char* buffer, uint bytesToTransmit);
bool checkDriverVersion(FT_HANDLE ftHandle, bool showMsg);
```

During a testing of FITPix device at higher transfer speeds several stability issues were encountered. It was determined that it was caused by bug in FTDI drivers. Several different versions of FTDI drivers were tested with these results:

Drivers 2.04.16 do not support USB 2.0 FT2232H chip

Drivers 2.06.02 were working properly, but were not WHQL certified and in Windows 7 were replaced by a newer version.

Drivers 2.08.02 had serious instabilities while reading data

Drivers 2.08.08 had serious problem with locking up the computer when a FTDI device was physically disconnected from PC

Drivers 2.08.12 are working properly

Drivers 2.08.14 are working properly and are WHQL certified

In order to avoid instabilities the method *checkDriverVersion()* was added into the FTDIHandler. When the device is initialized, the drivers version is checked and the user is notified if a wrong version of drivers is installed.

5.3.2 Medipix Commands Manager

Medipix Commands Manager handles, similarly as in the case of USB interface, communication protocol (described in 2.3.3) and creates an abstract layer between MedipixDevice and FTDIHandler. A list of main methods is shown in listing 5.14. The device is opened by *openDevice()* method and closed by *closeDevice()*. The connection status can be determined by *isConnected()* or *isResponding()* methods.

Listing 5.14: Medipix Commands Manager

```
int openDevice(string &devSerNo, string &devDescription, bool
    startBackMsgMonitor = true, bool checkDriverVersion = true);
int closeDevice();
int reconnectDevice();
bool isConnected();
bool isResponding();
int sendCommand(CommandInfoType commandInfo, char params[3]);
int sendData(CommandInfoType commandInfo, char* data, uint dataSize);
int fetchSingleResponse(ResponseInfoType &response, bool background);
int fetchCommandData(char** data, uint &dataLength);
int fetchCommandDataAndAck(ResponseInfoType &response,uint dataLength,
     CommandInfoType &command);
int waitForResponse(ResponseInfoType &response, const char command,
    double waitTime);
int sendCommandAndWaitForResponse(CommandInfoType& command, char
    params[3], char* data, uint dataLength,ResponseInfoType &response,
     uint recDataLength, double waitTime, uint retries);
void backgroundMessagesMonitorThreadFunc();
```

All the control data (commands and their responses) are sent in 6 bytes long frames. Data streams such as matrix data or DACs stream are sent between two frames. *sendCommand()* method takes as argument *commandInfo* structure containing one byte command code and its description. The second argument are 3 one-byte command parameters. *fetchSingleResponse()* reads single frame, *fetchCommandData()* and *fetchCommandDataAndAck()* read data stream such as matrix data. The method *sendCommandAndWaitForResponse()* combines several other methods. It sends command with parameters and optional data stream, waits for command response and reads, if desired, data stream from device.

Similarly as in the case of USB library, there is a background messages thread that periodically checks for incoming frames and process any background messages. There are currently two background/status messages: Measurement Status and Test Pulse Status.

5.3.3 Medipix Device

The Medipix Device contains methods common to Medipix2 and Medipix3, and methods related to FITPix interface. When the device is initialized it reads firmware version, system frequency and measurement frequency. The first is frequency of the FITPix read-out interface, the second one is frequency from which external clock signal for Timepix detector can be obtained. The Medipix Device class sets also bias voltage and reads analog values of chip DACs.

5.3.4 Medipix2 Device

The Medipix2 Device is the core class of FITPix library. It handles Medipix2 and Timepix chips. The principal methods are shown in Listing 5.15. The major-

ity of methods work in a similar way as in USB library. However, acquisition and matrix read-out methods are more complex. Two mechanisms were implemented to reach maximal read-out speed and improve stability: burst mode and automatic communication error recovery.

Listing 5.15: Medipix2 Device methods

```
int initDevice();
string getChipID();
int readChipInfo();
int eraseMatrix();
int setDACs(ushort DACs[], uint size, byte senseChip, byte extDacChip,
    int codes[], uint tpReg, char chipIDStream[4]);
int senseDAC(uchar index, uint chipNumber, double* value);
int setExtDAC(char index, double value);
int sendTestPulses(uint count, double period, double pulseHeight);
int readMatrix(short matrix[], uint size);
int writeMatrix(short matrix[], uint size, bool randomize);
int setPixelCfgs(byte cfgs[], uint size);
int setHwInfo(uint index, HwInfoItem* item);
int setTimepixClock(double &clock);
int setSignalDelay(u16 &time);
int setTriggerDelay(double &time);
int readMatrixToBuffer();
int startBurstAcquisition(int acqCount);
int correctionAcqAndMatrixRead(ResponseInfoType &response, uint
   streamLen);
```

Burst mode

With high speed data transfers (up to 90 frames per seconds) every milliseconds counts. When using FTDI drivers each data exchange (sending data and receiving response) has minimal delay of about 2 ms. For standard acquisition of one frame several commands are exchanged:

```
(00:25:02.402): CMD [0x22] Measurement Start
(00:25:02.404): CMD [0x22] Measurement Start OK
------------------------------------------------------------
(00:25:02.407): RX  [0x06] [22|22|22] Meas. Status (Started)  <<<
(00:25:02.407): RX  [0x06] [44|44|44] Meas. Status (Finished) <<<
------------------------------------------------------------
(00:25:02.407): CMD [0x20] Read Matrix
(00:25:02.419): CMD [0x20] Read Matrix OK
```

It is clear that in order to start a measurement, 2 to 5 ms are lost during communication. This significantly decreases frame rate (down to about 60 frames/s). To increase transfer speed a burst mode was implemented. In the burst mode before starting measurement a number of frames to be acquired is sent to the device. Afterwards the device is performing acquisition of each frames automatically. It notifies the PC about finished acquisition and waits until the matrix data are read-out to start the acquisition again. The command exchange is reduced to receiving events and reading of the data:

```
(17:56:57.400): RX  [0x06] [22|22|22] Meas. Status (Started)  <<<
(17:56:57.400): RX  [0x06] [44|44|44] Meas. Status (Finsihed) <<<
-------------------------------------------------------------
(17:56:57.400): CMD [0x20] Read Matrix
(17:56:57.411): CMD [0x20] Read Matrix OK
-------------------------------------------------------------
(17:56:57.411): RX  [0x06] [22|22|22] Meas. Status (Started)  <<<
(17:56:57.411): RX  [0x06] [44|44|44] Meas. Status (Finished) <<<
-------------------------------------------------------------
(17:56:57.411): CMD [0x20] Read Matrix
(17:56:57.422): CMD [0x20] Read Matrix OK
-------------------------------------------------------------
(17:56:57.422): RX  [0x06] [22|22|22] Meas. Status (Started)  <<<
(17:56:57.422): RX  [0x06] [44|44|44] Meas. Status (Finished) <<<
-------------------------------------------------------------
(17:56:57.422): CMD [0x20] Read Matrix
(17:56:57.433): CMD [0x20] Read Matrix OK
```

It is clearly visible that transfer speed has increased significantly. Another limiting factor reducing speed was frame processing in Pixelman (deserialization, derandomization, etc.). The solution was to do pre-caching of frames in hardware library. When an event *acquisition finished* is received, matrix is read-out immediately to start acquisition of new frame. The frame is process in parallel thread and transferred to Pixelman when demanded. The next frame is read-out when the previous one is passed to Pixelman. This has also a positive impact on transfer speed. The burst algorithm diagram is shown in Figure 5.3

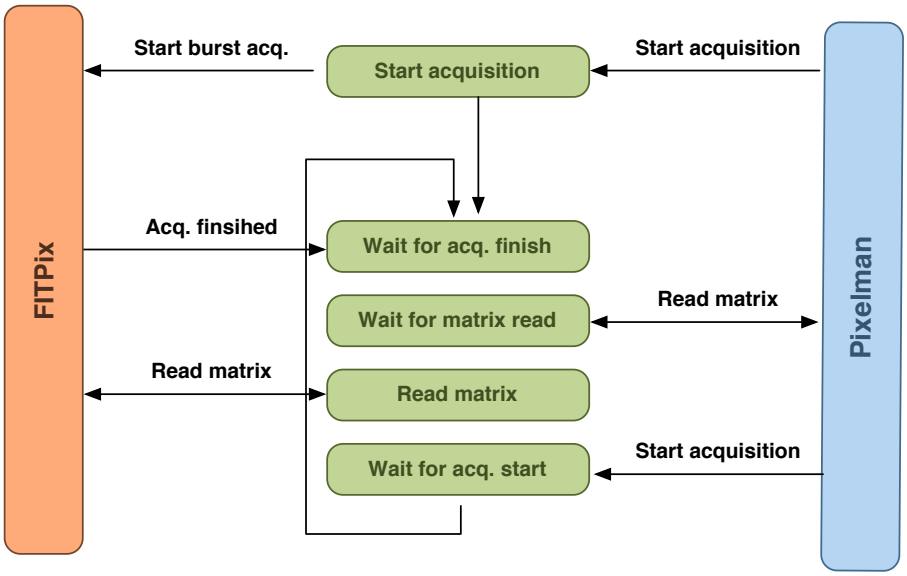


Figure 5.3: Burst algorithm diagram

Automatic Error Recovery

With higher transfer speeds a probability of communication error increases. During the testing of the device sometimes a matrix data were received incomplete. This

was handled by the library, error was reported to the user and measurement was stopped. This behavior is, however, not desirable when longer measurements are performed and user is not present at the PC. When error occurs the measurement is stopped and until user reacts no data are taken. In order to avoid such situations a special automatic auto error recovery mechanism was implemented in FITPix hardware library.

When the matrix data are not read-out completely or are corrupted, the hardware library starts automatically new acquisition and tries to read data again. It is possible to choose how many times error recovery should be attempted. The auto error recovery functionality is available both for normal acquisition mode and for burst mode. There is also an option to return empty matrix when an error occurs. This is useful when a coincidence measurement is performed with several FITPix read-outs and data integrity is required.

5.3.5 Medipix3 Device

The Medipix3 chip support in FITPix is limited. FITPix read-out interface was design mainly for Medipix2 and Timepix chips. In order to use FITPix with Medipix3 chip few hardware modifications are necessary. FITPix hardware library was developed to support Medipix3 chips and was tested with modified version of FITPix interface.

The Medipix3 support is not optimal and does not make use of full potential of FITPix device. However for the future, a dedicated version of FITPix interface for Medipix3 is planned.

5.4 Installing and Using FTDI Drivers

5.4.1 Windows

The FTDI drivers for Microsoft Windows are distributed as single executable file. This installer installs system driver and FTD2XX.dll library into the system. As it was mentioned above some versions of FTDI drivers have shown instabilities. In order to avoid this, compatible FTDI drivers are supplied with Pixelman.

5.4.2 MAC OS X

The default FTDI drivers for MAC OS X downloadable from manufacture web page do not contain any installer. They have to be manually copied into several directories. When newer drivers are installed, older version must be first manually removed. This is not very convenient for the end users. Therefore a standard MAC OS X installer package was created that automatically installs drivers and removes their older version from system.

5.4.3 Linux

The FTDI drivers for Linux do not have installer and have to be copied manually into the system as well. Moreover for using the drivers under user different than root, special permissions have to be set for the FTDI device. To avoid this complex installation the start up script of Pixelman checks if the drivers are installed in the system and in a case they are not, the script installs them and sets up corresponding permissions. Linux drivers are also supplied with Pixelman.

6 Java Interface

One of the goals of this work was to extend the original software Pixelman so that it would be possible to operate it on different platforms: Windows, Linux and Mac OS X. The core of Pixelman is written in C++ without any GUI. Only parts of the software using platform-specific functions such as filesystem or thread functions had to be modified in order to be multi-platform. Modifying plugins that were extending Pixelman functionality was more complicated, because they possessed Windows specific GUI written in MFC (Microsoft Foundation Class). This GUI could not be ported to other platform. Moreover most of the plugins used Windows specific functions. Therefore it was decided to look for a way to create one common GUI that could be used on all three platforms. Several different libraries, frameworks and programming languages were considered. On of the main conditions was possibility to interface Pixelman C++ core with a particular language or framework. The second condition was to facilitate development of GUI as much as possible.

In the end Java programming language and its Swing framework were chosen. Java allows interfacing C++ code with Java applications. It is a very flexible, robust object oriented language. It is the most popular programming language in use today. There are many frameworks and libraries available for it that simplifies developments of applications. Java is by its nature also a platform independent language. That means that only one code has to be written for all three platforms.

The Swing framework is a very flexible and extensible API for providing a GUI that can be use on multiple platforms. One of the big advantages is that it can use system native look and feel. Another advantage is that the design of GUI can be easily changed during runtime of application, thus making it easy to perform advanced GUI features like docking of windows or changing look of components.

This chapter will describe a Java interface for C++ core of Pixelman. Different approaches of interfacing C++ language with java language will be compared and chosen approach will be described in detail. A bridge between Pixelman C++ core and Java interface will be presented. Lastly java Graphical User Interface (GUI) and Java plugins will be described.

6.1 Interfacing Java with C++ Core

6.1.1 Different approaches

There are several possible ways to interface native C/C++ libraries with Java. They differ in performance and ease of use. In order to select a suitable framework, it must fulfill certain requirements. First it must be fast enough. Secondly it has to be able to call function from java and vice versa. The most popular libraries that can be used in Java to interface C++ are described below.

Java Native Interface (JNI)

JNI is a programming framework that allows Java code to call and to be called by native applications and libraries written in other languages such as C/C++. In order to use JNI, a special wrapper functions have to be created for every C/C++ function that is to be accessible by Java. JNI is the most low-level framework to access C/C++ from Java. Both JNA and SWIG use it internally.

Java Native Access (JNA)

JNA provides Java applications easy access to native libraries without need to create any boilerplate or generated glue code. Native libraries can be used in non-modified form. It is possible to call both C/C++ code from Java and Java code from C/C++. However JNA performs slower than pure JNI code. According to tests the call overhead of JNA is about one order of magnitude greater than JNI. This makes JNA not very suitable library for bridging Java and Pixelman core.

Simplified Wrapper and Interface Generator (SWIG)

SWIG is a software development tool that connects programs written in C or C++ with multiple high-level programming languages. SWIG can generates automatically glue code for C/C++ interfaces. To generate a C++ wrapper SWIG expects a file containing C/C++ declarations and special SWIG directives. This is a special SWIG interface file .i or .swg. Output of SWIG is a C/C++ wrapper file and a file in target language, in this case Java. Calling a C/C++ function is straightforward with SWIG. However registering a callback from C++ to Java is complicated and must be written manually.

Comparing all the 3 possible methods, JNI framework was chosen as the best. It is the fastest and most flexible one of all these 3 frameworks.

6.1.2 Java Native Interface (JNI)

JNI allows to call C/C++ functions from Java application. Native functions are declared in Java with keyword *native* without function body. For instance function in class *Test* would be written:

```
public int native testFunc(String text);
```

In C++ this function would be translated into JNI wrapper function like this:

```
JNIEXPORT jint JNICALL Java_Test_testFunc(JNIEnv* env, jobject obj,
    jstring text)
{
  // ...
  return 0;
}
```

When this function is invoked by JVM[1] a JNIEnv pointer and jobject pointer are passed to the function along with any other Java arguments declared by the Java method. The env pointer is a structure that contains an interface to JVM. It contains all the necessary functions to work with Java objects such as accessing object's members, converting native arrays to/from Java arrays, converting native strings to/from Java strings, instantiating objects, etc. The parameter obj refers to the current object.

JNI defines a set of C/C++ data types that correspond to primitive and reference types in Java. List of data types mapping is shown in Table 6.1. Table 6.2 contains JNI mapping of reference types.

Native Type	Java Type	Description	Signature
unsigned char	jboolean	unsigned 8 bits	Z
signed char	jbyte	signed 8 bits	B
unsigned short	jchar	unsigned 16 bits	C
short	jshort	signed 16 bits	S
long	jint	signed 32 bits	I
long long	jlong	signed 64 bits	J
float	jfloat	32 bits	F
double	jdouble	64 bits	D

Table 6.1: Mapping of types between Java and native code

Non-primitive types have signature *"L fully-qualified-class ;"*. For example String class has a signature *"Ljava.lang.String;"*. Prefixing *[* to the signature of type makes an array of that type. For example *[I* means int array type. The primitive (see Table 6.1) are interchangeable. For instance type jint can be used where int is used and vice versa without any type casting. However, mapping between Java String and arrays to native strings and arrays requires use of conversion functions such as *GetStringUTFChars()*, *GetIntArrayRegion()*, etc.

[1] Java Virtual Machine

JNI Type	Java Type
jclass	java.lang.Class
jstring	java.lang.String
jobjectArray	Object[ ]
jbooleanArray	boolean[ ]
jcharArray	char[ ]
jshortArray	short[ ]
jintArray	int[ ]
jfloatArray	float[ ]
jdoubleArray	double[ ]

Table 6.2: JNI reference types

6.2 Java Wrapper

Pixelman API for plugins consists of one header file *mpxpluginmgrapi.h* containing a table of functions that plugins can use. In order to make these functions available to Java part they have to be wrapped in JNI functions. This process involves writing large amount of glue code, especially for cases like passing data buffers from C++, converting strings or arrays. To simplify this process several C++ macros were written that converts data types.

6.2.1 Data types

Simple variables

Simple variables like jint, jdouble, jfloat, jshort needs to be only retyped to they C++ counter parts using macro:

Listing 6.1: JNI simple variable

```
#define NEW_SIMPLE_VAR(variable, typec, typej)\
    typec _##variable = (typec)variable;
```

Unsigned type are converted to the closest high precision java data type (unsigned char -> short, unsigned int -> long), because Java does not support unsigned data types.

Pointers to primitive types

Pointers to primitive data types used in functions as output variables are converted by *NEW_NUM_OUT* and *DELTE_NUM_OUT_ASSIGN* (see Listing 6.2) macros. In Java an array of primitive data type is used. After calling the JNI function first element in the Java array is filled out with an output value.

Listing 6.2: JNI output variables

```
// gets array element to pass to a functions
#define NEW_NUM_OUT(variable, typec, arraytype)\
    typec* _##variable = NULL;\
    if (variable)\
        _##variable = (typec*)jenv->Get##arraytype##ArrayElements(
            variable, 0);

// assigns value back to Java array
#define DELETE_NUM_OUT_ASSIGN(variable, typej, arraytype)\
    if (_##variable != NULL)\
        jenv->Release##arraytype##ArrayElements(variable, (typej*)_##
            variable, 0);
```

Strings

Java strings are converted by macro *NEW_CONST_CHAR* to *const char** variables. Java strings are encoded in UTF-8 encoding, but Pixelman uses local encoding in all strings. Therefore before passing to C++ part they are converted by function *jStringToLocaleString()*. When a function passed string as an output (char*) *NEW_CHAR_OUT* and *DELETE_CHAR_OUT_ASSIGN* macros are used. The second macro also converts local encoded string to Java string using *localeStringToJString()*.

Listing 6.3: JNI strings

```
#define NEW_CONST_CHAR(variable)\
    const char* _##variable; \
    std::string std_##variable; \
    if (variable) { \
        if (jStringToLocaleString(jenv,variable, std_##variable) != 0)
            {\
            return -1;\
        }\
        _##variable = std_##variable.c_str();\
    }else\
        _##variable = NULL;

#define NEW_CHAR_OUT(variable, length)\
  char* _##variable = NULL;\
  if(variable)\
      _##variable = new char[length];

#define DELETE_CHAR_OUT_ASSIGN(variable)\
    if (_##variable != NULL){\
        jstring _j##variable = localeStringToJString(jenv, _##variable
            ); \
        if (_j##variable != NULL){ \
            jenv->SetObjectArrayElement(variable, 0, _j##variable);\
            jenv->DeleteLocalRef(_j##variable);\
            delete[] _##variable;\
        }else{ \
            delete[] _##variable;\
            return -1; \
```

```
        } \
    }
```

Data buffers

Large data buffers of primitive types are transferred to Java through *ByteBuffer* class. This class has an advantage that its data can be accessed directly in C++, just a pointer to memory is passed. Macro *NEW_BYTE_BUFFER* is used to get pointer to the data.

Listing 6.4: JNI data buffers

```
#define NEW_BYTE_BUFFER(variable)\
    byte* _##variable = (variable != NULL) ? (byte*)jenv->
        GetDirectBufferAddress(variable) : NULL;
```

Class fields

It is possible in JNI to obtain values of Java class fields. This is wrapped by macros *GET_NUM_FIELD* and *GET_OBJECT_FIELD*. First macro obtain value of primitive data type field (int, byte, short, etc.). The second one gets arbitrary Java object field. To change value of a field macros *SET_NUM_FIELD* and *SET_OBJECT_FIELD* are used.

Listing 6.5: JNI class fields

```
#define GET_NUM_FIELD(funcn, object, name, type,typefield, signature)\
    jfieldID jfid_##name = jenv->GetFieldID(jcls_##object, #name,
        signature);\
    if (jfid_##name == 0)  return -1;\
    type _##name=(type)jenv->Get##typefield##Field(object, jfid_##name
        );

#define SET_NUM_FIELD(func, object, name, typefield,signature, value)\
   jfieldID jfid_##name = jenv->GetFieldID(jcls_##object, #name,
      signature);\
   jenv->Set##typefield##Field(object, jfid_##name, value);
```

Special structures

Pixelman API exports, apart from the primitive data types, also several C structures. These structures cannot be mapped to java directly. For each structure one corresponding Java class and two C++ function were created. One of these functions takes as a parameter Java object and fills out a corresponding C structure with its data. The other function does the opposite operation.

6.2.2 Function pointers (Callbacks)

Function pointers are used in C++ on everyday basis. In Java, they do not exist. Neither is possible to pass a pointer to Java method through JNI. However, Pixelman API uses such function pointers for event callbacks. A solution to this problem was to create a function in C++ that would be passed to Pixelman API and that would, when called, call handling function in Java. The only remaining problem was to distinguish response call to different registered callbacks. Fortunately, all callback functions in Pixelman API support a user data parameter. This parameter is used to index different Java callbacks. However, it is still possible to use the user parameter in Java, because a value of the user parameter is stored before the user parameter is used as callback index. When the callback is invoked, the user parameter is set to its original value. Since function pointers do not exist in Java, an interfaces are used instead. An example of such an interface and its process function is shown in Listing 6.6

Listing 6.6: Callback interface

```
public interface CallbackFuncInterface {
    public void callbackFunction(long callbackParam, long userData);
}

public static void callbackFunctionProcess(long callbackParam, long
    userData){
     CallbackFuncInterface callbackInterface = callbackFunctionsTable.
         get((int)userData).func;
     if (callbackInterface != null){
          callbackInterface.callbackFunction(callbackParam,
              callbackFunctionsTable.get((int)userData).userData);
      }
}
```

6.2.3 Automatic generation of Java Wrapper

Pixelman API contains about 150 functions. To write a JNI wrapper function for every single one of them manually would be inefficient. A generation script in Ruby language was written that takes as an input a header file with API functions (*mpxpluginmgrapi.h*) and outputs these files:

JavaWrapper.java contains declaration of native functions.

PixelmanFuncs.java implements *PixelmanFuncsInterface* and wraps calls to JavaWrapper. It replaces also few functions with more convenient interface. For example *mpxCtrlGetFirstMpx()* and *mpxCtrlGetNextMpx()* are replaced by function *mpxCtrlGetMpxsList()* that returns list of integers.

PixelmanFuncsInterface.java is an interface that is passed to Java plugins. Apart from function it also contains constants definitions.

PluginsFuncWrapper.h is a header file for PluginsFuncsWrapper.cpp

PluginsFuncWrapper.cpp is file containing JNI wrapper functions for all Pixelman API functions.

The generation script does not generate only functions, but it also parses constants definitions and function comments that are assigned to Java functions. An example of function *mpxCtrlGetAcqMode()* is shown in Listing 6.7 and 6.8).

Listing 6.7: JNI wrapped function

```
JNIEXPORT jint JNICALL
    Java_cz_ieap_pixelman_javawrapper_JavaWrapper_mpxCtrlGetAcqMode(
    JNIEnv *jenv, jobject jobj, jint devID, jintArray mode)
{
     NEW_SIMPLE_VAR(devID,DEVID,jint)
     NEW_NUM_OUT(mode, int, Int)
     int res = mgr->mpxCtrlGetAcqMode(_devID, _mode);
     DELETE_NUM_OUT_ASSIGN(mode, jint, Int)
     return res;
}
```

Listing 6.8: Example of Java API function

```
// definition in JavaWrapper.java
public static native int mpxCtrlGetAcqMode(int devID, int[] mode);

/**
 * gets currently selected acquisition mode for selected Medipix
     device <br>
 * [in] devID - medipix device identification <br>
 * [out] mode - current acq mode (check ACQMODE_MANUAL, ACQMODE_xxxx)
     <br>
 *
 * <br><p><b>! Comment has been taken from mpxpluginapi.h when
     generating this file.
 * Comments might be not exact, event invalid for functions in java.
 * Should be used just for basic information !</b></p>
 */
public int mpxCtrlGetAcqMode(int devID, int[] mode){
    return JavaWrapper.mpxCtrlGetAcqMode(devID, mode);
}
```

6.3 Architecture

Figure 6.1 shows all the main parts of Java Interface. The Pixelman core is represented by Manager and Medipix Control Library. The JavaWrapper library (.dll or .so) wraps all the Pixelman API to JNI functions. When JMpxLoader starts it loads JavaWrapper library and JavaWrapper loads Pixelman core along with all the C++ plugins and hardware libraries.

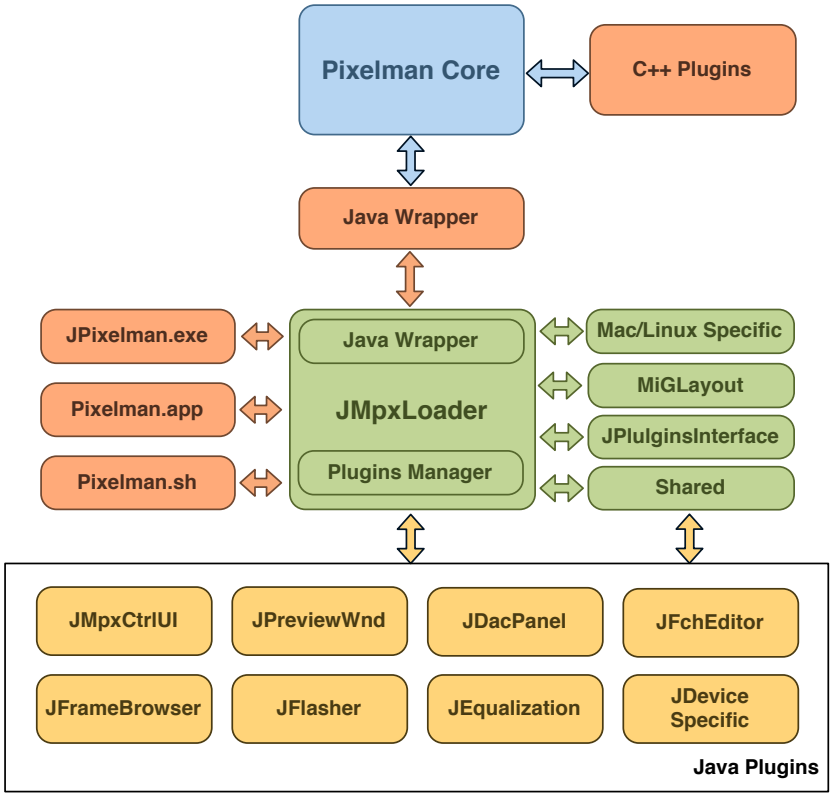


Figure 6.1: Java Pixelman architecture

6.3.1 Java Mpx Loader

JMpxLoader is a central component of Pixelman Java Interface. It has two main purposes. First, loading of JavaWrapper native library that loads Pixelman core and secondly, loading and managing Java plugins and extensions.

Loading of native library is done through *JavaWrapper.java*. It contains mapping for all the JNI functions exported by JavaWrapper. All the fields and methods in this class are static. The class contains also a few private API functions of Pixelman used for MpxManager library initialization and exit. JavaWrapper class is wrapped by *PixelmanFuncs* class that for some functions creates a more user friendly interface. PixelmanFuncs class also implements *PixelmanFuncsInterface* that is passed to all Java plugins.

Loading of Plugins

All the java plugin management is done in *PluginsManager* class. The Java plugins are a single .jar archives. Every plugin implements *JPluginInterface*. Classes contained in plugin jar file are loaded by *PluginClassLoader* class. Internally it uses modified version of *URLClassLoader*. All the resources in the jar archive such as images are unpacked into a temporary directory. Java plugin interface will be described in detail in 6.3.4.

6.3.2 Platform Specific Extensions

Apart from a normal Java plugins a special version of plugins called Platform Specific Extensions exists. They implement *JPluginInterface* and also *JMpxLoaderExtensionInterface*. They take care of implementing an alternative GUI to system tray icon and menu on Windows for other platforms.

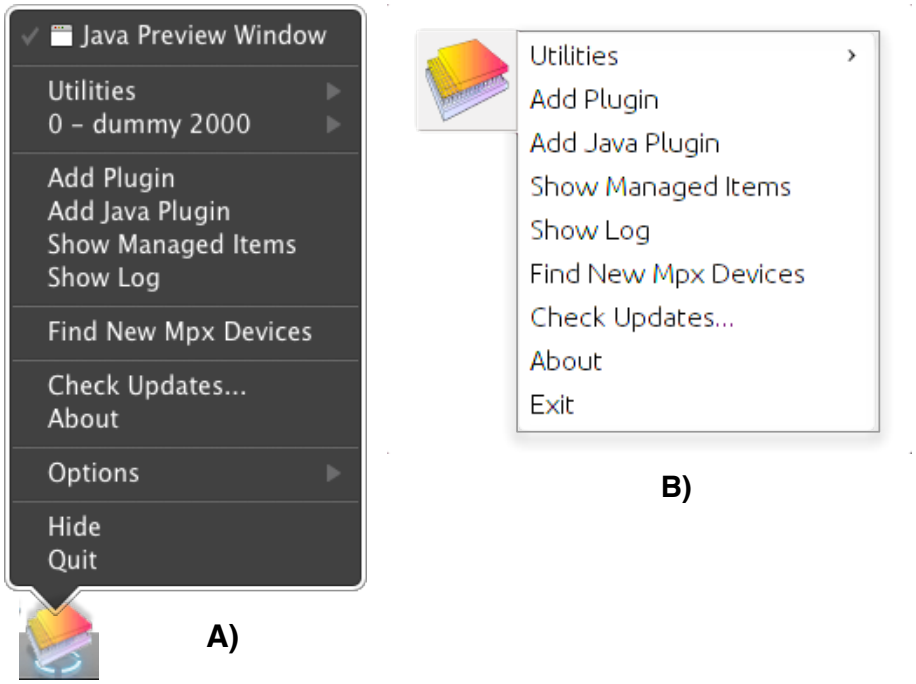


Figure 6.2: Example of platform specific GUI: A) Mac OS X Dock icon, B) floating window for Linux

Mac OS X

MacSpecific extension is used for Mac OS X operating system. Instead of system tray icon it uses the application icon in Dock. Pixelman menu is added to this icon and is accessible by right click. (see Figure 6.2A) Applications on Mac OS X are usually exited through application menu and Quit menu item. MacSpecific registers a callback function for this event to support this behavior.

Linux

Linux operating system has many different distribution and variation. Pixelman supports Debian, Ubuntu and Scientific Linux. Even on these platforms users can use different window managers and there is not a standard solution for creating a system tray icon. As a solution a small floating window with Pixelman icon was created for Linux platform to allow users to access Pixelman menu (see Figure 6.2B).

Windows

On Windows no special Java extension is needed for Pixelman, because Pixelman core uses *mpxMgrUtils.dll* to create system tray icon and corresponding menu. Java plugins can create menu items in Pixelman menu and these are created automatically by Pixelman core.

6.3.3 MiG Layout Manager

One of the requirements for Pixelman Java GUI was to emulate the original MFC GUI as much as possible. The other condition was to have very similar look on all three supported platforms. This poses high requirements on Swing layout managers. Several standard layout managers were considered (BorderLayout, BoxLayout, GridBagLayout, etc.). Writing a complex layout using these layout managers were a slow and complex task. Neither automatic layout generators like Netbeans IDE fulfilled the task. As a result alternative layout managers were considered and MiGLayout was chosen as the most suitable one.

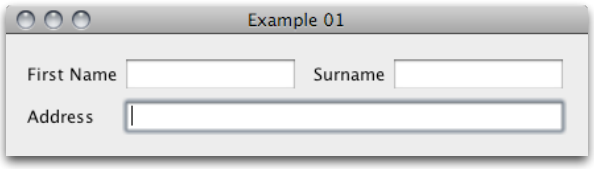


Figure 6.3: MiGLayout Example [24]

MiGLayout is a layout manager for Java Swing framework. It is a very versatile Swing layout manager that makes developing of complex layout very easy. It uses String or API type-checked constrains to format the layout. MiGLayout can produce flowing, grid based, absolute and grouped layouts. Example of a simple layout is shown in Listing 6.9 and the corresponding window in Figure 6.3.

Listing 6.9: MiGLayout example

```
JPanel panel = new JPanel(new MigLayout());
panel.add(firstNameLabel);
panel.add(firstNameTextField);
panel.add(lastNameLabel, "gap unrelated");
panel.add(lastNameTextField,   "wrap");
```

```
panel.add(addressLabel);
panel.add(addressTextField,    "span, grow");
```

6.3.4 Java Plugins

Java plugins have a form of a jar archive. In order to be recognized by Pixelman they have to meet few conditions. Firstly, a jar archive has to contain a file *manifest.jplugin*. This file has only one line that specify main class of the plugin. Example for JMpxCtrlUI plugin:

```
Main-Class: cz.ieap.pixelman.jmpxctrlui.JMpxCtrlUI
```

The main class of a plugin has to implement JPluginInterface (see Listing 6.10). This interface includes version of API and method *getAPIVersion()* to check if plugin is compiled with correct version and to avoid possible incompatibility problems. Every plugin also must return its own name in function *getPluginName()*. Name has to be lowercase and should contain only alphanumeric characters. The *JPluginInterface* is defined in *JPluginInterface* library. This library contains also definition of all necessary constants, structures and callback interfaces needed by Pixelman API. Every plugin has to import this library.

After all classes of a plugin are loaded, plugins manager calls plugin *initializePlugin()* method and passes as arguments: Pixelman API interface *pixelmanFuncsInterface*, path to temporary directory where resources are unpacked and path to native libraries directory where native libraries of the plugin are stored, if used. When all plugins are loaded and initialized, plugins manager calls *startPlugin()* method of each plugin. When the software is exiting plugin's *finalizePlugin()* method is called. The method *setPluginsManagerInterface* is used to pass an interface to PluginsManager to plugins, so that they can communicate (exchange object, call methods, etc.) with other Java plugins.

Listing 6.10: JPluginInterface.java

```
public interface JPluginInterface {
  public final static int API_VERSION = 206;

  public String getPluginName();
  public int getAPIVersion();
  public int initializePlugin(PixelmanFuncsInterface
      pixelmanFuncsInterface, String temporaryDirectory, String
      nativeLibDirectory);

  public void startPlugin();
  public void finalizePlugin();
  public void setPluginsManagerInterface(JPluginsManagerInterface
      pluginsManagerInterface);
}
```

6.4 Shared Library

Shared.jar library is a collection of different utilities and common classes used in Java plugins. It contains modified GUI components, classes for logging and settings, helper classes for manipulation with buffers, etc. This library is also needed by JMpxLoader and its loaded automatically. Therefore plugins do not have to load it by themselves. Description of main packages in Shared library follows.

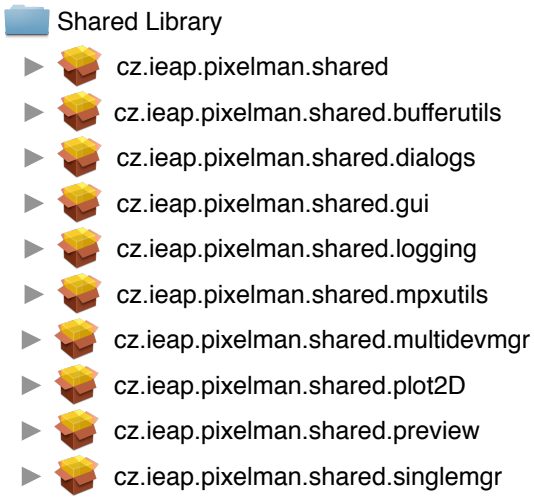


Figure 6.4: Shared library

6.4.1 Buffer Utilities

This package contains several classes that facilitate manipulation of input and output parameters of exported functions. Every plugin can register (export) a function that other plugins can call. This function has this syntax in C++ :

```
int function(INTPTR userData, byte* in, byte* out);
```

Parameters in and out are pointers to memory buffers that contain input and output parameters of a function. Number of parameters may vary and they are described in description string of exported function. Parameters are mapped into memory buffer one after each other. In Java exported function is translated to:

```
public int function(long userData, long in, long out);
```

Parameters in and out are C++ pointers saved in Java long data type. Normally for all buffers in Pixelman API ByteBuffer class is used. However, in this case size of neither buffer is passed and JavaWrapper cannot allocate memory automatically. Instead pointers are passed to java in raw form, and two functions *setBufferAtAddress()* and *getBufferAtAddress()* are used to copy data from or into a ByteBuffer.

Since filling out a memory manually with different data types is not very efficient, helper classes in this package were created.

For each Pixelman data type a corresponding class was created (BOOL, DOUBLE, I16, etc.) that automatically reads or writes its value from/to ByteBuffer. Class *Parameters* then takes variable number of instances of such classes in its constructor and automatically allocates ByteBuffer of right size. If pointer to C++ buffer is passed to constructor, allocated ByteBuffer is also filled with data at pointer address. Example of reading parameters from buffer and writing into a buffer is shown in Listing 6.11.

Listing 6.11: Parameters example

```
public int function(long userData, long in, long out)
{
    // reading parameters
    Parameters param = new Parameters(in, mFuncs, new I32(),
                                      new U32(),  new DOUBLE());
    AbstractParameter[] params = param.getParams();
    int param1 = params[0].getI32();
    long param2  = params[1].getU32();
    double param3  = params[2].getDOUBLE();

    // writing parameters
    Parameters paramOut = new Parameters(new I32(param1),
                                         new U32(param2),
                                         new DOUBLE(param3));
    paramOut.setDataToPointer(mFuncs, out);
}
```

6.4.2 Dialogs

This package contains sets of common dialogs used in Java plugins. Also, it defines modified JFrame (JFrameWithIcon) and JDialog (JEscDialog) classes that are basis for all windows in Java interface. The first adds Pixelman icon into title of window, the second possibility to close dialog with Esc key.

OpenSaveDialog class implements simple interface to open and save file/directory dialogs. It includes also predefined files filters for most common file types in Pixelman. Listing 6.12 shows typical usage of this class.

Listing 6.12: Save dialog example

```
String fileName = OpenSaveFileDialog.showDialog(false, jPnlMainPanel,
    "Save matrix", OpenSaveFileDialog.ASCII_MATRIX);
```

6.4.3 GUI Package

This package contains small GUI utilities such as JTextField validators or number formatters and new or modified swing components. Here is a list of few of them:

CheckBoxList - modified JList that adds for each item a check box

FileJTree - modified JTree that visualizes file system and loads content of opened directories on the fly

FlatButton - special button with flat look

JEditableTable - modified JTable that marks with colors cells that user can modified and allows cell data modification

TitledPanel - modified JPanel that adds title bar, used for docking purposes

A second part of GUI package represent helper classes for implementing better model-view-controller architecture with Swing components. To separate more clearly application logic from GUI classes *Observable* and *Observer* were created. They are based on java.util Observer and Observable, but in contrast to these classes they take advantage of Java 5 generics. To simplify an usage of observables and observers with Swing components modified versions of most often used components were created: ObservableJCheckBox, ObservableJButton, ObservableJMenuItem, etc.

Listing 6.13: Observer and Observable example

```
private ObservableJCheckBox jChbTest=new ObservableJCheckBox("Test");

// create observer
Observer<Boolean> observer = new Observer<Boolean>() {
       public void update(Observable<Boolean> o, Boolean arg) {
           System.out.println("Checkbox is checked: ", arg);
       } };

// add observer to check box
jChbTest.addObserver(observer);
```

6.4.4 Multi Device Manager

Multi Device Manager package simplifies management of device specific plugins. There are two main groups of plugins in Pixelman - general utilities that does not depend on Medipix devices and device specific plugins that interacts with Medipix devices. The later plugins have to react when device is connected on the fly - add a menu item into device menu, change title of device window, etc. Also plugin should save and load its window position into a config file. These are the tasks that repeats for every device specific plugin.

Multi Device Manager class address this problem. It automatically creates a controller and view (a window) for every connected device. The window title is set automatically to plugin name and chip ID of the device. The manager also registers a menu item in the device menu. Window position and other settings are loaded from configuration file and when Pixelman exits they are also saved to this file.

Multi Device Manager also implements a simple window docking architecture that facilitates manipulation of windows when Pixelman controls several Medipix devices at the same time. Two predefined Docking windows are available: *DockView* that puts different windows into sheets in one window, and *TitlesDockView* that puts each window into a small panel in the window. Example of TilesDockView docking two preview windows is shown in Figure 6.5.

In order to use Multi Device Manager a plugin must create two classes implementing *DevViewInterface* and *DevControllerWithViewInterface* and call 3 methods of the manager: *initialize()* in plugin init method, *start()* in *startPlugin()* method, and *exit()* in *finalizePlugin()* method.

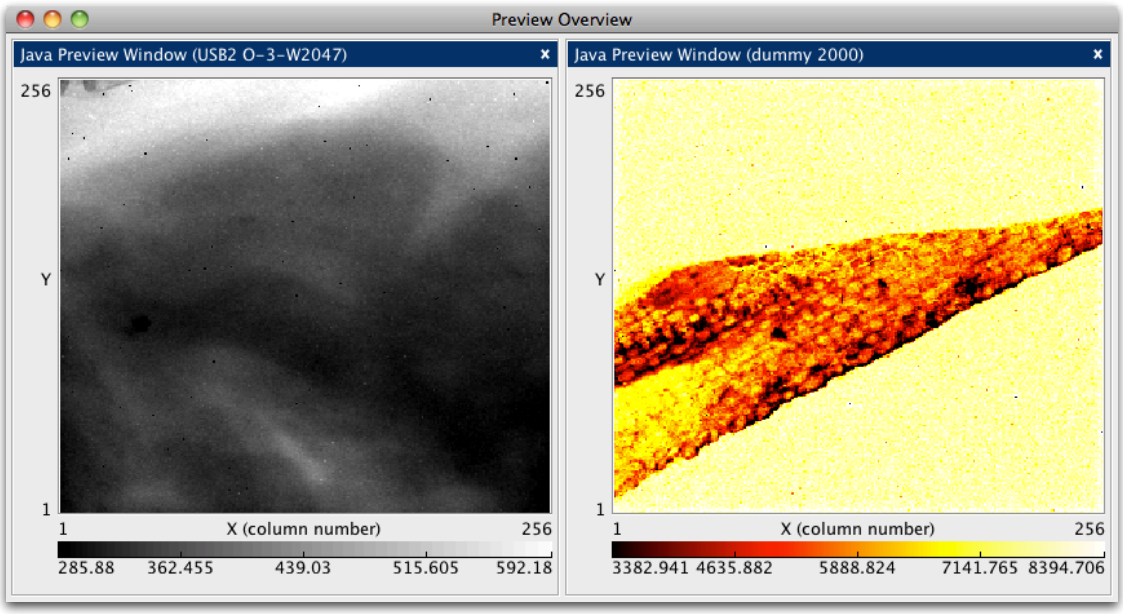


Figure 6.5: Example of TilesDockView with two docked Preview Windows. On the left image of a mouse, on the right pice of painting.

6.4.5 Other Utilities

Plot 2D is a set of classes for painting custom 2D plots. The class is capable of creating lines plots, points plots or histogram plots.

Settings is a class that facilitates loading and saving settings parameters of plugins. It uses a ini file format to store settings.

PixelmanMenuHandler loads content of Pixelman menu into Java and builds corresponding Swing JMenu and JPopupMenu.

Logging are classes that take care of logging error and debug messages into a file.

6.5 Java Device Control Plugin

Java Device Control (JMpxCtrlUi.jar) is a plugin that manages configuration parameters and controls acquisition of one Medipix device. The plugin provides a simple GUI (see Figure 6.6) for accessing all the properties of Medipix devices and their read-out interfaces.

6.5.1 Main Window

The main window allows access to most frequently used acquisition options. A user can specify directory where the data acquired from the device will be saved, format of the data, acquisition time, etc. The window also shows progress of running acquisition and error messages, in a case an error occurs.

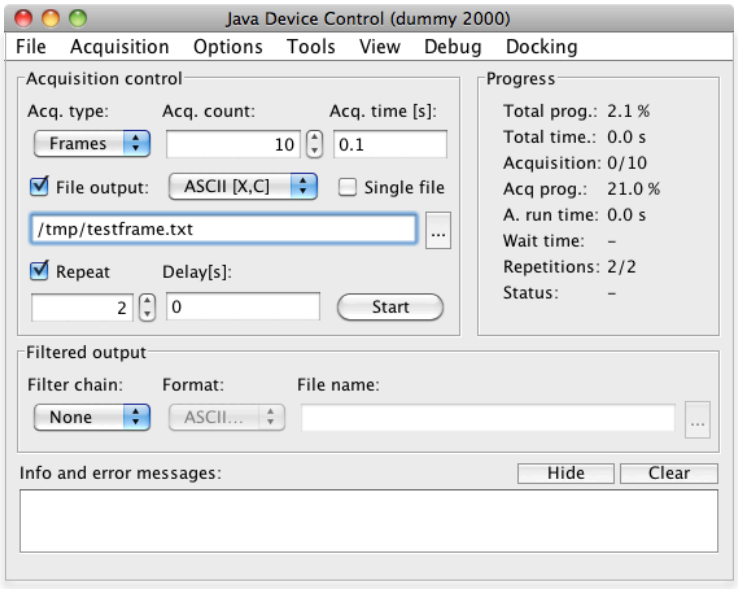


Figure 6.6: Main window of Java Device Control Plugin

Acquisition Control

- **Acq. type** - specifies type of acquisition
  - **Frames** - each frame is saved into a separate file and its description file
  - **Integral** - frames are added into one integral frame
  - **Integral Count Lim.** - integral acquisition that stops when certain number of counts in pixels is reached
  - **Test Pulses** - allows testing of detector with test pulses
- **Acq. count** - number of frames that will be acquired

- **Acq. time** - acquisition time of each frame (time period detector shutter is opened)
- **File output** - frame file path and frame data format - ascii matrix, sparse ascii, binary matrix or binary ascii, sparse binary
- **Single file** - when this option is checked, instead of saving each frame into a separate file, all frames from acquisition are saved into one multi-frame file
- **Repeat** - number of repetitions of acquisition series. Value 0 means infinite number of repetitions
- **Delay** - time delay between repetitions

Filtered output

This panel allows to apply a certain "chain of filters" on every acquired frame. The chain of filters is a list of filters that are sequentially applied on the frame. Filter itself is a function that in some way modifies the frame. It can for example perform various optimization such as flat field correction. The filtered frame then can be saved into a file specified in File name field.

GUI features

The main window of Java Device Control plugin have some new GUI features that the original software did not have. Firstly, several device control panels can be docked into one window or each panel can be docked into corresponding Preview Window. This is very convenient when Pixelman manages multiple Medipix devices. For example for 5 connected detectors, instead of having 10 windows (control window + preview window), all the windows can be docked into one. Secondly, all the panels in the window (filtered output, error messages and comment panel) can be hidden when not used. This saves the space on the screen. The error message panel shows up again automatically when an error message is received.

Medipix3 support

Java Device control plugin creates also an interface (dialog) to set all the Medipix3 chip specific acquisition settings such as:

- Regions of Interest
- Counter depth and Counter selection
- Acquisition modes - FSM, CSM, CRW, ...

6.5.2 Device Settings Dialog

Device Settings dialog access more detailed Medipix and read-out interface settings. It is accessible from main window menu Options -> Device Settings. The dialog consists of 3 or 4 sheets (see Figure 6.7). The first sheet shows detailed device information such as number of chips, chip type, read-out interface type, etc. It allows also to specify device acquisition mode - whether hardware or software triggering is enabled. To test the device a digital test procedure is accessible on this sheet. The digital test procedure checks if the device works properly digitally. It writes a random matrix to the device, reads it back and then compares if they match.

The second sheet controls settings of device DAC values. Analog voltage of each DAC is sensed and shown to check if the DAC works properly. The nominal DAC values can be applied by clicking on Default Values button.

The third sheet (see Figure 6.8) shows information that are relevant only for a read-out interface. It is a list of parameters exported by hardware library that informs about device and read-out parameters such as temperature, bias voltage, etc. Parameters highlighted with orange color can be modified by double-clicking on the value.

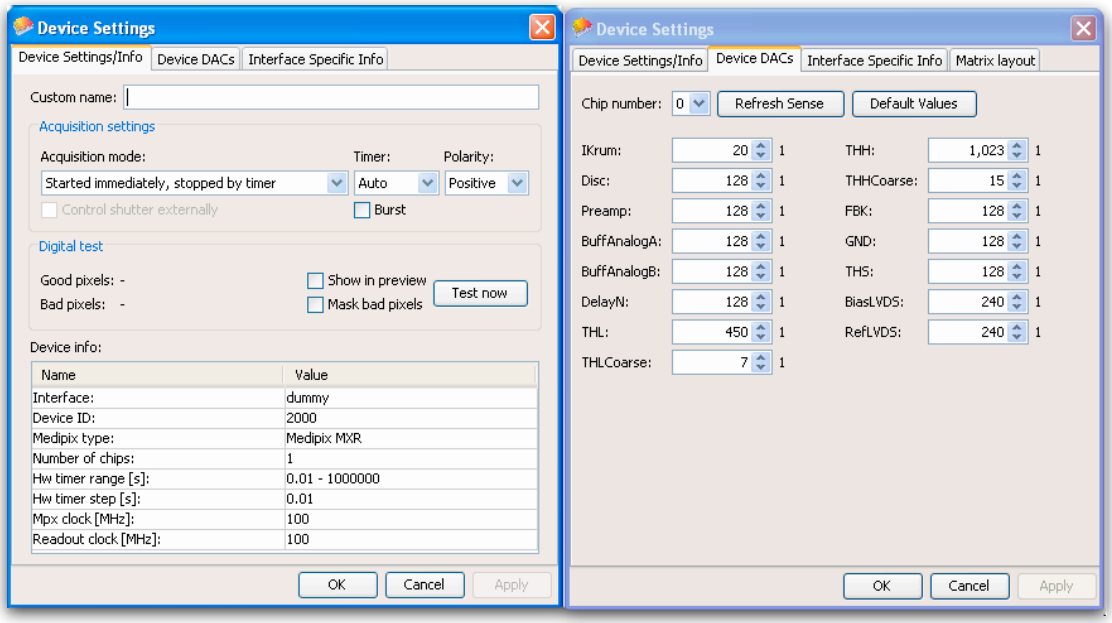


Figure 6.7: Device Settings Dialog - Device Info and Device DACs

The last sheet is visible only when more than one chip is present. It allows users to modify a layout of chips in the matrix. The number of chips in a row and column can be adjusted in Width and Height spinner. The order of chips can be changed by dragging a selected chip with mouse to another position. It is even possible to change rotation of chips by double-clicking on them. The rotation is constrained only to multiples of 90 degrees. The *Create subframes* check box specifies whether sub-frames for each chip should be automatically created when a frame is acquired.

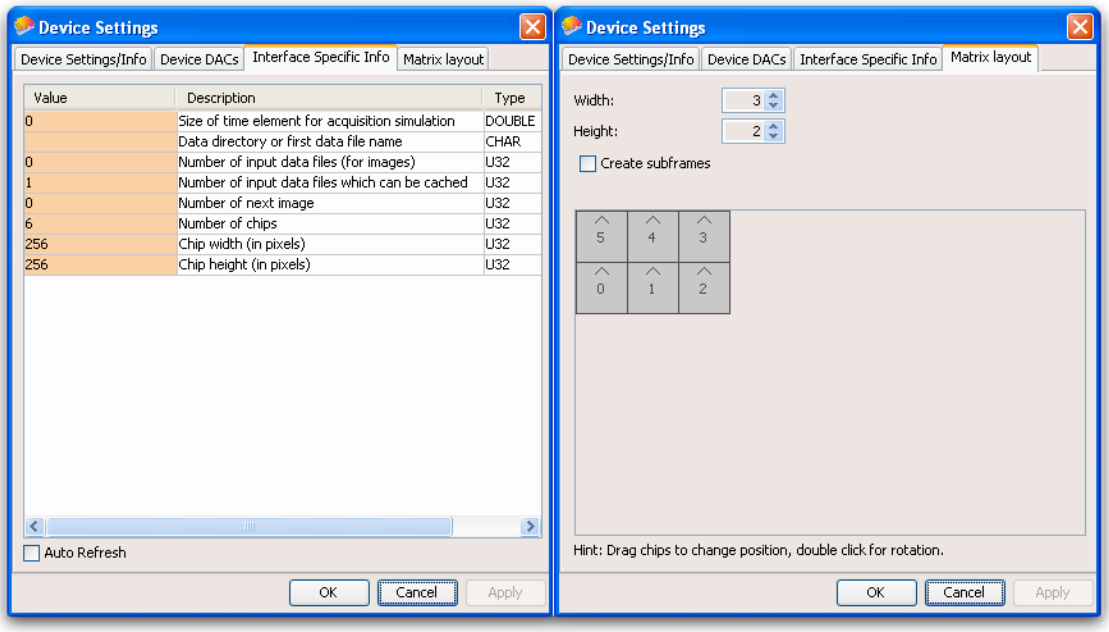


Figure 6.8: Device Settings Dialog - Interface Specific and Matrix layout

6.6 Java Preview Window Plugin

Java Preview Window is a plugin that visualizes graphically frames acquired from Medipix detectors. Figure 6.9 shows main window of the plugin. Compared to its C++ predecessor Java Preview Window has a several new graphical features.

6.6.1 Zooming and Scrolling

The old C++ Preview Window supported only basic zooming functionality. User could select region that would be zoomed, but it was not possible to move the zoom rectangle once zoomed. The only way was to return to the original scale and zoom again. Another problem was when user accidentally zoomed the window to one or two pixels. That could have happened by trying to double-click but instead doing two separate clicks close to each other. Since the zoom was indicated only in axis numbers it was not easily noticed and frames seemed empty. This led to situation where user was trying to find out while the detector is not working whilst the problem was zoomed preview window. The Java Preview Window solves this problem by adding a scrolling capability. When user select a region, it is zoomed and scrollbars are shown. This way user can see that the frame is zoomed and cam move the zoomed region position.

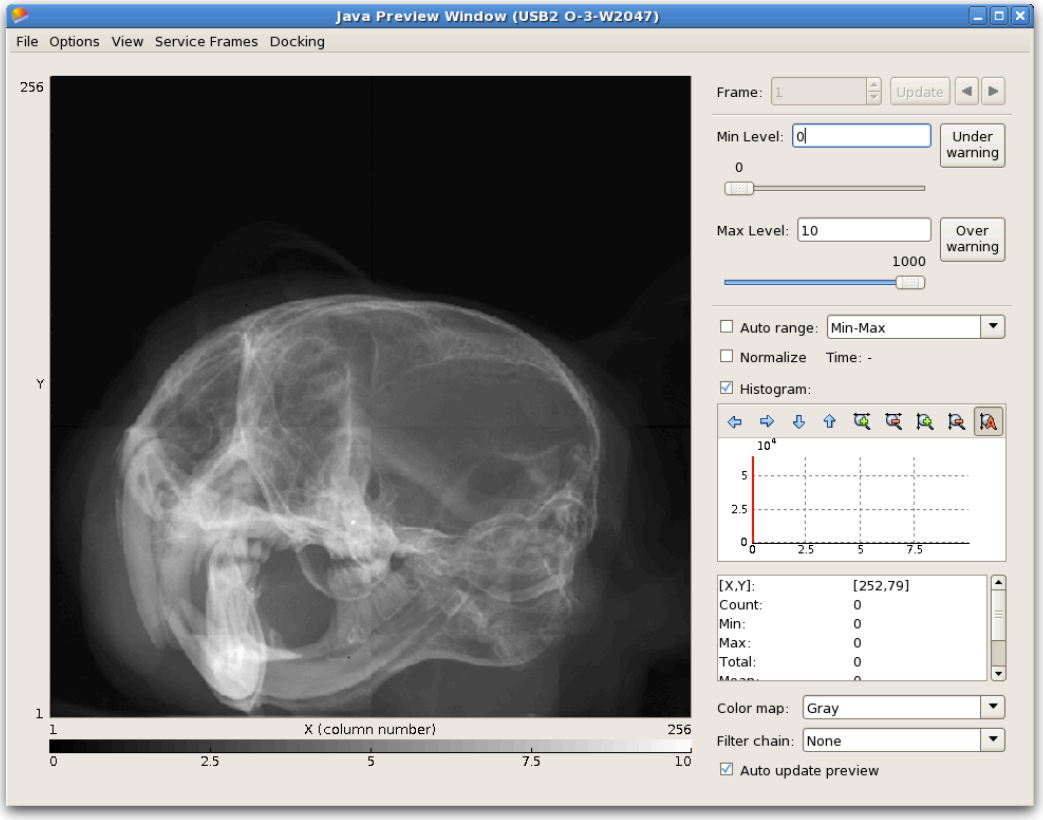


Figure 6.9: Java Preview Window with an image of mouse scull

6.6.2 Docking

Java Preview Window is using the docking capability of Multi Device Manager. Thus, it is possible to dock several preview windows into one common window, which is very convenient when Pixelman manages more Medipix detectors. Moreover, it creates two special docking places - one for Java Device Control Panel and one for Java DAC Control Panel. These panels together with Preview window are the most frequently used plugins to control measurements with Medipix detectors. The DAC panel is docked into the bottom part of the window and the control panel to top right corner. When the control panel is docked, a layout of preview window is changed automatically to save screen space. Both panels can be "minimized" by double-clicking on the title or using minimize icon so that only their title is visible.

Docking these two panels into preview window allows users to have all the needed control items in one window. All the preview windows can be also docked in to one common window. This way, when for example 5 detectors are connected, each detector can be represented by one "preview sheet". This is a great improvement compared to old C++ GUI. In the old version for 5 detectors 15 windows would be shown to display all needed control items, which was confusing.

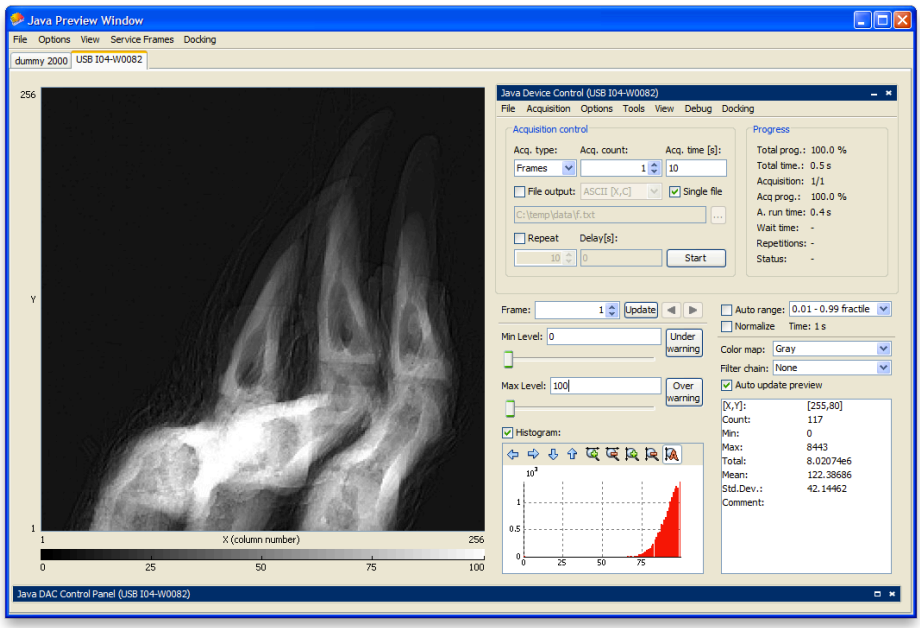


Figure 6.10: Java Preview Window with docked panels and image of a mouse leg

6.6.3 Preview Overview

Preview Overview is a special way of docking preview windows. It is intended for cases when Pixelman is used with several detectors and a status of each detector has to be monitored. When the preview window is docked into Preview Overview, all the controls are hidden and only a frame image is visible. Preview windows are docked into a *TilesDockView* and are placed in a grid (see Figure 6.11).

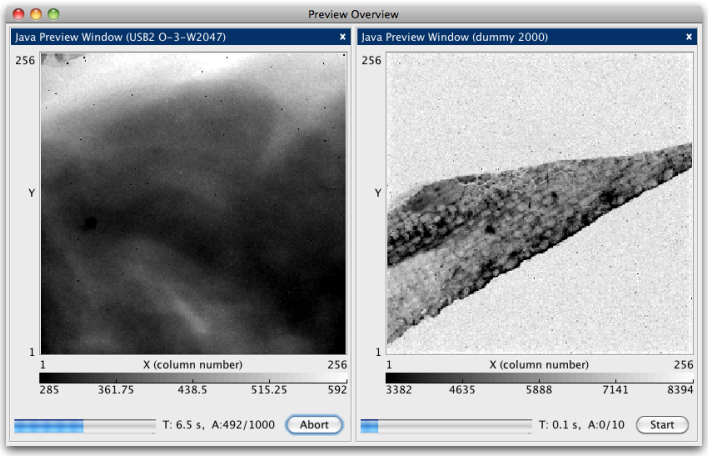


Figure 6.11: Preview Overview. On the left image of a mouse, on the right piece of painting

When a control panel is docked in a preview window and the preview window is docked in Preview Overview, then the control panel is transformed to a minimalistic

layout and shown in Preview Overview as well. In this layout the control panel consists only of a progress bar and status texts that show progress of acquisition and start/abort button to control the acquisition. This way an user can monitor both acquired frames and status of the acquisition.

6.6.4 Subframes

There were two main reasons for new Pixelman GUI. One was a need for multi-platform GUI, the other was an ability to visualize new features of Medipix3 detector. One of new these features is a presence of two counters in each pixel. To store and visualize these two counters a subframes functionality was added to Pixelman.

Subframes are a small frames that are part of a parent frame. They can have different dimensions and different data type than the parent frame. Each parent frame can contain variable number of subframes. The old C++ Preview does not support subframes. Java Preview Window visualizes each subframe as a sheet with a tab below the frame image (see Figure 6.12). The frame itself is always shown in first sheet labeled *Frame*.

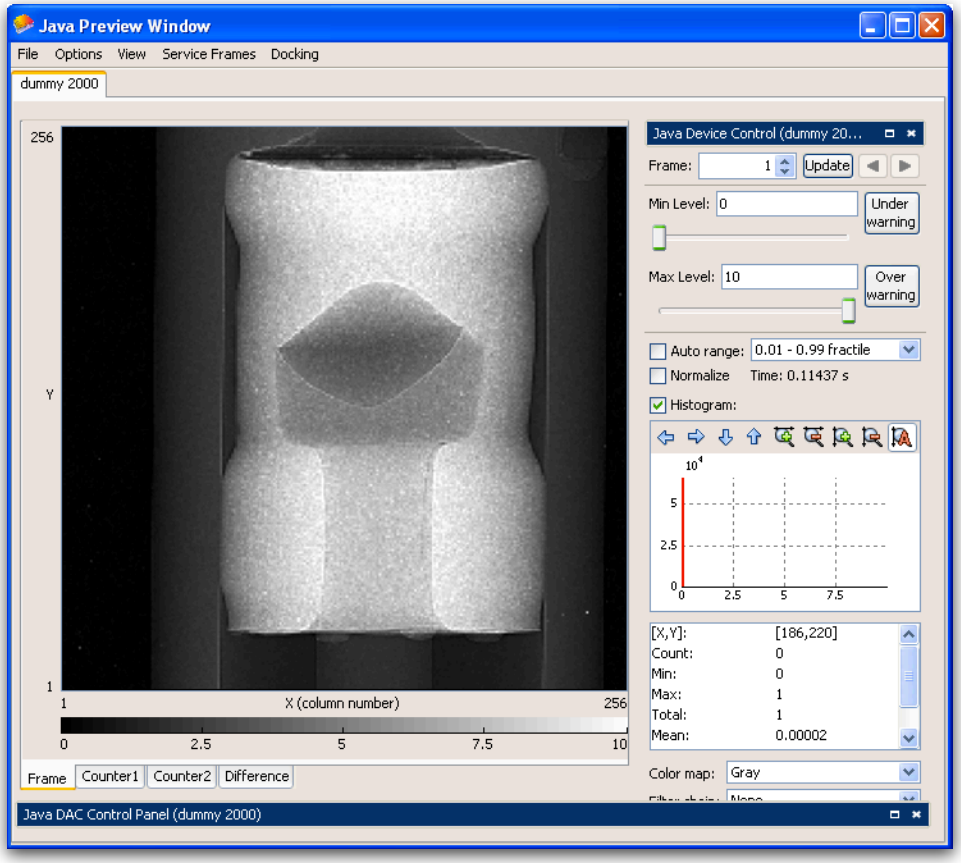


Figure 6.12: Preview Window - Subframes visualization. Image of syringe.

When an acquisition of both Medipix3 counters is preformed, preview window

automatically creates 3 subframes for each frame: *Counter1*, *Counter2* and *Difference*. *Counter1* and *Counter2* are filled with content of each counter. *Difference* is a subtraction of the two: *Difference=Counter1-Counter2*.

6.7 Other Java Plugins

6.7.1 Java DAC Control Panel

Java DAC Control Panel allows to change DAC values instantly, in between acquisitions. If the DAC value cannot be changed immediately (e.g. acquisition is running) the operation is postponed and performed after e.g. the acquisition of a frame is finished. When the Medipix chipboard contains more than one chip, a user can change DAC values of each chip individually. Chip selection is performed by clicking on the button bar in left top corner of the window (see Figure 6.13).

Java DAC Panel can be docked into the preview window and the window can be resized. A scrollbar is shown when DACs cannot fit into the window. This is an improvement, because the old DAC panel could not be resized this way. For example in case of Medipix3, which has 24 DACs, the window was too big to fit on the screen.

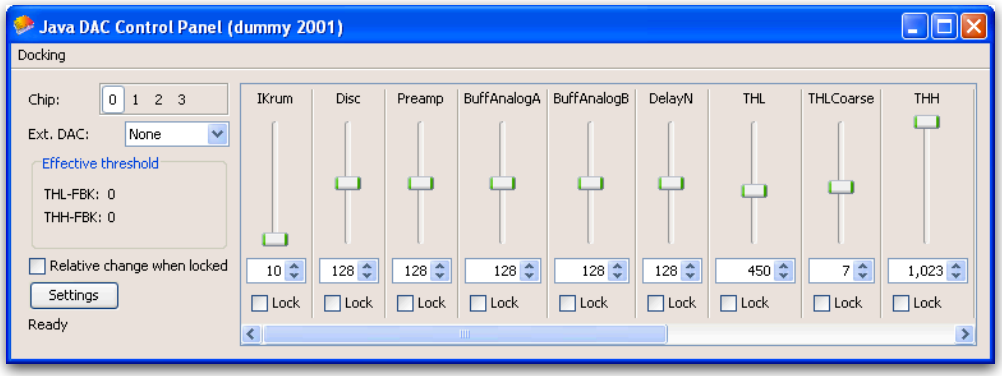


Figure 6.13: Java DAC Control Panel

6.7.2 Java Filter Chain Editor

Java Filter Chain Editor is a plugin for managing chains of filters (see Figure 6.14). User can add and remove filter chains, add filters into a specific chain and change parameters of adjustable filters.

A filter is a special function that perform some computation on a frame. Several such functions (filers) can be chained to create a "processing pipeline". Such a filter chain than can be applied to individual frames in preview window, or can be applied directly on acquired frames in device control panel.

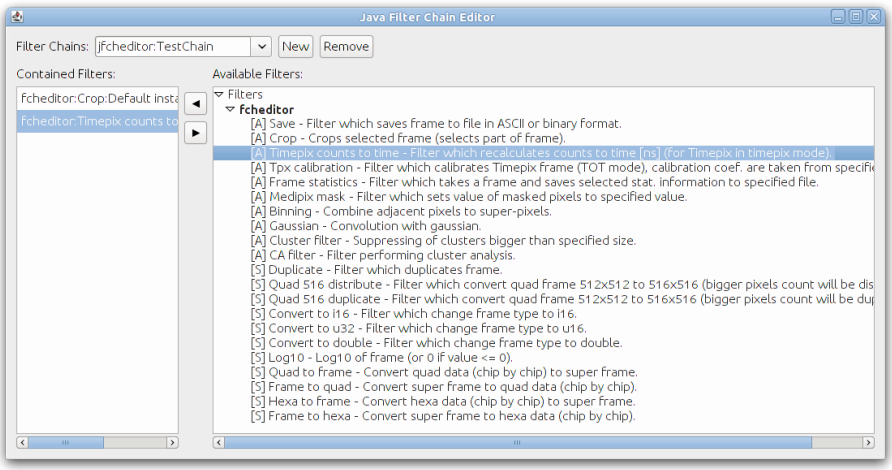


Figure 6.14: Java Filter Chain Editor

6.7.3 Java Frame Browser

Java Frame Browser is a plugin for easy manipulation with measured frames (see Figure 6.15). It allows to browse through frames, see frames attributes such as acquisition time, date, dimension, type, etc. Frames can be loaded from disk, saved to disk, converted to different type. The plugin support both subframes and multi-frame files.

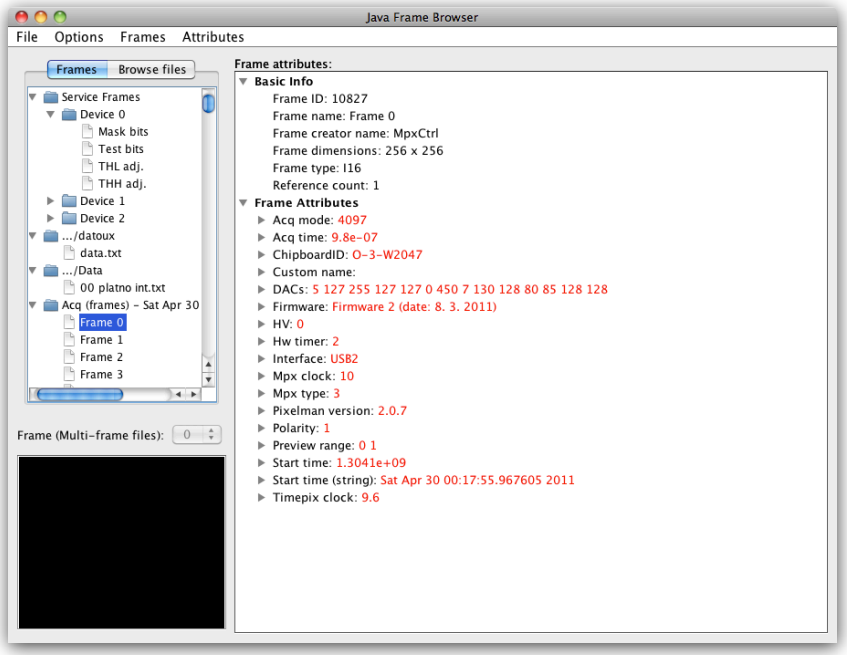


Figure 6.15: Java Frame Browser

6.7.4 Firmware Flasher

Firmware flasher is a plugin that can upload a new firmware into USB Interface via USB protocol. The motivation for a flasher plugin was a fact that Medipix3 requires different firmware than Medipix2/Timepix. Since the same USB interface can be used for both Medipix3 and Medipix2/Timepix chips and users might change the firmware relatively frequently, a need for flasher plugin has arisen.

The plugin has two parts: Java GUI and native C++ library. The native library is required, because Java does not give a direct access to hardware. Thus, the native library takes care of all communication with USB interface. Java part then only calls flash function from native library and shows status messages during flashing. Figure 6.16 shows a window of the plugin.

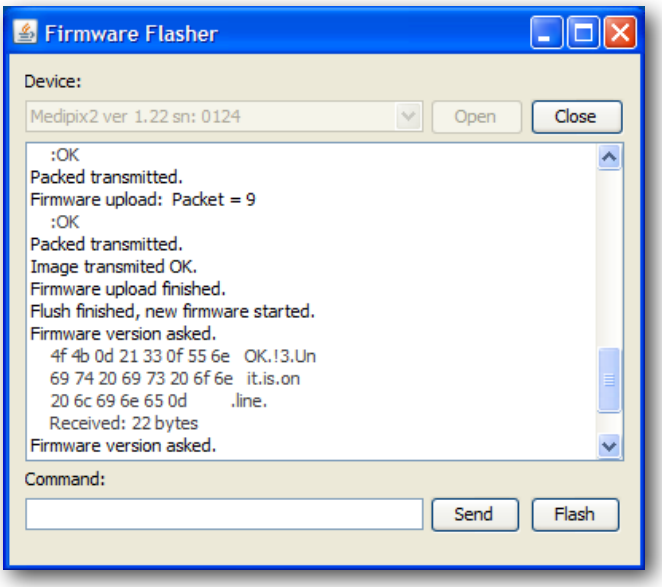


Figure 6.16: Firmware flasher

6.7.5 Java Device specific

Java Device Specific is a plugin that controls device specific parameters like Timepix clock, bias voltage or threshold DAC. The plugin window can be docked into Java Device Control Panel (see Figure 6.17). This way users have access to most frequent parameters from Interface Specific sheet in Device Settings dialog.

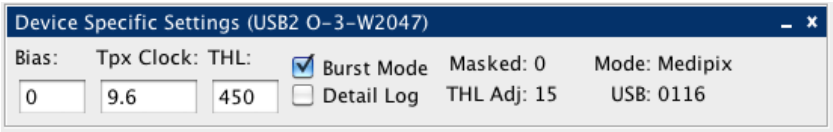


Figure 6.17: Java Device Specific

The plugin automatically changes visible parameters according to different device type. For USB Interface and FITPix interface panel offers parameters: bias voltage, Timepix clock, threshold DAC, burst mode switch and detail logging switch. It also informs about: masked pixels count, value of threshold adjustment bit in pixel configuration, Timepix mode and serial number of USB Interface or FITPix device (see Figure 6.18). For FileDevice (see Figure 6.17) it shows parameters: File/Directory with data that should be "replayed", next file index and index of next frame in multi-frame file.

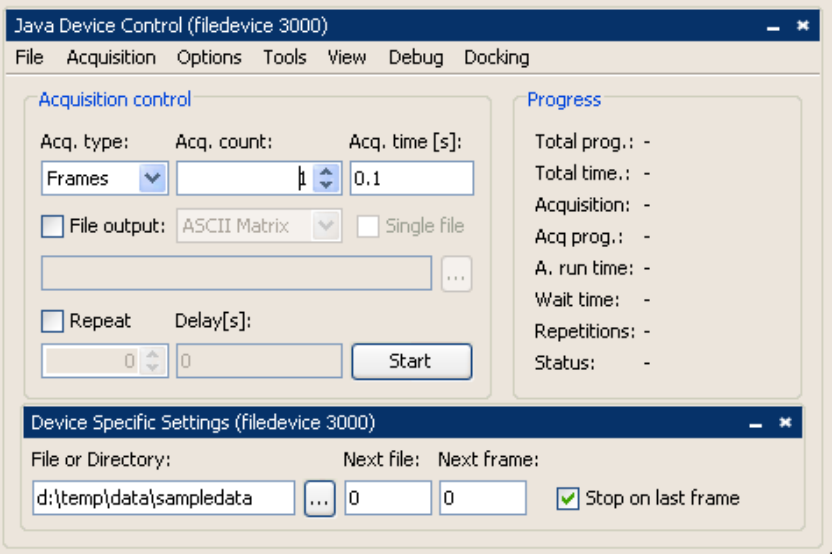


Figure 6.18: Java Device Specific docked

7 Applications

This chapter will demonstrate a flexibility and extendibility of Pixelman. The software can be used in many different applications in high energy physics or medicine. Here, I will present only two of such applications that I have developed software for.

7.1 ATLAS-MPX Network at CERN

7.1.1 ATLAS-MPX Network

The ATLAS-MPX Network[25][26] is a network of 16 Medipix2-based devices (ATLAS-MPX devices) that have been installed in various positions in the ATLAS detector (see Figure 7.1). The aim of the network is to perform real-time measurement of spectral characteristics and composition of radiation inside the ATLAS detector during its operation.

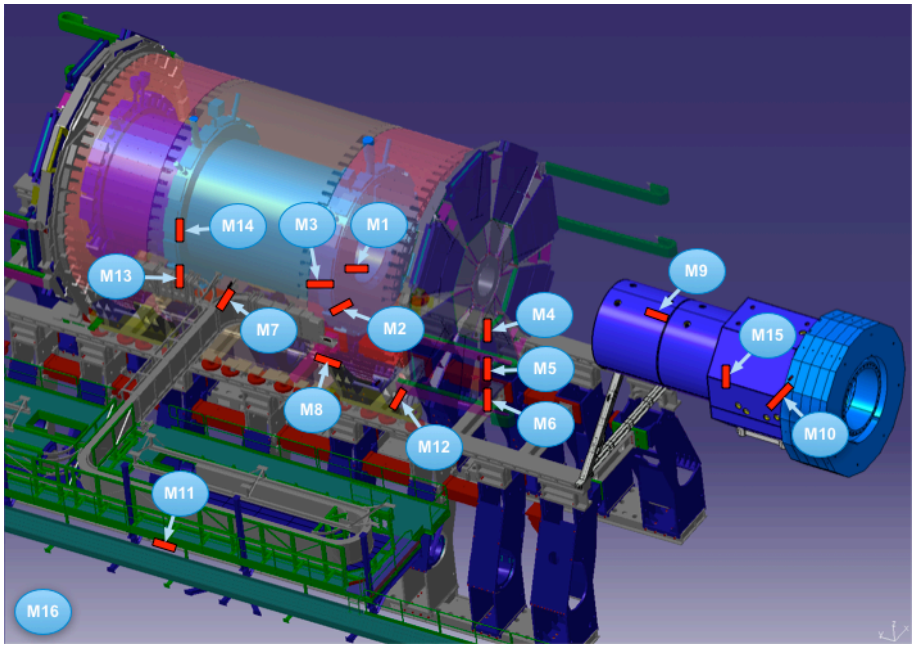


Figure 7.1: ATLAS-MPX Network

The main advantage of ATLAS-MPX network compared to other sub-detectors of ATLAS is, that Medipix detectors can distinguish different particle types (see Figure 7.2) and therefore provide more precise information about composition of radiation inside ATLAS detector.

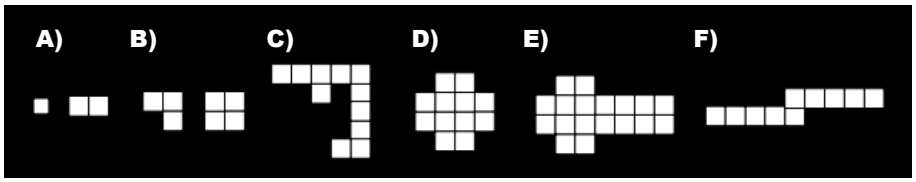


Figure 7.2: Different types of particles: A) Dots (photons and electrons 10 keV), B) Small blobs (photons and electrons), C) Curly tracks (electrons 10 MeV), D) Heavy blobs (heavy ionizing particles - alpha particles,...), E) Heavy tracks (heavy ionizing particles - protons, ...), F) Straight tracks (MIP, Muons, ...)

The ATLAS-MPX Network consists of 16 ATLAS-MPX devices and 4 computers. The devices are connected to computers by USB Interface with radiation hard LVDS extenders. Three of the 4 computers are running Microsoft Windows operating system and Pixelman. Each of these 3 computers is controlling 5 or 6 ATLAS-MPX devices. The last computer is running Linux and hosts control and visualization applications. This computer controls the whole network, stores measured data and presents results of measurements through two web based interfaces.

The ATLAS-MPX devices are based on the Medipix2 chip. Each detector is assembled with a 300 μm thick silicon pixel sensor which is covered by a mask of neutron converting materials (see Figure 7.3) - 6LiF and polyethylene (PE).

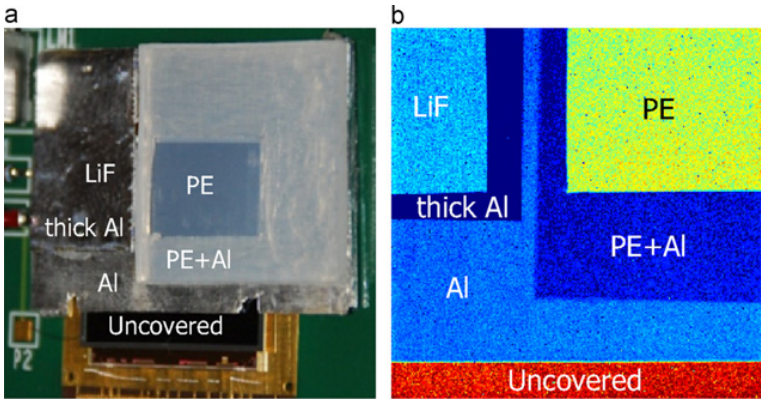


Figure 7.3: Photograph of the Medipix2 detector with mask of neutron converting materials on the top. b) X-ray image of the six different detection regions

7.1.2 Remote Control Plugin

The ATLAS-MPX network consists of 16 Medipix devices and 3 Pixelmans. Even with the new Java Interface, each detector has to be configured separately. For

16 detectors and 3 computers this is not very efficient. Therefore a need for more efficient and central control of all devices had arisen. As a result a remote control plugin for Pixelman was created.

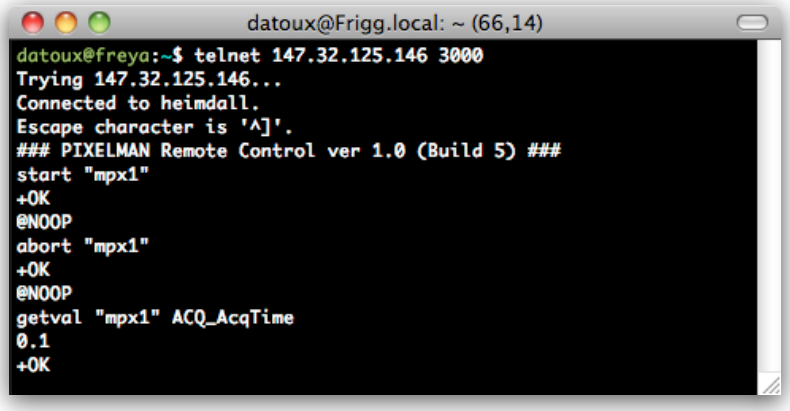


Figure 7.4: Remote Control Plugin protocol example

The Remote Control is a plugin written in C++ that can be complied on all three platforms. It extends Pixelman with a TCP/IP network interface with a simple text protocol (see Figure 7.4). This interface allows for control of acquisition and parameters of Medipix devices connected to Pixelman over a network via terminal such as telnet. It is also possible to call functions offered by other plugins and change options of filter chains in Pixelman. Thus it is very flexible way to control Pixelman remotely. Example of the protocol is shown in Figure 7.4.

7.1.3 Java Remote Control Application

The Java Remote Control Application[26] was created to control Pixelman remotely via Remote Control plugin in more user-friendly way. It can connected to several Pixelmans at the same time. It provides users with web interface (see Figure 7.5), thus it is accessible from different operating systems and can be even controlled over the Internet.

The detectors might be configured separately one by one or a common configuration can be applied to all detectors in one step. Users might create groups of detectors that can be configured separately according to the needs of an experiment. The configuration of detectors can be saved into profiles and loaded again later. Progress of measurement of each detector might be also observed. The system distinguishes different access levels and for each user different access rights might be specified.

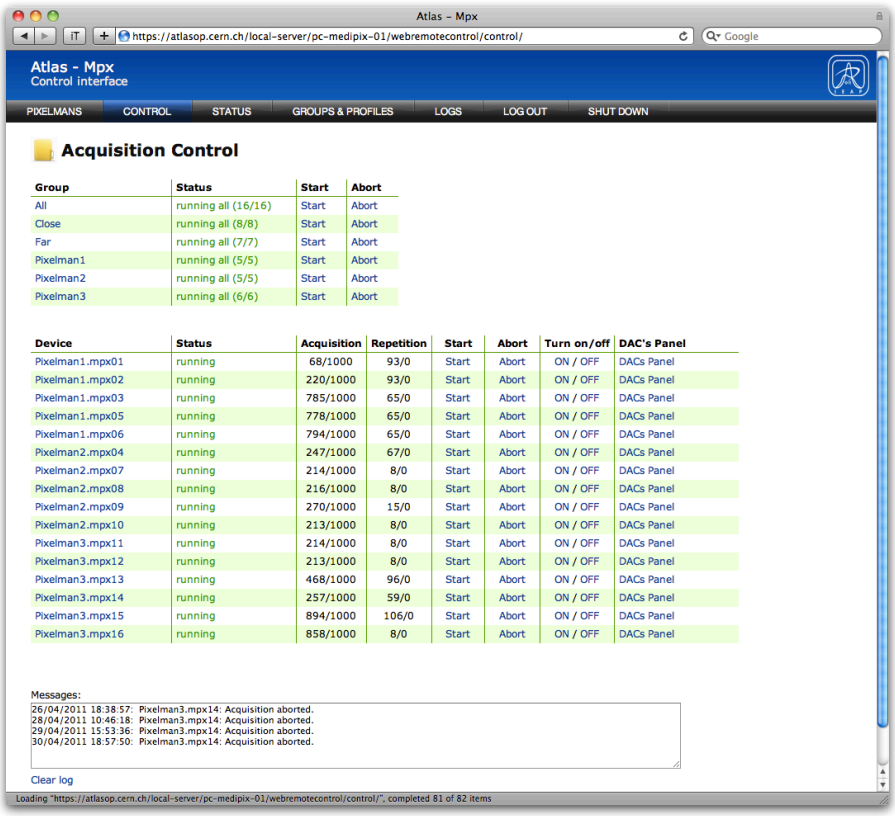


Figure 7.5: Java Remote Control Application

7.1.4 Data Visualization Application

Data Visualization Application features a web interface as well. It is intended for scientific community as well as for wide public. It allows users to browse through history of measured frames from all detectors and search for specific frame by date and time (see Figure 7.6). Frames images are generated from data files on the fly. It is also possible to generate integral frames over selected period of time. For each frame (or integral frame) statistics are shown.

The measured data frames are analyzed by a Cluster Analysis[27] plug-in for Pixelman. Results of this analysis are recognized characteristic traces corresponding to different types of ionizing radiation. Every track is also assigned to a region corresponding to a converting mask. Output of cluster analysis is saved along with the data to a disk. The Data Visualization application reads these data and puts results of cluster analysis into a MySQL database for faster data processing.

From the data in the database graphs containing number of different characteristic traces in each frame in specified time range might be rendered. (see Figure 7.7). It is also possible to render number of traces in specified region or number of traces of one trace type in different regions.

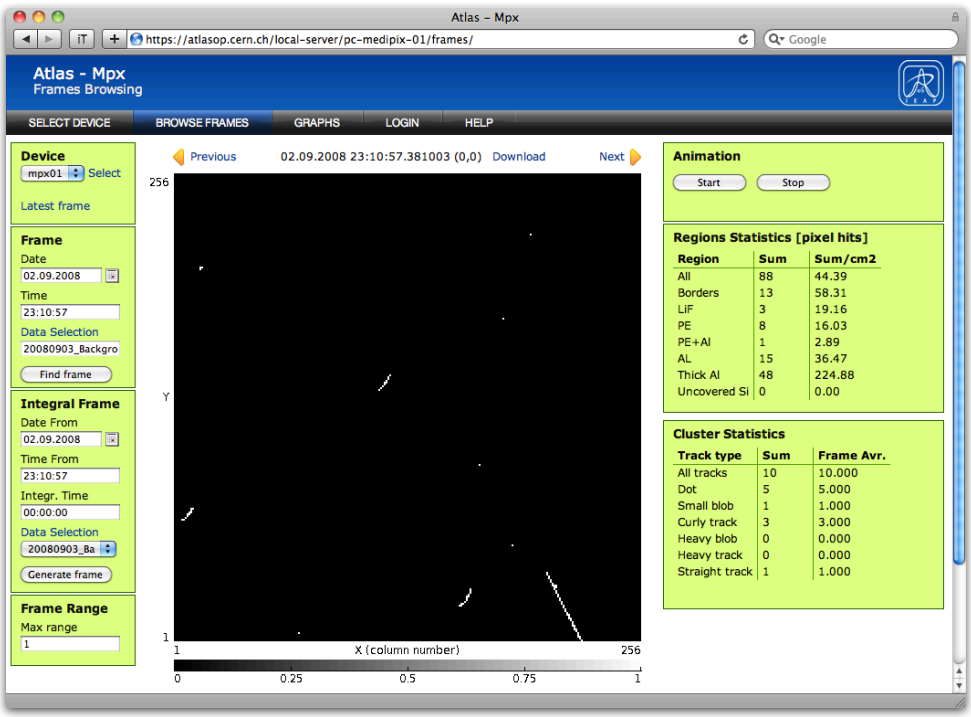


Figure 7.6: Data Visualization Application - Frames

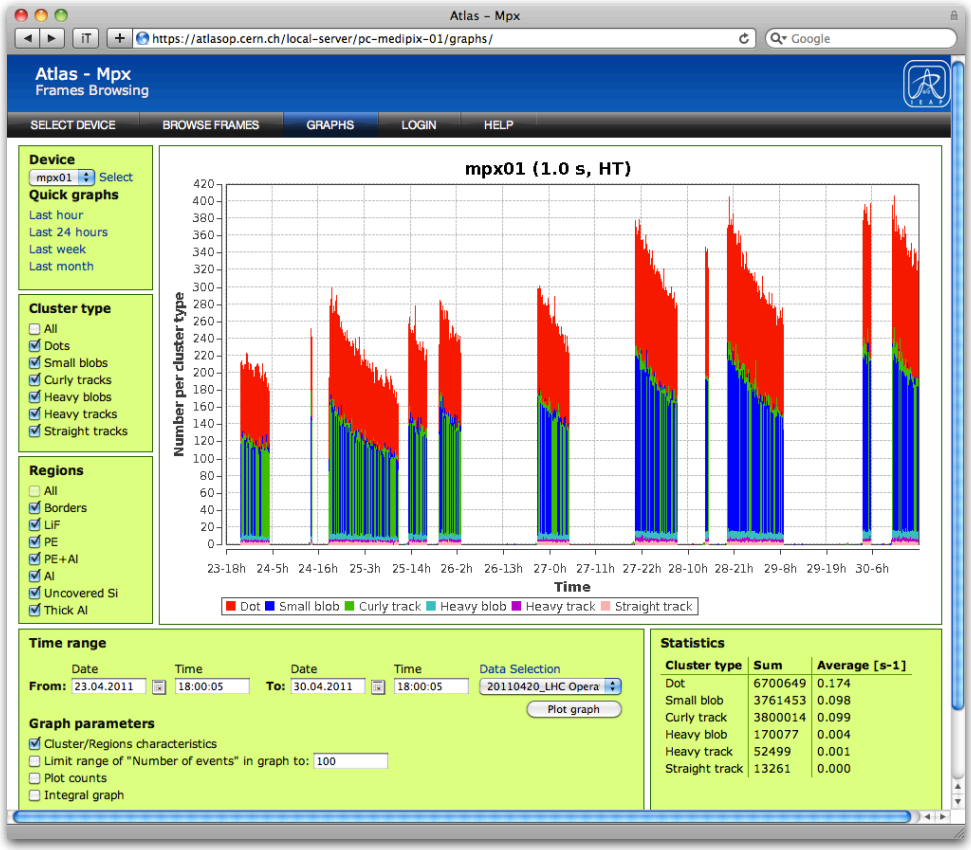


Figure 7.7: Data Visualization Application - Graphs

7.2 Dosimeter for International Space Station

The radiation environment in space is different than on Earth. The standard detection methods that are used nowadays fails. The reason is that most of the dose comes from interactions of heavy ions, mainly protons, that are not present on Earth. Measuring a track of particles and their deposited energy allows us to distinguish different particles. This information can be used for sorting of particles into different categories. Each category can be assigned a quality factor corresponding to the energy a particle would deposit in human tissue. By summing the dose of all particles an estimate of total dose rate can be calculated.

Timepix detector possesses suitable properties for measurements of this type. It is a position sensitive pixelated detector. Its ability to visualize tracks of ionizing particles was already demonstrated. Together with NASA and University of Houston an experiment was designed to demonstrate possibilities of dosimetry measurements with Timepix detector in space. For this purpose a miniature version of USB Interface (USB Lite Interface [7]) with Timepix detector together with Pixelman will be sent to International Space Station and will be tested during a mission planned towards the end of year 2011. The device will be connected to one of the on-board laptops and special version of Pixelman with dosimetric plugin will be measuring a dose rate.

7.2.1 Dosimetric Plugin

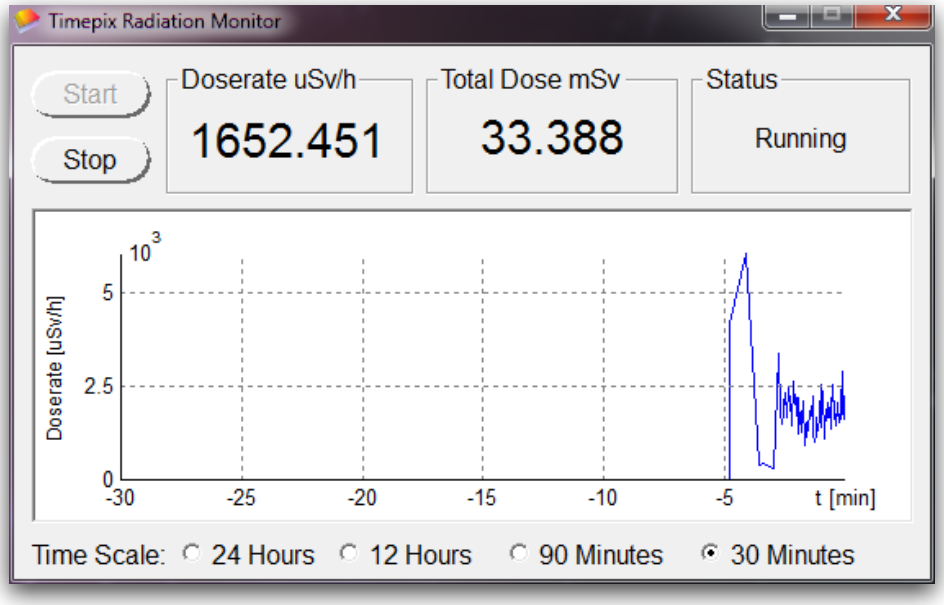


Figure 7.8: Main window of dosimetric plugin

Dosimetric Plugin controls the acquisition, analyzing of frames and calculating of dose rate. On each acquired frame a cluster analysis is performed, parameters of

particle tracks are determined and from the parameters a dose rate is calculated. The plugin has a very minimalistic GUI consisting only of start/stop button, information about dose rate and plot showing history of dose in last 24 hours (see Figure 7.8). This is done on purpose so that no special training is required for astronauts to operate the software. The acquisition and dose rate calculation parameters will be adjusted in a config file.

The plugin, when started, operates fully automatically. It adjusts the acquisition time and frame rate so that an optimal performance of dose rate calculation algorithm is achieved. The measured data and log files will be saved to server and from there transferred to Earth for further analysis. In the opposite direction config files for adjusting plugin parameters will be transferred. The plugin is very flexible and highly configurable. All the parameters of frame rate and dose calculation algorithm can be changed in a config file. Also other parameters such as frequency of data uploading to server, path to config and data directories or logging level can be loaded from a config file as well.

Conclusion

This work has been focused on development of a multi-platform software for radiation detectors Medipix. The first 3 chapters summarize properties of Medipix detectors and commonly used read-out systems. The architecture of the original software, which is upgraded and extended in this work, is described as well.

A development of hardware libraries for USB Interface and FITPix interface is a topic of the fifth chapter. The new hardware libraries are created with an accent on multi-platform usage. They are compatible with all three supported operating systems. The FITPix library implements advance automatic error recovery features to ensure uninterrupted data acquisition. A special burst acquisition mode is implement in order to achieve maximal read-out speed of about 90 frames per second.

The most significant extension to the original software is the addition of Java interface. A special bridge between Pixelman core written in C++ and Java interface was made. This bridge ports to Java Pixelman API, thus making it possible to write extension modules (plugins) in Java programming language. This is then used to develop several Java plugins that creates a new multi-platform GUI. This new user interface implements features such as window docking, that makes operating the software more comfortable for the users.

The last chapter describes two real-life application where Pixelman is used. First of these is distributed acquisition system in ATLAS detector at CERN, where 16 Medipix2 MXR devices are controlled by 3 Pixelmans. For this purpose Pixelman is extended with a remote network interface and a control application with web interface is made to operate all the devices remotely from one place. Data from this experiment are visualized in a second web application. The second real-life application is a project in cooperation with NASA - a small Timepix dosimeter for International Space Station. A special modified version of Pixelman with a dosimetric plugin that fully automatically evaluates a dose rate at the station and dynamically change acquisition parameters in order to adapt to changing conditions.

Pixelman with the new Java interface, support for fast FITPix read-out interface and compatibility with the 3 major operating systems, is a important tool for physical scientific experiments and various medical application with Medipix detectors. Support for Medipix3 detector and FITPix interface opens up a new range applications. Pixelman is used worldwide in many scientific institutions. Thanks to its flexible architecture its functionality can be extended with C++ and Java plugins in order to accommodate the demands of future experiments.

My Publications

Author

[1] D. Tureček, T. Holý, J. Jakůbek, S. Pospíšil, Z. Vykydal, *Pixelman: a multi-platform data acquisition and processing software package for Medipix2, Timepix and Medipix3 detectors*, Journal of Instrumentation 6 C01046, DOI: 10.1088/1748-0221/6/01/C01046 (2011)

[2] D. Turecek, T. Holy, S. Pospisil, Z. Vykydal: *Remote control of ATLAS-MPX Network and Data Visualization*, Nucl. Inst. and Meth., DOI: 10.1016/j.nima.2010.06.117.

Co-author

[3] R. Ballabriga, G. Blaj, M. Campbell, M. Fiederle, D. Greiffenberg, E. Heijne, X. Llopart, R. Plackett, S. Procz, L. Tlustos, D. Tureček, W. Wong, *Characterization of the Medipix3 pixel readout chip*, Journal of Instrumentation 6 C01052, DOI: 10.1088/1748-0221/6/01/C01052 (2011)

[4] E. Heijne, R. Ballabriga, D. Boltje, M. Campbell, J. Idarraga, J. Jakůbek, C. Leroy, X. Llopart, L. Tlustos, R. Plackett, S. Pospíšil, D. Tureček, J. Vermeulen, J. Visschers, J. Visser, Z. Vykydal, W. Wong, *Vectors and submicron precision: redundancy and 3D stacking in silicon pixel detectors*, Journal of Instrumentation 5 C06004, DOI: 10.1088/1748-0221/5/06/C06004 (2010)

[5] E. Gimenez-Navarro, R. Ballabriga, M. Campbell, I. Horswell, X. Llopart-Cudie, J. Marchal, K. Sawhney, N. Tartoni, D. Turecek, *Characterization of Medipix3 with Synchrotron Radiation*, Transactions on Nuclear Science 58 DOI: 10.1109/TNS.2010.2089062

[6] J. Bouchami, F. Dallaire, A. Gutierrez, J. Idarraga, V. Kral, C. Leroy, S. Picard, S. Pospisil, O. Scallon, J. Solc, M. Suk, D. Turecek, Z. Vykydal and J. Zemlicka, *Estimate of the neutron fields in ATLAS based on ATLAS-MPX detectors data*, Journal of Instrumentation 6 C01042, DOI: 10.1088/1748-0221/6/01/C01042

References

[1] *Medipix Collaboration homepage* [online]. [cit. 2011-04-10], Available at: http://www.cern.ch/medipix.

[2] X. Llopart, M. Campbell, R. Dinapoli, D. San Segundo, E. Pernigotti, *Medipix2, a 64k pixel readout chip with 55 um square elements working in single photon counting mode*, IEEE Trans. Nucl. Sci., vol. 49, 2279-2283, 2002

[3] X. Llopart, *Medipix2 MXR Users's Manual*, CERN, distributed only between members of the Medipix Collaboration, 2006

[4] X. Llopart, R. Ballabriga, M. Campbell, L. Tlustos, W. Wong, *Timepix, a 65k programmable pixel readout chip for arrival time, energy and/or photon counting measurements*, Nucl. Instr. and Meth. A 581, 21 October 2007, Pages 485-494 and erratum Nucl. Instr. and Meth. A 585 (2008) 106

[5] *Medipix at NIKHEF* [online]. 2008-08 [cit. 2011-04-10]. Available at: http://www.nikhef.nl/pub/experiments/medipix/muros.html

[6] Z. Vykydal, *USB Interface for Medipix2 Pixel Device Enabling Energy and Position Detection of Heavy Charged Particles (Supervisor: J. Jakubek)*, Master Thesis, CTU Prague, 2004

[7] Z. Vykydal, J. Jakůbek, *USB Lite - Miniaturized readout interface for Medipix2 detector*, Nucl. Inst. and Meth., DOI: 10.1016/j.nima.2010.06.118

[8] T. Holý, J. Jakůbek, S. Pospíšil, J. Uher, D. Vavřík, Z. Vykydal, *Data acquisition and processing software package for Medipix-2 device*, NIM A Vol. 563, pages 254-258 (2006)

[9] V. Kraus, M. Holík, J. Jakůbek, M. Kroupa, P. Soukup, Z. Vykydal, *FITPix – Fast Interface for Timepix Pixel Detectors*, JINST 6 C01079 (2011)

[10] X. Llopart, *Medipix2 User's Manual*, CERN, distributed only between members of the Medipix Collaboration, 2001

[11] X. Llopart, *TimePix Users's Manual*, CERN, distributed only between members of the Medipix Collaboration, 2006

[12] X. Llopart, *Design and characterization of 64-K pixels chips working in single photon processing mode*, PhD Thesis, Mid Sweden University, 2007

[13] R. Ballabriga, *The Design and Implementation in 0.13um CMOS of an Algorithm Permitting Spectroscopic Imaging with High Spatial Resolution for Hybrid Pixel Detectors*, Universitat Ramon Llull, 2009

[14] X. Llopart, R. Ballabriga, W. Wong, *Medipix3 User's Manual*, CERN, distributed only between members of the Medipix Collaboration, 2009

[15] R. Ballabriga, M. Campbell, E. H. M. Heijne, X. Llopart, L. Tlustos, *The Medipix3 Prototype, a Pixel Readout Chip Working in Single Photon Counting Mode with Improved Spectrometric Performance*, IEEE Trans. on Nucl. Sci., 2007, Vol. 54, pp. 1824-1829

[16] M. Maiorino, *Marino Maiorino - Medisoft 3 Software* [online]. 2010 [cit. 2011-04-10], Available at: http://personal.ifae.es/maiorino/Medisoft3.html

[17] J. Jakůbek, Semiconductor Pixel Detectors and their Applications in Life Sciences, JINST 4 P03013 (2009)

[18] T. Holý, *Pixelman architecture* [online]. 2008 [cit. 2011-04-10], Available at: http://aladdin.utef.cvut.cz/ofat/others/Pixelman/download/Pixelman_architecture_1.0.pdf

[19] E. Bertolucci, M. Maiorino, G. Mettivier, M. C. Montesi, P. Russo, *Medisoft 4: A Software Procedure for the Control of the Medipix2 Readout Chip*, IEEE Nuclear Science Symposium and Medical Imaging Conference 4–10 Nov. 2001, San Diego, CA, Vol. 2, pp. 714–718.

[20] FTDI Ltd., *Software Application Development D2XX Programmer's Guide* [online] 2011. [cit. 2011-04-10], Available at: http://www.ftdichip.com/Support/Documents/ProgramGuides/D2XX_Programmer's_Guide(FT_000071).pdf

[21] *SWIG-2.0 Documentation* [online] 2011-03 [cit. 2011-04-10], Available at: http://www.swig.org/Doc2.0/SWIGDocumentation.pdf

[22] *Java Native Access(JNA): Pure Java Access to Native Libraries - Java.net* [online]. 2011 [cit. 2011-04-10]. Available at: http://jna.java.net/

[23] Sheng Liang, *The JavaTM Native Interface. Programmer's Guide and Specification* [online]. 1999 [cit. 2011-04-10].

[24] *MiGLayout - The Java Layout Manager for Swing, SWT and JavaFX* [online]. 2007 [cit. 2011-04-10]. Available at: http://www.miglayout.com/

[25] Z. Vykydal, J. Bouchami, M. Campbell, Z. Doležal, M. Fiederle, D. Greiffenberg, A. Gutierrez, E. Heijne, T. Holý, J. Idarraga, J. Jakůbek, V. Král, M. Králík, C. Lebel, C. Leroy, X. Llopart, D. Maneuski, M. Nessi, V. O/'Shea, M. Platkevič, S. Pospíšil, M. Suk, L. Tlustos, P. Vichoudis, J. Visschers, I. Wilhelm, J. Žemlička, *The Medipix2-Based Network for Measurement of Spectral Characteristics and Composition of Radiation in ATLAS Detector*, NIM A, Vol. 607, Issue 1, p. 35-37 (2009)

[26] D. Turecek, T. Holy, S. Pospisil, Z. Vykydal: *Remote control of ATLAS-MPX Network and Data Visualization*, Nucl. Inst. and Meth., DOI: 10.1016/j.nima.2010.06.117.

[27] T. Holý, E. Heijne, J. Jakůbek, S. Pospíšil, J. Uher, Z. Vykydal, *Pattern recognition of tracks induced by individual quanta of ionizing radiation in Medipix2 silicon detector*, NIM A, Vol. 591, Issue 1, p. 287-290 (2008)

[28] B. Eckel, *Thinking in C++*, 814 p., Prentice Hall, 2004, ISBN 0-13-035313-2.

[29] B. Eckel, *Thinking in Java*, 1482 p., Prentice Hall, 2006, ISBN ISBN 0-13-0187248-6.

A Java Plugin Interface

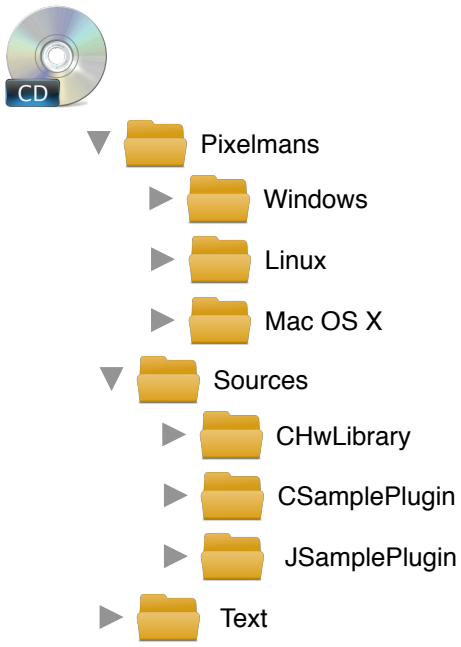
```
public interface JPluginInfoInterface {

    /** object is not assigned to specified device, but it's general
       for whole plugin*/
    public final static int ALL_DEVICES = -1;

    /**
     * Gets short name of the plugin (e.g. jmpxctrlui)
     * @return
     */
    public String getName();

    /**
     * Gets  path and file name of the plugin
     * (e.g. /home/user/Pixelman/jplugins/jmpxctrlui.jar)
     * @return plugin path and filename
     */
    public String getPath();

    /**
     * Gets Object (ObjectInfo) with name objectName for all/no device
         (ALL_DEVICES).
     * @param objectName object name to retrieve
     * @return ObjectInfo or if object not found null.
     */
    public ObjectInfo getObject(String objectName);

    /**
     * Gets object (ObjectInfo) with object name for specified device.
     * @param deviceIndex index of the device or ALL_DEVICES
     * @param objectName name of the object to retrieve
     * @return ObjectInfo or if object not found null.
     */
    public ObjectInfo getObject(int deviceIndex, String objectName);
}
```

B Sample Java Plugin

```
package cz.ieap.pixelman.sampleplugin;

import cz.ieap.jplugininterface.JPluginInterface;
import cz.ieap.jplugininterface.JPluginsManagerInterface;
import cz.ieap.jplugininterface.PixelmanFuncsInterface;
import cz.ieap.jplugininterface.callbacks.PluginFuncInterface;
import cz.ieap.jplugininterface.structs.ByteBufferRef;
import cz.ieap.jplugininterface.structs.DataTypes;
import cz.ieap.jplugininterface.structs.ExtFunctionInfo;
import cz.ieap.jplugininterface.structs.ExtParamInfo;

public class SamplePlugin implements JPluginInterface{

    public final static String PLUGIN_NAME  = "sampleplugin";

    /**
     * <p>Return name of the plugin that implements this interface.
     * This function should return valid value even if it is called
     * before initialize plugin. </p>
     * <p>Plugin name should contain only these characters: a-z A-Z
        0-9 _</p>
     * @return name of the plugin
     */
    public String getPluginName() {
        return PLUGIN_NAME;
    }



    /**
     * Initialize plugin.
     * @param pixelmanFuncsInterface interface containing pixelman
        functions.
     * @param temporaryDirectory directory where files from jar of
        plugin will be unpacked.
     * this means that resources that might be in jar will be in this
        directory
     * @param nativeLibDirectory directory where native libraries
        should be,
     * if plugin needs them. If null, native libraries are located in
        temporaryDirectory.
     * @return if plugin is initialized successfully, should return 0.
     * if error occured should return non-zero value.
     */
```

```
    public int initializePlugin(PixelmanFuncsInterface
       pixelmanFuncsInterface, String temporaryDirectory, String
       nativeLibDirectory)
    {
        pixelmanFuncsInterface.logMsg(PLUGIN_NAME, "Hello world from
           Java Plugin.", 0);
        pixelmanFuncsInterface.addMenuItem(PLUGIN_NAME, "Test", "",
           null, 0, null, 0);

        return 0;
    }


    /**
     * Called when JMpxLoader is fully loaded and all plugins are
        loaded and initialized.
     */
    public void startPlugin() {

    }


    /**
     * Called when pixelman exits for plugin to clean up resources etc
        .
     */
    public void finalizePlugin() {

    }

    /**
     * Sets JPluginsManagerInterface, which plugin may use to get
        information about
     * othe plugins. This method is called before initializing of
        plugin, so it can be
     * used in initializePlugin method.
     * @param pluginsManagerInterface plugin's manager interface
     */
    public void setPluginsManagerInterface(JPluginsManagerInterface
       pluginsManagerInterface) {

    }

    public int getAPIVersion() {
        return API_VERSION;
    }
}
```

C Content of the Attached CD



Pixelmans ... Standard build of Pixelman for Windows, Linux and Mac OS X
Sources ... Sample plugins and hardware library source codes
Text ... Electronic version of this thesis and LaTeX source code