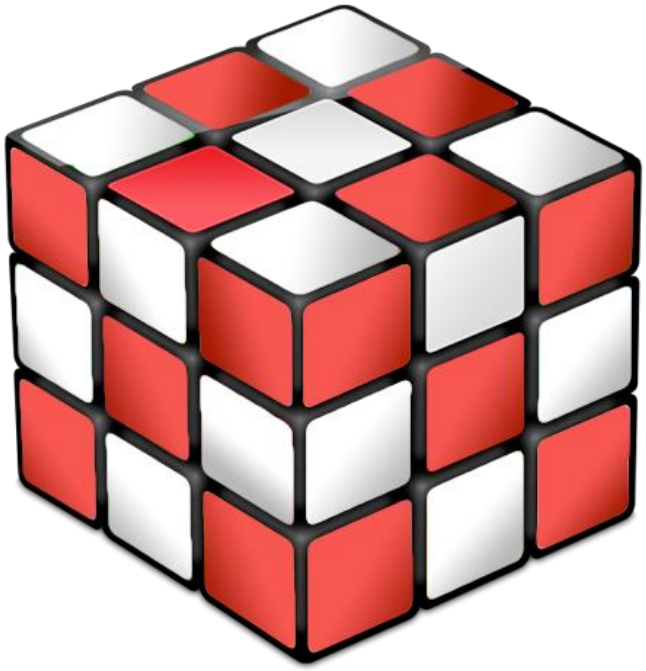
ADVACAM



PIXET

Developer Documentation

Table of Contents

1. Introduction

The **PIXet** is a multi-platform software developed in ADVACAM company. It is a basic software that allows measurement control and saving of measured data with Medipix detectors. It supports Medipix2, Medipix3, Timepix and Timepix3 detectors and all the readout-devices sold by ADVACAM company such as FitPIX, AdvaPIX, WidePIX, etc. It is written in C++ language and uses multi-platform Qt libraries.

This document describes a developer interface of the **PIXet** software. This developer interface consists of dynamic linked library **pxcore.dll** (Windows) or **libpxcore.so** (Mac or Linux), the corresponding header file for the library **pxcapi.h** and few other supporting libraries (fitpix.dll, Visual Studio runtime libraries, etc.).

2. Requirements

2.1 Hardware

This API requires computer with x86 compatible architecture (**no ARM**), 64bit Windows or Linux and connected some Advacam hardware with imaging chip. Medipix3, Timepix, Timepix2, Timepix3, etc. Some functions are universal for all hardwares (pxcInitialize, pxcGetDeviceName, etc), some is specialized for only one chip type (pxcMeasureSingleFrameTpx3 is Timepix3 only).

Specialized functions have names with chip type included:

pxcSet**Timepix**CalibrationEnabled – Timepix only (no Timepix3)
pxcMeasure**Tpx3**DataDrivenMode – Timepix3 only
pxcMeasureSingleFrame**Mpx3** – Medipix3 only

The attempt to use the function if compatible hardware (in initialized state) not present, end with error.
Return code is **PXCERR_UNEXPECTED_ERROR**.

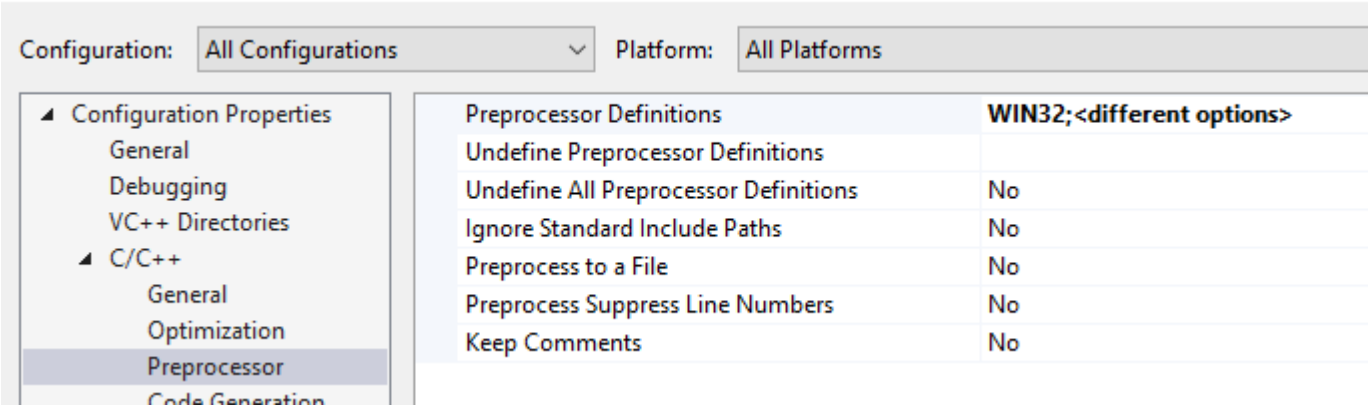
2.2 Software

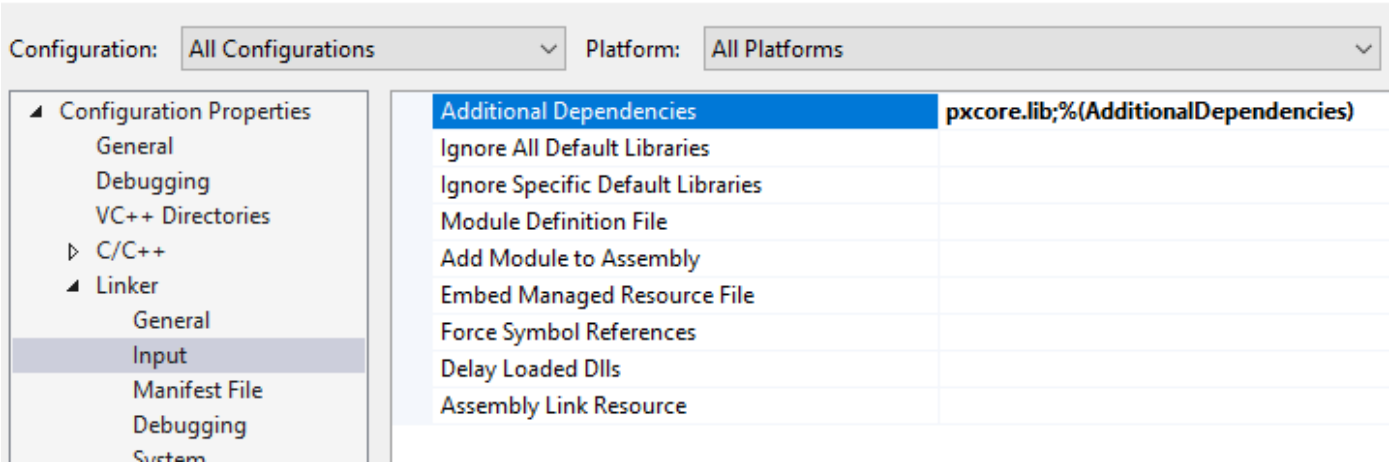
All the API functions have heads in **pxcapi.h**, implemented for Windows in the **pxcore.dll** and for linking must use the **pxcore.lib** in the linker settings. Implementation for Linux is in the **libcore.so**. The library is **64bit only**.

Compiled program need the **pixet.ini** file with proper hwlibs list inside, necessary hardware dll files (eq **minipix.dll** for Minipixes), optional special files (eq **zestwpx.bit** for Widepixes), subdirectory "**factory**" with default config files for all present imaging devices (eq MiniPIX-I08-W0060.xml) and the Pixet core will create subdirectory "**configs**" to save changed configs on exit.

Usually, for build, just set the compiler to use 64bit and the linker to use the pxcore.lib file.

In Microsoft visual studio, it is also necessary to insert the use of WIN32 definition into the project settings (C/C++ / Preprocessor / Preprocessor definitions):





2.2.1 The Pixet core, additional libraries and other files

pxcore.dll (standard API), **pxproc.dll** (clustering API) or equivalent SO files on Linux
pxcore.lib, **pxproc.lib** (for compile only, Windows only)

The pixet.ini and the hwlibs

In the active directory must be the pixet.ini file. It must contains the [hwlibs] section with list of hwlib DLLs (or SOs) for devices that your project may supports. The hwlib files must be located in the locations specified in the pixet.ini. A semicolon at the beginning of a line disables the line.

Example1:
```
[Hwlibs]
hwlibs\minipix.dll
hwlibs\zest.dll
```

Example2:
```
[Hwlibs]
minipix.dll
zest.dll
;zem.dll
```

(Examples is for a Minipix and a Widepix with Ethernet)
Example1 is for hwlibs in the "hwlibs" subdirectory.
Example2 is for hwlibs with all files in the active directory and with Advapix disabled.

Hwlibs list:		
	Minipix:	minipix.dll
	Widepix with Eth:	zest.dll
	Widepix without Eth:	widepix.dll
	Advapix:	zem.dll

Additional files for some devices

Device INI files list:	Minipix:	minipix.ini
	Widepix with Eth:	zest.ini
	Widepix without Eth:	widepix.ini
Firmware images list:	Widepix-L:	zestwpx.bit
	Widepix-F:	zemwpxf.rbf
	Advapix-Tpx3:	zemtpx3.rbf
	Advapix-Tpx3-quad:	zemtpx3quad.rbf
	Advapix-Timepix:	zemtpx.rbf

The factory and the configs directories

The **factory** directory should contain the factory default configuration XML files. The Pixet core use it while starting, if the configuration file is not in the configs directory. This directory not need if the device has an internal config memory (Minipix for example).
The **configs** directory contain configuration XML files. The Pixet core try to use it while starting and automatically save the current settings to it, if exiting.

This process works the same way when you start and quit the Pixet program.

3. API Functions

3.1 Auxiliary functions

3.1.1 Start-up and end

3.1.1.1 pxcInitialize

Summary

This function initializes the Pixet software and all connected devices. This function has to be called first before any other function except **pxcGetLastError**.

Definition

PXCAPI int **pxcInitialize**(int argc = 0, char const* argv[] = NULL)

Parameters

argc – number of program command line arguments (optional parameter)
argv – command line program arguments (optional parameter)

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code.

3.1.1.2 pxcExit

Summary

This function deinitializes Pixet software and all the connected devices. This function has to be called as last function before unloading the pxcore library.

Definition

PXCAPI int **pxcExit**()

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.1.1.3 pxcRefreshDevices

Summary

This function looks for newly connected devices and removed disconnected devices from the device list.

Definition

PXCAPI int **pxcRefreshDevices**()

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code.

3.1.1.4 pxcReconnectDevice

Summary

If the device was disconnected or experienced communication problems, this function will try to reconnect the device and reinitialize it.

Definition

PXCAPI int **pxcReconnectDevice**(unsigned deviceIndex)

Parameters

deviceIndex - index of the device, starting from zero

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code.

3.1.1.5 Example

```
#define CHT_Si      0
#define CHT_CdTe    1
char chipType = CHT_Unknown;
int main (int argc, char const* argv[]) { // ######################################
    int rc = pxcInitialize();
    if (rc) {
        printf("Could not initialize Pixet:\n");
        printErrors("pxcInitialize", rc, ENTER_ON);
        return -1;
    }
    int connectedDevicesCount = pxcGetDevicesCount();
    printf("Connected devices: %d\n", connectedDevicesCount);
    if (connectedDevicesCount == 0) return pxcExit();

    for (unsigned devIdx = 0; (signed)devIdx < connectedDevicesCount; devIdx++){
        char deviceName[256];
        memset(deviceName, 0, 256);
        pxcGetDeviceName(devIdx, deviceName, 256);

        char chipID[256];
        memset(chipID, 0, 256);
        pxcGetDeviceChipID(devIdx, 0, chipID, 256);
        printf("Device %d: Name %s, (first ChipID: %s)\n", devIdx, deviceName, chipID);
    }
    double bias;
    rc = pxcGetBias(devIdx, &bias);
    if (bias<0.0) {
        if (devIdx==0) chipType = CHT_CdTe;
        printf("Chip material detected: CdTe\n");
    } else if (bias==0.0) {
        printf("Chip material not detected!\n");
    } else {
        if (devIdx==0) chipType = CHT_Si;
        printf("Chip material detected: Si\n");
    }
    printf("================================================================\n");

    // here can be working code (calling some example function from this manual)

    return pxcExit();
}
```

3.1.2 Parameter Get/Set functions (direct)

Functions described in this chapter working directly, function name defines parameter name and type.

Example:

```
// Set the operating mode
rc = pxcSetTimepix3Mode(deviceIndex, PXC_TPX3_OPM_TOATOT);
printErrors("pxcSetTimepix3Mode", rc);
```

3.1.2.1 pxcGetLastError

Summary

Returns text of last error. This function can be called even before pxcInitialize()

Definition

PXCAPI int **pxcGetLastError**(char* errorMsgBuffer, unsigned size)

Parameters

errorMsgBuffer - buffer where text will be saved
size - size of supplied buffer

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Example

```
#define ERRMSG_BUFF_SIZE    512
#define ENTER_ON            true
#define ENTER_OFF           false

// (the function used in most examples in this manual)
void printErrors(const char* fName, int rc, bool enter) {
    char errorMsg[ERRMSG_BUFF_SIZE];
    pxcGetLastError(errorMsg, ERRMSG_BUFF_SIZE);
    if (errorMsg[0]>0) {
        printf("%s %d err: %s", fName, rc, errorMsg);
    } else {
        printf("%s %d err: ---", fName, rc);
    }
    if (enter) printf("\n");
}
```

Now you can use it:

```
int mode = PXC_TPX3_OPM_EVENT_ITOT;
rc = pxcSetTimepix3Mode(deviceIndex, mode);
printErrors("pxcSetTimepix3Mode", rc, ENTER_ON);
```

If mode set was suscessfull, result is:

pxcSetTimepix3Mode 0 err: ---

If not, you can see some as:

pxcSetTimepix3Mode -2 err: Invalid device index

3.1.2.2 pxcGetDevicesCount

Summary

This function returns number of connected and initialized devices.

Definition

PXCAPI int **pxcGetDevicesCount**()

Return Value

Number of devices, otherwise the return value is a PXCERR_XXX code

3.1.2.3 pxcGetDeviceName

Summary

This function returns the full name of the selected device.

Definition

PXCAPI int **pxcGetDeviceName**(unsigned deviceIndex, char* nameBuffer, unsigned size)

Parameters

deviceIndex - index of the device, starting from zero
nameBuffer - buffer where the name of the device will be saved. Cannot be NULL
size - size of the supplied name buffer

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code.

3.1.2.4 pxcGetDeviceChipCount

Summary

This function returns number of chips in the device.

Definition

PXCAPI int **pxcGetDeviceChipCount**(unsigned deviceIndex)

Parameters

deviceIndex - index of the device, starting from zero

Return Value

Number of chips if successful, otherwise the return value is a PXCERR_XXX code

3.1.2.5 pxcGetDeviceChipID

Summary

This function returns the ID of chip of the detector connected to the readout device.

Definition

PXCAPI int **pxcGetDeviceChipID**(unsigned deviceIndex, unsigned chipIndex, char* chipIDBuffer, unsigned size)

Parameters

deviceIndex - index of the device, starting from zero
chipIndex – index of the chip in the device, starting from zero
chipIDBuffer - buffer where the chipID of the detector will be saved. Cannot be NULL
size - size of the supplied chipID buffer

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.1.2.6 pxcGetBias

Summary

This function gets the high voltage (bias voltage) of the sensor on Medipix/Timepix chip.

Definition

PXCAPI int **pxcGetBias**(unsigned deviceIndex, double* bias)

Parameters

deviceIndex - index of the device, starting from zero
bias – pointer to double variable where current bias will be returned

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.1.2.7 pxcGetBiasRange

Summary

This function gets the range of the allowed minimal and maximal bias values.

Definition

PXCAPI int **pxcGetBiasRange**(unsigned deviceIndex, double* minBias, double* maxBias)

Parameters

deviceIndex - index of the device, starting from zero
minBias – pointer to double variable where minimum allowed bias will be returned
maxBias – pointer to double variable where maximum allowed bias will be returned

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.1.2.8 pxcSetBias

Summary

This function sets the high voltage (bias) of the detector.

Definition

PXCAPI int **pxcSetBias**(unsigned deviceIndex, double bias)

Parameters

deviceIndex - index of the device, starting from zero
bias – high voltage in volts (0 to 100 V)

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.1.2.9 pxcGetThreshold

Summary

This function gets the threshold of the Medipix/Timepix detector in detector DAC values.

Definition

PXCAPI int **pxcGetThreshold**(unsigned deviceIndex, unsigned thresholdIndex,

double* threshold)

Parameters

deviceIndex - index of the device, starting from zero
thresholdIndex – for Timepix and Timepix3 always 0, for Medipix3 index of corresponding threshold starting from zero
threshold – pointer to double variable where threshold will be saved. detector threshold (0 to 1024). The sense is reversed, for higher threshold lower number and the other way around

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.1.2.10 pxcGetThresholdRange

Summary

This function gets the allowed range of values for threshold

Definition

PXCAPI int **pxcGetThresholdRange**(unsigned deviceIndex, int thresholdIndex, double* minThreshold, double* maxThreshold)

Parameters

deviceIndex - index of the device, starting from zero
thresholdIndex – for Timepix and Timepix3 always 0, for Medipix3 index of corresponding threshold starting from zero
minThreshold – pointer to double variable where the minimal allowed threshold will be returned
maxThreshold – pointer to double variable where the maximal allowed threshold will be returned

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.1.2.11 pxcSetThreshold

Summary

This function sets the threshold of the detector in KeV.

Definition

PXCAPI int **pxcSetThreshold**(unsigned deviceIndex, unsigned thresholdIndex, double threshold)

Parameters

deviceIndex - index of the device, starting from zero
thresholdIndex - for Timepix and Timepix3 always 0, for Medipix3 index of corresponding threshold starting from zero
threshold – detector threshold in keV.

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.1.2.12 pxcGetDAC

Summary

This function gets a single DAC value of the detector.

Definition

PXCAPI int **pxcGetDAC**(unsigned deviceIndex, unsigned chipIndex, unsigned dacIndex, unsigned short* value);

Parameters

deviceIndex - index of the device, starting from zero
chipIndex – index of the chip, starting from zero
value – returned value

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.1.2.13 pxcSetDAC

Summary

This function sets a single DAC value of the detector.

Definition

PXCAPI int **pxcSetDAC**(unsigned deviceIndex, unsigned chipIndex, unsigned dacIndex, unsigned short value);

Parameters

deviceIndex - index of the device, starting from zero
chipIndex – index of the chip, starting from zero
value – new DAC value

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.1.2.14 pxcGetTimepixClock

Summary

This function gets the current value of measurement clock for Timepix detector (in MHz).

Definition

PXCAPI int **pxcGetTimepixClock**(unsigned deviceIndex, double* clock)

Parameters

deviceIndex - index of the device, starting from zero
clock – pointer to double variable where the clock will be saved

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Note

Only for the Timepix devices, not usable on Timepix3 and other TimepixN. Timepix3 have a fixed clock of 40 MHz.

3.1.2.15 pxcSetTimepixClock

Summary

This function sets the value of measurement clock for Timepix detector (in MHz).

Definition

PXCAPI int **pxcSetTimepixClock**(unsigned deviceIndex, double clock)

Parameters

deviceIndex - index of the device, starting from zero
clock – new value of the measurement clock for Timepix detector

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Note

Only for the Timepix devices, not usable on Timepix3 and other TimepixN. Timepix3 have a fixed clock of 40 MHz.

3.1.2.16 pxcGetTimepixMode

Summary

This function gets the current value of the Timepix mode (Counting, Energy,...)

Definition

PXCAPI int **pxcGetTimepixMode**(unsigned deviceIndex)

Parameters

deviceIndex - index of the device, starting from zero

Return Value

Timepix mode if successful, otherwise the return value is a PXCERR_XXX code.
Timepix mode can be:
PXC_TPX_MODE_MEDIPIX – counting mode
PXC_TPX_MODE_TOT – energy mode
PXC_TPX_MODE_TIMEPIX – timepix mode

Note

Only for the Timepix devices, not usable on Timepix3 and other TimepixN.

3.1.2.17 pxcSetTimepixMode

Summary

This function sets the value of Timepix mode

Definition

PXCAPI int **pxcSetTimepixMode**(unsigned deviceIndex, int mode)

Parameters

deviceIndex - index of the device, starting from zero
mode – new value of the Timepix mode. One of the values:
PXC_TPX_MODE_MEDIPIX – counting mode
PXC_TPX_MODE_TOT – energy mode
PXC_TPX_MODE_TIMEPIX – timepix mode

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Note

Only for the Timepix devices, not usable on Timepix3 and other TimepixN.

3.1.2.18 pxcSetTimepixCalibrationEnabled

Summary

This function enables or disables the calibration of Timepix ToT counts to energy in keV

Definition

PXCAPI int **pxcSetTimepixCalibrationEnabled**(unsigned deviceIndex, bool enabled)

Parameters

deviceIndex - index of the device, starting from zero
enabled – if the calibration is enabled or disable

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Note

Only for the Timepix devices, not usable on Timepix3 and other TimepixN.

3.1.2.19 pxcIsTimepixCalibrationEnabled

Summary

This function returns if the calibration of Timepix ToT counts to energy in keV is enabled

Definition

PXCAPI int **pxcIsTimepixCalibrationEnabled**(unsigned deviceIndex)

Parameters

deviceIndex - index of the device, starting from zero

Return Value

0 if disabled, greater than o enabled, negative value a PXCERR_XXX code

Note

Only for the Timepix devices, not usable on Timepix3 and other TimepixN.

3.1.2.20 pxcGetTimepix2Clock

Summary

This function gets the current clocks settings in the Timepix2 detector.

Definition

PXCAPI int **pxcGetTimepix2Clock**
(unsigned deviceIndex, double* totClock, double* toaClock, unsigned* divider)

Parameters

deviceIndex - index of the device, starting from zero
totClock – pointer to double variable where the ToT clock (in MHz) will be saved
toaClock – pointer to double variable where the ToA clock (in MHz) will be saved
divider – pointer to unsigned int variable where the divider value will be saved

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Note

Only for the Timepix2 devices, not usable on Timepix3 and other Timepix. Timepix3 have a fixed clock of 40 MHz.

3.1.2.21 pxcSetTimepix2Clock

Summary

This function sets Timepix2 detector clocks settings.

Definition

PXCAPI int **pxcSetTimepix2Clock**(unsigned deviceIndex, double clock, unsigned divider)

Parameters

deviceIndex - index of the device, starting from zero
clock – desired new value of the ToT clock (in MHz) for the Timepix2 detector. The real frequency will be nearest possible division by 2's power from the 50 MHz. Min is 1.5625 MHz.
Warning: The factory energy calibration is only for the 50 MHz ToA clock.
divider – value of the ToA divider index.
Values means 0: disable, 1: no division, 2-30: div. by $2^{(n-1)}$.
The ToA clock will be divided from the ToT clock.

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Note

Only for the Timepix2 devices, not usable on Timepix3 and other Timepix. Timepix3 have a fixed clock of 40 MHz.

3.1.2.22 pxcSetTimepix2Mode

Summary

This function sets the value of Timepix2 mode

Definition

PXCAPI int **pxcSetTimepix2Mode**(unsigned deviceIndex, int mode)

Parameters

deviceIndex - index of the device, starting from zero
mode – new value of the Timepix2 mode. One of the values:
PXC_TPX2_OPM_TOT10_TOA18
PXC_TPX2_OPM_TOT14_TOA14
PXC_TPX2_OPM_CONT_TOT10_CNT4
PXC_TPX2_OPM_CONT_TOT14
PXC_TPX2_OPM_CONT_TOA10
PXC_TPX2_OPM_CONT_TOA14
PXC_TPX2_OPM_CONT_CNT10
PXC_TPX2_OPM_CONT_CNT14
PXC_TPX2_OPM_ITOT10_TOA18
PXC_TPX2_OPM_ITOT14_TOA14
PXC_TPX2_OPM_CONT_ITOT10_CNT4
PXC_TPX2_OPM_CONT_ITOT14
TOT – time of threshold in ToT ticks, or energy if calibrated
TOA – time of arrival in ToA ticks
CNT – count of hits
ITOT – integrated time of threshold in the pixel, or estimate energy if calibrated
CONT – continual mode: One counter set counting while reading data from other counter set

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Note

Only for the Timepix2 devices, not usable on Timepix3 and other Timepix.

3.1.2.23 pxcSetTimepix2AdaptiveGainMode

Summary

This function sets the value of Timepix2 mode

Definition

PXCAPI int **pxcSetTimepix2AdaptiveGainMode**(unsigned deviceIndex, bool adaptiveGainOn)

Parameters

deviceIndex - index of the device, starting from zero
adaptiveGainOn – enable the adaptive gain feature

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Note

Only for the Timepix2 devices, not usable on Timepix3 and other Timepix.

3.1.2.24 pxcSetTimepix2AnalogueMaskingMode

Summary

This function sets the value of Timepix2 mode

Definition

PXCAPI int **pxcSetTimepix2AnalogueMaskingMode**(unsigned deviceIndex, bool analogMaskOn)

Parameters

deviceIndex - index of the device, starting from zero
analogMaskOn – enable the analogue masking feature

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Note

Only for the Timepix2 devices, not usable on Timepix3 and other Timepix.

3.1.2.25 pxcSetTimepix2CalibrationEnabled

Summary

This function enables or disables the calibration of Timepix ToT counts to energy in keV

Definition

PXCAPI int **pxcSetTimepix2CalibrationEnabled**(unsigned deviceIndex, bool enabled)

Parameters

deviceIndex - index of the device, starting from zero
enabled – if the calibration is enabled or disable

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Note

You need different acquisition or frame reading functions if calibration is on or off.
Only for the Timepix2 devices, not usable on Timepix3 and other Timepix.

3.1.2.26 pxcIsTimepix2CalibrationEnabled

Summary

This function returns if the calibration of Timepix ToT counts to energy in keV is enabled

Definition

PXCAPI int **pxcIsTimepix2CalibrationEnabled**(unsigned deviceIndex)

Parameters

deviceIndex - index of the device, starting from zero

Return Value

0 if disabled, greater than o enabled, negative value a PXCERR_XXX code

Note

You need different acquisition or frame reading functions if calibration is on or off.
Only for the Timepix2 devices, not usable on Timepix3 and other Timepix.

3.1.2.27 pxcSetTimepix3Mode

Summary

Sets the operation mode of Timepix3 detector

Definition

PXCAPI int **pxcSetTimepix3Mode**(unsigned deviceIndex, int mode)

Parameters

deviceIndex - index of the device, starting from zero
mode – mode of the detector PXC_TPX3_OPM_XXX values

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Note

Only for the Timepix3 devices, not usable on other Timepixes or Medipixes.

3.1.2.28 pxcSetMedipix3OperationMode

Summary

Sets the operation mode of Medipix3 detector

Definition

PXCAPI int **pxcSetMedipix3OperationMode**(unsigned deviceIndex, int opMode)

Parameters

deviceIndex - index of the device, starting from zero
opMode – mode of the detector PXC_MPX3_OPM_XXX values

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Note

Only for the Medipix3 devices, not usable on Timepixes or other Medipixes.

3.1.2.29 pxcSetMedipix3GainMode

Summary

Sets the gain mode of Medipix3 detector

Definition

PXCAPI int **pxcSetMedipix3GainMode**(unsigned deviceIndex, int gain)

Parameters

deviceIndex - index of the device, starting from zero
gain – mode of the detector PXC_MPX3_GAIN_MOD_XXX values

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Note

Only for the Medipix3 devices, not usable on Timepixes or other Medipixes.

3.1.2.30 pxcSetMedipix3AcqParams

Summary

Sets acquisition parameters for Medipix3

Definition

PXCAPI int **pxcSetMedipix3AcqParams**(unsigned deviceIndex, bool colorMode,
bool csm, int gain, bool equalize)

Parameters

deviceIndex - index of the device, starting from zero
colorMode – if color mode is enabled
csm – if charge sharing mode is enabled

gain – gain settings (PXC_MPX3_GAIN_XXX values)
equalize – if equalization bit in Medipix3 is enabled

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Note

Only for the Medipix3 devices, not usable on Timepixes or other Medipixes.

3.1.2.31 pxcSetMedipix3MatrixParams

Summary

Sets parameters of the Meidpix3 pixel matrix

Definition

PXCAPI int **pxcSetMedipix3MatrixParams**(unsigned deviceIndex, int depth,
int counter, int colBlock, int rowBlock)

Parameters

deviceIndex - index of the device, starting from zero
depth – depth of the counters PXC_MPX3_CNTD_XXX values
counter – selected counter (PXC_MPX3_CNT_XXX values)
colBlock – region of interest readout (PXC_MPX3_COLB_XXX values)
rowBlock – region of interest readout (PXC_MPX3_ROWB_XXX values)

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Note

Only for the Medipix3 devices, not usable on Timepix or other Medipixes.

3.1.2.32 pxcSetPixelMatrix

Summary

Sets the pixel matrix configuration. This is low level function for advanced users.

Definition

PXCAPI int **pxcSetPixelMatrix** (unsigned deviceIndex, unsigned char* maskMatrix,
unsigned size)

Parameters

deviceIndex - index of the device, starting from zero
maskMatrix – pixel mask matrix. 0 = masked, 1= unmasked
size – size of the mask matrix

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.1.2.33 pxcGetPixelMatrix

Summary

Gets the pixel matrix configuration. This is low level function for advanced users.

Definition

PXCAPI int **pxcGetPixelMatrix** (unsigned deviceIndex, unsigned char* maskMatrix, unsigned byteSize)

Parameters

deviceIndex - index of the device, starting from zero
maskMatrix – buffer where the mask matrix will be stored. 0=masked, 1=unmasked
size – size of the mask matrix

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.1.2.34 Example: Device detect

This API currently have no device detection function. If You need it, You can try to use specialized functions:

```
std::string detectDev(int devIdx) {
	int rc; // return code
	rc = pxcSetTimepix3Mode(devIdx, 0);
	if (rc==0) return "Tpx3";
	rc = pxcSetTimepix2Mode(devIdx, 0);
	if (rc==0) return "Tpx2";
	rc = pxcSetTimepixMode(devIdx, 0);
	if (rc==0) return "Tpx";
	rc = pxcSetMedipix3OperationMode(devIdx, 0);
	if (rc==0) return "Mpx3";
	return "(unknown)";
}
```

3.1.3 Parameter Get/Set functions (using text paramName)

In this chapter are a functions that working with named parameters. Example:

```
// Data Driven Block Size [B], default 66000
rc = pxcSetDeviceParameter(deviceIndex, "DDBlockSize", 6000);
printf("pxcSetDeviceParameter %d", rc);
```

Warning: Most parameters are for testing purposes only and you will not need them in normal use.

3.1.3.1 pxcGetDeviceParameter

Summary

Returns the value of integer device parameter (e.g. settings of trigger)

Definition

PXCAPI int **pxcGetDeviceParameter**(unsigned deviceIndex, const char* parameterName)

Parameters

deviceIndex - index of the device, starting from zero
parameterName – name of the device parameter

Return Value

Value of the device parameter or PXCERR_XXX code if error occurs

3.1.3.2 pxcSetDeviceParameter

Summary

Sets a value of the integer device parameter (e.g. settings of trigger)

Definition

PXCAPI int **pxcSetDeviceParameter**(unsigned deviceIndex, const char* parameterName,
Int parameterValue)

Parameters

deviceIndex - index of the device, starting from zero
parameterName – name of the device parameter
parameterValue – new value of the parameter

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.1.3.3 pxcGetDeviceParameterDouble

Summary

Returns the value of device double parameter

Definition

PXCAPI int **pxcGetDeviceParameterDouble**(unsigned deviceIndex,
const char* parameterName, double* parameterValue)

Parameters

deviceIndex - index of the device, starting from zero
parameterName – name of the device parameter
parameterValue – pointer to double variable where the parameter value will be saved

Return Value

PXCERR_XXX code if error occurs

3.1.3.4 pxcSetDeviceParameterDouble

Summary

Sets a value of the device double parameter

Definition

PXCAPI int **pxcSetDeviceParameterDouble**(unsigned deviceIndex, const char* parameterName,
double parameterValue)

Parameters

deviceIndex - index of the device, starting from zero
parameterName – name of the device parameter
parameterValue – new value of the parameter

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.1.3.5 pxcGetDeviceParameterString

Summary

Returns the value of device string parameter

Definition

PXCAPI int **pxcGetDeviceParameterString**(unsigned deviceIndex,
const char* parameterName, const char* parameterValue, unsigned size)

Parameters

deviceIndex - index of the device, starting from zero
parameterName – name of the device parameter
parameterValue – pointer to string buffer where the parameter value will be saved
size – size of the passed buffer

Return Value

PXCERR_XXX code if error occurs

3.1.3.6 pxcSetDeviceParameterString

Summary

Sets a value of the device string parameter

Definition

PXCAPI int **pxcSetDeviceParameterString**(unsigned deviceIndex, const char* parameterName,
const char* parameterValue)

Parameters

deviceIndex - index of the device, starting from zero
parameterName – name of the device parameter
parameterValue – new value of the parameter

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.1.3.7 Tpx3 parameter names list

Warning: Most parameters are for testing purposes only and you will not need them in normal use.

```
#define PAR_LIBVER              "HwLibVer"
#define PAR_DEBUGLOG            "DebugLog"
#define PAR_DUMMYACQ            "DummyAcqNegativePolarity"
#define PAR_TEMP                "Temperature"
#define PAR_TEMP_CHIP           "TemperatureChip"
#define PAR_TEMP_CPU            "TemperatureCpu" // mimipix/tpx3 only
#define PAR_TEMP_CHIP_CPU       "TemperatureChipCpu" // mimipix/tpx3 only
#define PAR_TEMP_READ_ACQSERIE  "TemperatureReadBeforeAcqSerie" // mimipix/tpx3 only
#define PAR_TEMP_READ_EVERYACQ  "TemperatureReadBeforeEachAcq" // mimipix/tpx3 only
#define PAR_TEMP_CHECK_IN_SW    "CheckMaxTempInSW" // mimipix/tpx3 only
#define PAR_TEMP_CHECK_IN_CPU   "CheckMaxChipTempInCPU" // mimipix/tpx3 only
#define PAR_TEMP_MAX_ALLOWED_TEMP   "MaxAllowedChipTemp" // mimipix/tpx3 only
#define PAR_DAC_BANGAP          "DacBandGap"
#define PAR_DAC_TEMP            "DacTemp"
#define PAR_BIAS_SENSE_VOLT     "BiasSenseVoltage"
#define PAR_BIAS_SENSE_CURR     "BiasSenseCurrent"
#define PAR_DD_BUFF_SIZE        "DDBuffSize"
#define PAR_DD_BLOCK_SIZE       "DDBlockSize"
#define META_SHUTTER_TIME       "Shutter open time" // mimipix/tpx3 only
#define PAR_CHAN_MASK           "ChanMask" // no net/tpx3
#define PAR_READOUT_CLOCK       "ReadoutClock" // no net/tpx3
#define PAR_TRG_STG             "TrgStg"
#define PAR_TRG_TIMESTAMP       "TrgTimestamp"
#define PAR_TRG_T0SYNC_RESET    "TrgT0SyncReset"
#define PAR_TRG_READY           "TrgReady"  // no net/tpx3
#define PAR_TRG_OUTLEVEL        "TrgOutLevel" // mimipix/tpx3 only
#define PAR_TRG_OUT_ENABLE      "TrgOutEnable" // mimipix/tpx3 only
#define PAR_TRG_IS_MASTER       "IsMaster"
#define PAR_MOTOHOURS           "Motohours"  // mimipix/tpx3 only
#define PAR_MTX                 "MTX" // mimipix/tpx3 only
#define PAR_SEND_TOA_PIXELS     "SendDummyToaPixels" // mimipix/tpx3 only
#define PAR_DUMMYSPEED          "DDDummyDataSpeed"  // no mimipix/tpx3
#define PAR_BLOCKCOUNT          "BlockCount" // no mimipix/tpx3
#define PAR_PROCESSDATA         "ProcessData" // no mimipix/tpx3
#define PAR_TRG_MULTI           "TrgMulti" // no mimipix/tpx3
#define PAR_ADVAPIX_ADC         "AdvaPixADC" // no mimipix/tpx3
#define PAR_TRG_READY           "TrgReady" // zem only
#define PAR_TRG_CMOS            "TrgCmos" // zem only
#define PAR_READOUT_CLOCK       "ReadoutClock" // zem only
```

3.1.3.8 Tpx2 parameter names list

Warning: Most parameters are for testing purposes only and you will not need them in normal use.

```
const static char* PAR_LIBVER = "HwLibVer";
const static char* PAR_DEBUGLOG = "DebugLog";
const static char* PAR_BIAS_SENSE_VOLT = "BiasSenseVoltage";
const static char* PAR_BIAS_SENSE_CURR = "BiasSenseCurrent";
const static char* PAR_READOUT_CLOCK = "ReadoutClock";
const static char* PAR_TRG_STG = "TrgStg";
const static char* PAR_TRG_IS_MASTER = "IsMaster";
const static char* PAR_MOTOHOURS = "Motohours";
const static char* PAR_TEMP_CPU = "TemperatureCpu";
const static char* PAR_TEMP_MAX_ALLOWED_TEMP = "MaxAllowedChipTemp";

const static char* PAR_POWER_VOLT = "PowerSupplyVoltage";
const static char* PAR_CPU_SUPPLY_VOLT = "CPUSupplyVoltage";
const static char* PAR_CHIP_LDO_VOLT = "ChipLDOVoltage";

const static char* PAR_INPUT_CURRENT = "DeviceInputCurrent";
const static char* PAR_CHIP_CURRENT = "ChipCurrent";
```

3.1.3.9 Mpx2 parameter names list

Warning: Most parameters are for testing purposes only and you will not need them in normal use.

```
#define PAR_LIBVER        "HwLibVer" // all, include minipixes
#define PAR_DEBUGLOG      "DebugLog" // all, include minipixes
#define CFG_BINPIXCFG     "BinaryPixelCfg" // fei-minipix only

#define PAR_FIRMWARE        "Firmware" //widepix only
#define PAR_PS_COUNT        "PreShutterClockCount"
#define PAR_PS_DIVIDER      "PreShutterClockDivider"
#define PAR_PS_DELAY        "PreShutterDelayClockCount"
#define PAR_TEMP            "Temperature" // no zem
#define PAR_BIASINCPU       "BiasInCpu" // widepix only
#define PAR_TRG_STG         "TriggerStg"
#define PAR_TRG_WAITREADY   "TriggerWaitForReady"
#define PAR_TRG_MASTER      "TriggerMaster"
#define PAR_TRG_OUTLEVEL    "TriggerOutLevel"
#define PAR_TRG_ALTERNATIVE "TriggerAlternative" // fitpix only
#define PAR_TRG_TWODEVS     "TriggerTwoDevs" // fitpix only
#define PAR_BURST_DISABLE   "BurstDisable" // fitpix only

#define PAR_CPU_BIAS_SET        "*BiasSet" // widepix only
#define PAR_CPU_BIAS_VOLTSENSE  "BiasVolt" // widepix only
#define PAR_CPU_BIAS_CURRSENSE  "BiasCurr" // widepix only
#define PAR_CPU_TEMP_DET        "TempDet" // widepix only

#define PAR_FASTACQ             "FastAcq" // zem only
#define PAR_BURST_FRAME_COUNT   "BurstFrameCount" // zem only
#define PAR_PIXEL_BUFFSIZE      "PixelBuffSize" // zem only
```

3.1.3.10 Mpx3 parameter names list

Warning: Most parameters are for testing purposes only and you will not need them in normal use.

```
#define PAR_LIBVER              "HwLibVer"
#define PAR_DEBUGLOG            "DebugLog"
#define PAR_TEMP                "Temperature"
#define PAR_TRG_STG             "TriggerStg"
#define PAR_TRG_WAITREADY       "TriggerWaitForReady"
#define PAR_TRG_MASTER          "TriggerMaster"
#define PAR_TRG_OUTLEVEL        "TriggerOutLevel"
#define PAR_TRG_SERIES          "TriggerTdiSeries"
#define PAR_TDI_ROWCOUNT        "TdiRowCount"
#define PAR_BIASINCPU           "BiasInCpu"
#define PAR_BIAS_DISCHARGE      "BiasDischarge"

#define PAR_CPU_BIAS_SET        "*BiasSet"
#define PAR_CPU_BIAS_VOLTSENSE  "BiasVolt"
#define PAR_CPU_BIAS_CURRSENSE  "BiasCurr"
#define PAR_CPU_TEMP_DET        "TempDet"
```

3.1.3.11 Zest-wpxdev parameter names list

Warning: Most parameters are for testing purposes only and you will not need them in normal use.

```
const static char* PAR_LIBVER = "HwLibVer";
const static char* PAR_FIRMWARE = "Firmware";
const static char* PAR_FIRMWARE_CPU = "FirmwareCpu";
const static char* PAR_DEBUGLOG = "DebugLog";
const static char* PAR_TEMP = "Temperature";
const static char* PAR_TRG_STG = "TriggerStg";
const static char* PAR_TRG_WAITREADY= "TriggerWaitForReady";
const static char* PAR_TRG_MASTER = "TriggerMaster";
const static char* PAR_TRG_OUTLEVEL = "TriggerOutLevel";
const static char* PAR_BIAS_DISCHARGE = "BiasDischarge";
```

3.1.3.12 Zem-wpx7dev parameter names list

Warning: Most parameters are for testing purposes only and you will not need them in normal use.

```
#define PAR_LIBVER              "HwLibVer"
#define PAR_DEBUGLOG            "DebugLog"
#define PAR_PS_COUNT            "PreShutterClockCount"
#define PAR_PS_DIVIDER          "PreShutterClockDivider"
#define PAR_PS_DELAY            "PreShutterDelayClockCount"
#define PAR_ENC_PULSE_CNT       "EncoderPulseCount"
#define PAR_ENC_PULSE_DIR       "EncoderDirection"
#define PAR_ENC_PULSE_COUNTER   "EncoderPulseCounter"
```

3.1.4 pxcLoadDeviceConfiguration

Summary

This function loads device configuration from xml file

Definition

PXCAPI int **pxcLoadDeviceConfiguration**(unsigned deviceIndex, const char* filePath)

Parameters

deviceIndex - index of the device, starting from zero
filePath – path to xml configuration file

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.1.5 pxcSaveDeviceConfiguration

Summary

This function saves device configuration to xml file

Definition

PXCAPI int **pxcSaveDeviceConfiguration**(unsigned deviceIndex, const char* filePath)

Parameters

deviceIndex - index of the device, starting from zero
filePath – path to xml configuration file

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.1.6 pxcSetupTestPulseMeasurement

Summary

Enables / Disables and setups parameters of the test pulse measurements

Definition

PXCAPI int **pxcSetupTestPulseMeasurement**(unsigned deviceIndex, bool tpEnabled,
double height, double period,
unsigned count, unsigned spacing);

Parameters

deviceIndex - index of the device, starting from zero
tpEnabled - enables/disables test pulse measurement (in functions Measure..Frame(s))

height – test pulse height (0 – 1.5 V)
period – single test pulse period (1 – 256 us)
count – number of test pulses (1 – 10000)
spacing – spacing that is used during measurement (sub acquisition), good value is 4

Return Value
0 if successful, otherwise the return value is a PXCERR_XXX code

3.1.7 pxcRegisterAcqEvent

Summary
Registers an acquisition event callback that is called when corresponding event occurs

Definition
PXCAPI int **pxcRegisterAcqEvent**(unsigned deviceIndex, const char* event,
AcqFunc func, intptr_t userData)

Parameters
deviceIndex - index of the device, starting from zero
event – event name (PXC_ACQEVENT_XXX values)
func – callback function of type AcqFunc
userData – user data that are passed to callback function

Return Value
0 if successful, otherwise the return value is a PXCERR_XXX code

3.1.8 pxcUnregisterAcqEvent

Summary
Unregisters the acquisition event callback

Definition
PXCAPI int **pxcUnregisterAcqEvent**(unsigned deviceIndex, const char* event,
AcqFunc func, intptr_t userData)

Parameters
deviceIndex - index of the device, starting from zero
event – event name (PXC_ACQEVENT_XXX values)
func – callback function of type AcqFunc
userData – user data that are passed to callback function

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.1.9 pxcSetSensorRefresh

Summary

Sets the sensor refresh sequence text. The sensor refresh is used to clean the sensor of free charges. Process containing sequence of bias changes. Suitable values depend on chip manufacturing technology details.

Definition

PXCAPI int **pxcSetSensorRefresh**(unsigned deviceIndex, const char* refreshString)

Parameters

deviceIndex - index of the device, starting from zero
refreshString– sensor refresh string

refresh string defines steps with pairs of times [sec] and bias coefficients [1=100%]
(physical bias values limited to min/max chip properties, see **pxcGetBiasRange**)

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Example

```
// (devIdx, "time1, coef1; time2, coef2; time3, coef3; ...")
int rc = pxcSetSensorRefresh(0, "5, 2; 3, 1.5; 1, 1.2; 1, 1");
printErrors("pxcSetSensorRefresh", rc, ENTER_ON);
```

Example project

MiniPixTpx3-Maintenance

3.1.10 pxcDoSensorRefresh

Summary

Performs the sensor refresh (see details in pxcSetSensorRefresh)

Definition

PXCAPI int **pxcDoSensorRefresh**(unsigned deviceIndex)

Parameters

deviceIndex - index of the device, starting from zero

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.1.11 pxcEnableSensorRefresh

Summary

Enables automatic sensor refresh before each acquisition series and at periodic intervals (see details in pxcSetSensorRefresh)

Definition

PXCAPI int **pxcEnableSensorRefresh**(unsigned deviceIndex, bool enabled, double refreshTime)

Parameters

deviceIndex - index of the device, starting from zero
enabled – if automatic sensor refresh is enabled
refreshTime – sensor refresh is performed repeatedly after this time in seconds. If thime is 0, then the refresh is done only once before the measurement

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.1.12 pxcEnableTDI

Summary

Enables TDI (Time Delayed Integration) measurement (if device supports it)

Definition

PXCAPI int **pxcEnableTDI**(unsigned deviceIndex, bool enabled)

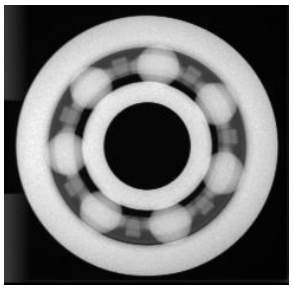
Parameters

deviceIndex - index of the device, starting from zero
enabled – if TDI is enabled

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.2 Frame-based measuring



The functions described in this chapter are used for frame-based measurements. Suitable for imaging, for example. After acquisition ends, you can read all the frame data (65536 pixels from every chip) to your buffer or save to the file. Acquisition can start by software (afther call a function, for example), or by HW trigger. Data types depends on chip technology and the operation mode.

Chip can generate one or two data blocks (event count and integrated times over threshold, for example) in one acquisition. Do not forget to set the operation mode. If mode not set, some devices measure something, but some other devices measure something else in this case.

Example projects

MiniPixTpx3-Frames – mode set, single frame, multiple frames with/without callback, continuous measuring

3.2.1 pxcMeasureSingleFrame

Summary

Performs a measurement of single frame and returns its data

Definition

PXCAPI int **pxcMeasureSingleFrame**(unsigned deviceIndex, double frameTime,
unsigned short* frameData, unsigned* size,
unsigned trgstg)

Parameters

deviceIndex - index of the device, starting from zero
frameTime - time of the measurment in seconds
frameData - pointer to buffer where data will be saved. For single detector size is 65536
size - pointer to varible with the size of the buffer. The actual size will be output to this variable
trgStg – settings of external trigger - one of the PXC_TRG_XXX values. Default PXC_TRG_NO.

Remark

This function have only one framebuffer. This is fully sufficient for simple modes. In combined modes (ToA+ToT or Event+IToT) only first data are available (ToA or Event). To take both outputs from the combined modes, it is necessary to use specialized functions for the given type of detector.

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.2.2 pxcMeasureSingleFrameMpx3

Summary

Performs a measurement of single frame and returns its data. This is only for Medipix3 chips

Definition

PXCAPI int **pxcMeasureSingleFrameMpx3**(unsigned deviceIndex, double frameTime,
unsigned* frameData1, unsigned* frameData2,
unsigned* size, unsigned trgstg)

Parameters

deviceIndex - index of the device, starting from zero
frameTime - time of the measurment in seconds
frameData1 - pointer to buffer where data from first counter will be saved. For single detector size is 65536
frameData2 - pointer to buffer where data from second counter will be saved. For single detector size is 65536
size - pointer to varible with the size of the buffer. The actual size will be output to this variable
trgStg – settings of external trigger - one of the PXC_TRG_XXX values. Default PXC_TRG_NO.

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Note

Only for the Medipix3 devices, not usable on Timepix or other Medipixes.

3.2.3 pxcMeasureSingleFrameTpx3

Summary

Performs a measurement of single frame and returns its data. This is only for Timepix3 detector.

Definition

PXCAPI int **pxcMeasureSingleFrame**(unsigned deviceIndex, double frameTime,
double* frameToaITot, unsigned short* frameTotEvent,
unsigned* size, unsigned trgstg)

Parameters

deviceIndex - index of the device, starting from zero
frameTime - time of the measurment in seconds
frameToaITot - pointer to buffer where data from ToA or iToT counter (based on set operation mode) will be saved. For single detector size is 65536
frameTotEvent - pointer to buffer where data from ToT or Event counter (based on set operation mode) will be saved. For single detector size is 65536
size - pointer to varible with the size of the buffer. The actual size will be output to this variable
trgStg – settings of external trigger - one of the PXC_TRG_XXX values. Default PXC_TRG_NO.

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Note

Only for the Timepix3 devices, not usable on other Timepixes or Medipixes.

Example

```
void pxcMeasureSingleFrameTpx3Test(unsigned di) { // di – device index ========
    int rc;                                 // return codes
    const unsigned  cSize = 65536;          // chip pixels count
    unsigned short  frameTotEvent[cSize]; // frame data - event count
    double          frameToaITot[cSize];  // frame data - integrated time over threshold
    double          time=1.0;             // frame acquisition time
    unsigned        size = cSize;         // buffer size and measured data size

    int mode = PXC_TPX3_OPM_EVENT_ITOT;
    rc = pxcSetTimepix3Mode(deviceIndex, mode);
    printErrors("pxcSetTimepix3Mode", rc, ENTER_ON);

    rc = pxcMeasureSingleFrameTpx3(di, time, frameToaITot, frameTotEvent, &size, PXC_TRG_NO);
    printErrors("pxcMeasureSingleFrameTpx3", rc, ENTER_ON);
    showFrameDual(frameTotEvent, frameToaITot, mode);
}
```

3.2.4 pxcMeasureSingleFrameTpx2

Summary

Performs a measurement of single frame and returns its data. This is only for Timepix2 detector and only if **calibration is disabled**.

Definition

PXCAPI int **pxcMeasureSingleFrameTpx2**
(unsigned deviceIndex, double frameTime, unsigned* frameData1, unsigned* frameData2, unsigned* size, unsigned trgStg)

Parameters

deviceIndex - index of the device, starting from zero
frameTime - time of the measurment in seconds
frameData1- pointer to buffer where data from the first counter (based on set operation mode) will be saved. For single detector size is 65536
frameData2- pointer to buffer where data from the second counter (based on set operation mode) will be saved. For single detector size is 65536
size - pointer to varible with the size of the buffer. The actual size will be output to this variable
trgStg – settings of external trigger - one of the PXC_TRG_XXX values. Default PXC_TRG_NO.

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Note

Only for the Timepix3 devices, not usable on other Timepixes or Medipixes.

Example

```
void pxcMeasureSingleFrameTpx2Test(unsigned di) { // di – device index ========
    int rc;                                 // return codes
    const unsigned  cSize = 65536;          // chip pixels count
    unsigned        frameData1[cSize];      // frame data – ToT data raw
    unsigned        frameData2[cSize];      // frame data – ToA data
    double          time=1.0;               // frame acquisition time
    unsigned        size = cSize;           // buffer size and measured data size

    int mode = PXC_TPX2_OPM_TOT10_TOA18;
    rc = pxcSetTimepix2Mode(deviceIndex, mode);
    printErrors("pxcSetTimepix2Mode", rc, ENTER_ON);

    rc = pxcMeasureSingleFrameTpx2(di, time, frameData1, frameData2, &size, PXC_TRG_NO);
    printErrors("pxcMeasureSingleFrameTpx2", rc, ENTER_ON);
    showFrameDual(frameData1, frameData2, mode);
}
```

3.2.5 pxcMeasureSingleCalibratedFrameTpx2

Summary

Performs a measurement of single frame and returns its data. This is only for Timepix2 detector and only if **calibration is enabled**.

Definition

PXCAPI int **pxcMeasureSingleCalibratedFrameTpx2**
(unsigned deviceIndex, double frameTime, double* frameData1, unsigned* frameData2, unsigned* size, unsigned trgStg)

Parameters

deviceIndex - index of the device, starting from zero
frameTime - time of the measurment in seconds
frameData1- pointer to buffer where calibrated data from the counter (based on set operation mode) will be saved. For single detector size is 65536
frameData2- pointer to buffer where data from the other counter (based on set operation mode) will be saved. For single detector size is 65536
size - pointer to varible with the size of the buffer. The actual size will be output to this variable
trgStg – settings of external trigger - one of the PXC_TRG_XXX values. Default PXC_TRG_NO.

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Note

Only for the Timepix3 devices, not usable on other Timepixes or Medipixes.

Example

```
void pxcMeasureSingleFrameTpx2Test(unsigned di) { // di – device index ========
    int rc;                                 // return codes
    const unsigned  cSize = 65536;          // chip pixels count
    double          frameData1[cSize];      // frame data – ToT data = calibrated to energy
    unsigned        frameData2[cSize];      // frame data – ToA data
    double          time=1.0;               // frame acquisition time
    unsigned        size = cSize;           // buffer size and measured data size

    int mode = PXC_TPX2_OPM_TOT10_TOA18;
    rc = pxcSetTimepix2Mode(deviceIndex, mode);
    printErrors("pxcSetTimepix2Mode", rc, ENTER_ON);

    rc = pxcMeasureSingleCalibratedFrameTpx2
                                        (di, time, frameData1, frameData2, &size, PXC_TRG_NO);
    printErrors("pxcMeasureSingleFrameTpx2", rc, ENTER_ON);
    showFrameDual(frameData1, frameData2, mode);
}
```

3.2.6 pxcMeasureMultipleFrames

Summary

Performs a measurement of several frames to memory

Definition

PXCAPI int **pxcMeasureMultipleFrames**(unsigned deviceIndex, unsigned fameCount, double frameTime, unsigned trgStg)

Parameters

deviceIndex - index of the device, starting from zero
frameCount - numeber of frames to measure
frameTime - time of the measurment in seconds
trgStg – settings of external trigger - one of the PXC_TRG_XXX values. Default PXC_TRG_NO.

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Example

```
void pxcMeasureMultipleFramesTpx3Test() { // =========================
    int rc;                               // return codes
    const unsigned  cSize = 65536;        // chip pixels count
    unsigned short  frameTotEvent[cSize]; // frame data - event count
    double          frameToaITot[cSize];  // frame data - integrated time over threshold
    double          frameTime = 1.0;      // frame acquisition time
    unsigned        frameCount = 5;       // frame count
    unsigned        size = cSize;         // buffer size and measured data size

    int mode = PXC_TPX3_OPM_EVENT_ITOT;
    rc = pxcSetTimepix3Mode(0, mode);
    printErrors("pxcSetTimepix3Mode", rc, ENTER_ON);

    printf("measuring %d frames...\n", frameCount);

    pxcMeasureMultipleFrames(0, frameCount, frameTime, PXC_TRG_NO);
    printErrors("pxcMeasureMultipleFrames", rc, ENTER_ON);

    for (unsigned n=0; n<frameCount; n++) {
        rc = pxcGetMeasuredFrameTpx3(0, n, frameToaITot, frameTotEvent, &size);
        printErrors("pxcGetMeasuredFrameTpx3", rc, ENTER_ON);
        size = cSize;
        showFrameDual(frameTotEvent, frameToaITot, mode);
    }
}
```

3.2.7 pxcMeasureMultipleFramesWithCallback

Summary

Performs a measurement of several frames to memory. When each frame is measured, the supplied callback function is called and the userData parameter is passed as argument.

Definition

PXCAPI int **pxcMeasureMultipleFramesWithCallback**(unsigned deviceIndex, unsigned fameCount, double frameTime, unsigned trgStg, FrameMeasuredCallback callback, intptr_t userData)

Parameters

deviceIndex - index of the device, starting from zero
frameCount - numeber of frames to measure
frameTime - time of the measurment in seconds
trgStg – settings of external trigger - one of the PXC_TRG_XXX values. Default PXC_TRG_NO.
callback – pointer to function of FrameMeasuredCallback type
userData – pointer to some user object/memory that is passed in callback function

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Example

```
void pxcMeasureMultipleFramesWithCallbackTestTpx3(unsigned deviceIndex) { // ==============
    int rc; // return codes
    double frameTime = 1;
    unsigned frameCount = 5;
    tMmfClbData usrData;

    usrData.di = deviceIndex;
    usrData.opm = PXC_TPX3_OPM_EVENT_ITOT;
    rc = pxcSetTimepix3Mode(deviceIndex, usrData.opm);
    printErrors("pxcSetTimepix3Mode", rc, ENTER_ON);

    rc = pxcMeasureMultipleFramesWithCallback
      (deviceIndex, frameCount, frameTime, PXC_TRG_NO, mmfCallback, (intptr_t)&usrData);
    printErrors("pxcMeasureMultipleFramesWithCallback", rc, ENTER_ON);
    // Measure function ends, after all callbacks are serviced
}
```

```
void mmfCallback(intptr_t acqCount, intptr_t userData) { // ================================
    int rc;                                 // return codes
    const unsigned  cSize = 65536;          // chip pixels count
    unsigned short  frameTotEvent[cSize]; // frame data - event count
    double          frameToaITot[cSize];  // frame data - integrated time over threshold
    unsigned        size = cSize;           // buffer size and measured data size
    tMmfClbData usrData = *((tMmfClbData*)userData); // data transferred from start function

    printf("mmfCallback acqCount=%d, di=%d\n", (unsigned)acqCount, usrData.di);

    rc = pxcGetMeasuredFrameTpx3
             (usrData.di, (unsigned)acqCount-1, frameToaITot, frameTotEvent, &size);
    printErrors("pxcGetMeasuredFrameTpx3", rc, ENTER_OFF); printf(", size=%d\n", size);

    if (rc==0) {
        printf("Measured frame index %lu, ", (unsigned)acqCount - 1);
        printf("count %d\n", pxcGetMeasuredFrameCount(usrData.di));

        showFrameDual(frameTotEvent, frameToaITot, usrData.opm);
    }
}
// The acqCount contains count of measured frames that are currently waiting in memory.
// You can use acqCount-1 as frame index to read.
// The userData is 64b number used to transfer some data from starting function to callback function.
// You can use it as data pointer like in this example, or simply as number with some overtyping.

typedef struct { // structure for userData, that is using in mmfCallback
    unsigned    di;     // device index
    int         opm;    // operation mode
} tMmfClbData;
```

3.2.8 pxcMeasureContinuous

Summary

Performs an "endless" measurement of several frames to memory. The measurement is run until it's aborted by pxcAbortMeasurement function. When each frame is measured, the supplied callback function is called and the userData parameter is passed as argument.

Definition

PXCAPI int **pxcMeasureContinuous**(unsigned deviceIndex, unsigned fameBufferSize,
double frameTime, unsigned trgStg,
FrameMeasuredCallback callback, intptr_t userData)

Parameters

deviceIndex - index of the device, starting from zero
frameTime - time of the measurment in seconds
frameBufferSize - numeber of frames in circular buffer
trgStg – settings of external trigger - one of the PXC_TRG_XXX values. Default PXC_TRG_NO.
callback – pointer to function of FrameMeasuredCallback type
userData – pointer to some user object/memory that is passed in callback function

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Remarks

- This function, differently to other multi-frame measurements, not waiting for process end. It simply start measurement and program continue next.
- You can set-up the callback by 2 ways:
  1. Simply use pointer to callback function in the pxcMeasureContinuous function. After do it, the program must not leave the function in which pxcMeasureContinuous was used until the measurement is canceled.
  2. First use the **pxcRegisterAcqEvent** to set-up the callback, second use pxcMeasureContinuous without callback pointer or with NULL. Now it is possible to leave the superior function and measuring continue working independent. Afther cancel of the measurement, may be needed use the **pxcUnregisterAcqEvent** before other use of the Acq event.

Example

```
void pxcMeasureContinuousTestTpx3(unsigned deviceIndex) { // ==============================
    int rc; // return codes
    double frameTime = 2;
    unsigned buffrerFrames = 3;
    tMmfClbData usrData;

    usrData.di = deviceIndex;
```

```
    usrData.cnt = 0; // num of frames to stop in callback (0 endless)
    usrData.opm = PXC_TPX3_OPM_EVENT_ITOT;
    rc = pxcSetTimepix3Mode(deviceIndex, usrData.opm);
    printErrors("pxcSetTimepix3Mode", rc, ENTER_ON);

    // method 1 (single step):
    //rc = pxcMeasureContinuous
       (deviceIndex, buffrerFrames, frameTime, PXC_TRG_NO, mcCallback, (intptr_t)&usrData);
    //printErrors("pxcMeasureContinuous", rc, ENTER_ON);

    // method 2 (2 steps):
    // register event
    pxcRegisterAcqEvent(0, PXC_ACQEVENT_ACQ_FINISHED, mcCallback, (intptr_t)&usrData);
    printErrors("pxcRegisterAcqEvent", rc, ENTER_ON);

    // start continuous measuring (method 2, step 2)
    rc = pxcMeasureContinuous(deviceIndex, buffrerFrames, frameTime);
    printErrors("pxcMeasureContinuous", rc, ENTER_ON);

    printf("waiting for callbacks...\n");
    getchar(); // waiting so that callbacks can come
    printf("pxcMeasureContinuousTest: user end\n");
}
```

The callback function is some as used in pxcMeasureMultipleFramesWithCallback example, but usrData structure contains extra member, the cnt. This can be used to stop, whitch is additional in the callback:

```
    static unsigned cnt=0;
    cnt++;
    if (usrData.cnt>0 && cnt>=usrData.cnt) {
        // stop continuous measuring on user defined number of frames
        printf("pxcAbortMeasurement...");
        rc = pxcAbortMeasurement(usrData.di);
        printErrors("pxcAbortMeasurement", rc, ENTER_ON);
        printf("Press enter to exit the program.\n");
    }
```

3.2.9 pxcAbortMeasurement

Summary

Stopts the currently running measurement.

Definition

PXCAPI int **pxcAbortMeasurement**(unsigned deviceIndex)

Parameters

deviceIndex - index of the device, starting from zero

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.2.10 pxcGetMeasuredFrameCount

Summary

Returns number of measured frames in memory

Definition

PXCAPI int **pxcGetMeasuredFrameCount**(unsigned deviceIndex)

Return Value

Number of measured frames, otherwise the return value is a PXCERR_XXX code

3.2.11 pxcSaveMeasuredFrame

Summary

Saves the measured frame to a file on the harddrive. This can be used instead of the **pxcGetMeasuredFrame**.

Definition

PXCAPI int **pxcSaveMeasuredFrame**(unsigned deviceIndex, unsigned frameIndex,
const char* filePath)

Parameters

deviceIndex - index of the device, starting from zero
frameIndex – index of the frame, starting from zero
filePath – path to the file where frame will be saved

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Example

```
void someCallback(intptr_t acqCount, intptr_t userData) { // ======================
    int rc; // return codes
    tMmfClbData usrData = *((tMmfClbData*)userData);

    rc = pxcSaveMeasuredFrame(usrData.di, (unsigned)acqCount - 1, "testFile.txt");
    printErrors("pxcSaveMeasuredFrame-txt", rc, ENTER_ON);

    rc = pxcSaveMeasuredFrame(usrData.di, (unsigned)acqCount - 1, "testFile.png");
    printErrors("pxcSaveMeasuredFrame-txt", rc, ENTER_ON);

/* supported file extensions (format automatically detected by extension in filename):
    #define PX_EXT_ASCII_FRAME          "txt"
    #define PX_EXT_BINARY_FRAME         "pbf"
    #define PX_EXT_MULTI_FRAME          "pmf"
    #define PX_EXT_BINARY_MULTI_FRAME   "bmf"
    #define PX_EXT_COMP_TPXSTREAM       "pcts"
    #define PX_EXT_TPX3_PIXELS          "t3p"
    #define PX_EXT_TPX3_PIXELS_ASCII    "t3pa"
    #define PX_EXT_TPX3_RAW_DATA        "t3r"
    #define PX_EXT_FRAME_DESC           "dsc"
    #define PX_EXT_INDEX                "idx"
    #define PX_EXT_CLUSTER_LOG          "clog"
    #define PX_EXT_PIXEL_LOG            "plog"
    #define PX_EXT_PNG                  "png"
    #define PX_EXT_PIXET_RAW_DATA       "prd"  */
}
```

3.2.12 pxcGetMeasuredFrame

Summary

Gets data of specified measured frame from memory

Definition

PXCAPI int **pxcGetMeasuredFrame**(unsigned deviceIndex, unsigned fameIndex,
unsigned short* frameData, unsigned* size)

Parameters

deviceIndex - index of the device, starting from zero
frameIndex – index of the frame, starting from zero
frameData - pointer to buffer where data will be saved. For single detector size is 65536
size - pointer to varible with the size of the buffer. The actual size will be output to this variable

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Remark

This function have only one framebuffer. This is fully sufficient for simple modes. In combined modes (ToA+ToT or Event+IToT) only first data are available (ToA or Event). To take both outputs from the combined modes, it is necessary to use specialized functions for the given type of detector.

3.2.13 pxcGetMeasuredFrameMpx3

Summary

Gets data of specified measured frame from memory. For Medipix3 chip

Definition

PXCAPI int **pxcGetMeasuredFrameMpx3**(unsigned deviceIndex, unsigned fameIndex, unsigned* frameData1, unsigned* frameData2, unsigned* size)

Parameters

deviceIndex - index of the device, starting from zero
frameIndex – index of the frame, starting from zero
frameData1 - pointer to buffer where data from first counter will be saved. For single detector size is 65536
frameData2 - pointer to buffer where data from second counter will be saved. For single detector size is 65536
size - pointer to varible with the size of the buffer. The actual size will be output to this variable

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Note

Only for the Medipix3 devices, not usable on Timepix or other Medipixes.

3.2.14 pxcGetMeasuredFrameTpx2

Summary

Gets data of specified measured frame from memory.
For Timepix2 chip only and only if **calibration disabled**.

Definition

PXCAPI int **pxcGetMeasuredFrameTpx2**(unsigned deviceIndex, unsigned frameIndex, unsigned* frameData1, unsigned* frameData2, unsigned* size)

Parameters

deviceIndex - index of the device, starting from zero
frameIndex – index of the frame, starting from zero
frameData1 – pointer to buffer where data from the first counter (based on set operation mode) will be saved. For single detector size is 65536
frameData2 – pointer to buffer where data from the second counter (based on set operation mode) will be saved. For single detector size is 65536size - pointer to varible with the size of the buffer. The actual size will be output to this variable

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Note

Only for the Timepix2 devices, not usable on other Timepixes or Medipixes.

3.2.15 pxcGetMeasuredCalibratedFrameTpx2

Summary

Gets data of specified measured frame from memory.
For Timepix2 chip only and only if **calibration enabled**.

Definition

PXCAPI int **pxcGetMeasuredCalibratedFrameTpx2**(unsigned deviceIndex, unsigned frameIndex, double* frameData1, unsigned* frameData2, unsigned* size)

Parameters

deviceIndex - index of the device, starting from zero
frameIndex – index of the frame, starting from zero
frameData1 – pointer to buffer where calibrated data from the counter (based on set operation mode) will be saved. For single detector size is 65536
frameData2 – pointer to buffer where data from the other counter (based on set operation mode) will be saved. For single detector size is 65536

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Note

Only for the Timepix2 devices, not usable on other Timepixes or Medipixes.

3.2.16 pxcGetMeasuredFrameTpx3

Summary

Gets data of specified measured frame from memory. For Timepix3 chip

Definition

PXCAPI int **pxcGetMeasuredFrameTpx3**(unsigned deviceIndex, unsigned fameIndex,
double* frameToaITot,
unsigned short* frameToTEvent, unsigned* size)

Parameters

deviceIndex - index of the device, starting from zero
frameIndex – index of the frame, starting from zero
frameToaITot - pointer to buffer where data from ToA or iToT counter (based on set operation mode) will be saved. For single detector size is 65536
frameTotEvent - pointer to buffer where data from ToT or Event counter (based on set operation mode) will be saved. For single detector size is 65536
size - pointer to varible with the size of the buffer. The actual size will be output to this variable

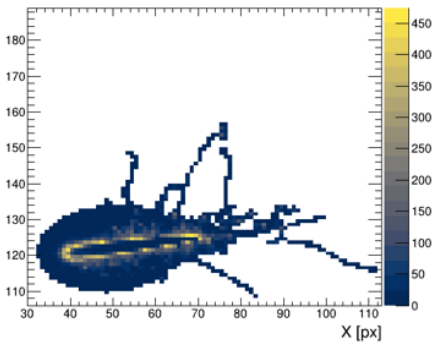
Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Note

Only for the Timepix3 devices, not usable on other Timepixes or Medipixes.

3.3 Data driven measuring



The functions described in this chapter can be used if frame-based measurement is not good choice, in particle collision research, for example. Afther measurement start, data from every event is stored in the buffer. Pixels have structure (defined in the **pxcapi.h**):

```
typedef struct _Tpx3Pixel {
    double toa;            // time of arrival     [1 ns]
    float tot;             // time over threshold [25 ns]
    unsigned int index;    // index of pixel position
} Tpx3Pixel;
```

The Pixet core has a circular buffer in memory. The default size is 100 MB. There are blocks in the buffer. The default block size is 66000 bytes. Incoming data is stored in a block. The "new data" event (that can start the callback or save the blockt to file) occurs if:

- The data size reaches the block size
- Some data is in block over the timeout (500 ms)
- The measurement ends

if you need to process data more often, you can set smaller block size. But if blocks are too small, data transfer speed can fall and some data can be lost. This can occurs if block size is less than approximately 5000 bytes (depends on computer speed and other circumstances).

Example projects

MiniPixTpx3-DataDriven – Set the parameters, use the callback and display the data, some auxiliary things and data convert. Save data to files.

Note

Functions in this chapter **is only for the Timepix3 devices**, not usable on other Timepixes or Medipixes.

3.3.1 pxcMeasureTpx3DataDrivenMode

Summary

Performs a measurement with Timepix3 detector in Data Driven Mode (event by event mode, when stream of pixels is sent). **Timepix3 only.**

Definition

PXCAPI int **pxcMeasureTpx3DataDrivenMode**(unsigned deviceIndex, unsigned measTime, const char* filename, unsigned trgStg, AcqEventFunc callback, intptr_t userData)

Parameters

deviceIndex - index of the device, starting from zero
measTImeTime – the total time of the measurement in seconds
filename – output file name and path (extensions must end *.t3pa, *.trp, *.t3r)
trgStg – settings of external trigger - one of the PXC_TRG_XXX values. Default PXC_TRG_NO.
callback – pointer to function of acqEventFunc type
userData – pointer to some user object/memory that is passed in callback function

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

Example1 – online data processing:

```
void timepix3DataDrivenGetPixelsTest(unsigned deviceIndex) { // ============================
    int rc; // return codes
    double measTime = 30;
    int devIdx = deviceIndex; // transmitted over pointer for use in the callback function

    // working with TOA, TOATOT, TOT_NOTOA, not working with EVENT_ITOT
    rc = pxcSetTimepix3Mode(deviceIndex, PXC_TPX3_OPM_TOA);
    printErrors("pxcSetTimepix3Mode", rc, ENTER_ON);

    rc = pxcSetDeviceParameter(deviceIndex, PAR_DD_BLOCK_SIZE, 6000); // block [B]
    printErrors("pxcSetDeviceParameter", rc, ENTER_OFF);
    rc = pxcSetDeviceParameter(deviceIndex, PAR_DD_BUFF_SIZE, 100);   // buffer [MB]
    printf(", %d", rc);
    rc = pxcSetDeviceParameter(deviceIndex, PAR_DCC_LEVEL, 80);
    printErrors(",", rc, ENTER_ON); // data consistency check level (50-150) (*)

    rc = pxcMeasureTpx3DataDrivenMode
             (deviceIndex, measTime, "", PXC_TRG_NO, onTpx3Data, (intptr_t)&devIdx);
    printErrors("pxcMeasureTpx3DataDrivenMode", rc, ENTER_ON);
}
```

* Data consistency check level: Detection level of ToA data inconsistency. If the chip is overloaded due too many particles in time, clock can freeze and a ToA inconsistency occurrs. If it is detedted, measuring immediately stop and error "Acquisition failed (Data flow corrupted !!)" occurrs. Level 50 is some like OFF, level 150 is highest sensitivity.

```
void onTpx3Data(intptr_t eventData, intptr_t userData) { // ================================
    int deviceIndex = *((unsigned*)userData);
    unsigned pixelCount = 0;
    int rc; // return codes

    rc = pxcGetMeasuredTpx3PixelsCount(deviceIndex, &pixelCount);
    printErrors("getMeasuredTpx3PixelsCount", rc, ENTER_OFF);
    printf(" PixelCount: %u\n", pixelCount);

    auto result = new Tpx3Pixel[pixelCount];
    rc = pxcGetMeasuredTpx3Pixels(deviceIndex, result, pixelCount);
    printErrors("pxcGetMeasuredTpx3Pixels", rc, ENTER_ON);

    dataUsingFunction(&result, pixelCount);

    delete[] result;
}
```

Example2 – Saving to file:

```
void timepix3DataDrivenToFileTest() { // ================================================
    int rc; // return codes

    // working with TOA, TOATOT, TOT_NOTOA, not working with EVENT_ITOT
    rc = pxcSetTimepix3Mode(0, PXC_TPX3_OPM_TOA);
    printErrors("pxcSetTimepix3Mode", rc);

    rc = pxcSetDeviceParameter(0, PAR_DD_BLOCK_SIZE, 6000); // in B
    printf("pxcSetDeviceParameter %d", rc);
    rc = pxcSetDeviceParameter(0, PAR_DD_BUFF_SIZE, 500); // in MB
    printf(", %d", rc);
    rc = pxcSetDeviceParameter(0, PAR_TRG_STG, 3);
    // (0=logical 0, 1 = logical 1, 2 = rising edge, 3 = falling edge)
    printErrors(",", rc);

    //rc = pxcMeasureTpx3DataDrivenMode(deviceIndex, 5, "test.t3r", PXC_TRG_NO, 0, 0);
    //rc = pxcMeasureTpx3DataDrivenMode(deviceIndex, 5, "test.t3pa", PXC_TRG_HWSTART, 0, 0);
    rc = pxcMeasureTpx3DataDrivenMode(deviceIndex, 5, "test.t3pa", PXC_TRG_NO, 0, 0);
    printErrors("pxcMeasureTpx3DataDrivenMode", rc);
}
```

3.3.2 pxcGetMeasuredTpx3Pixels

Summary

Gets the measured Timepix3 pixels data. **Timepix3 only.**

Definition

PXCAPI int **pxcGetMeasuredTpx3Pixels** (unsigned deviceIndex, Tpx3Pixel* pixels, unsigned pixelCount)

Parameters

deviceIndex - index of the device, starting from zero
pixels – pointer to array of Tpx3Pixels that will be filled with measured pixels
pixelCount – size of the supplied array as number of pixels

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.3.3 pxcGetMeasuredTpx3PixelsCount

Summary

Gets the number of measured Timepix3 pixels in data driven mode. **Timepix3 only.**

Definition

PXCAPI int **pxcGetMeasuredTpx3PixelsCount**(unsigned deviceIndex, unsigned* pixelCount)

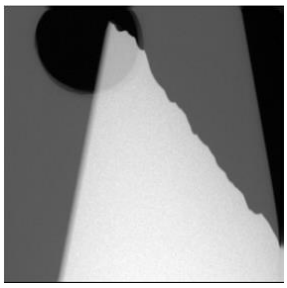
Parameters

deviceIndex - index of the device, starting from zero
pixelCount – pointer to unsigned variable where number of pixel count is set

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.4 Data processing: The beam hardening and bad pixels



The functions described in this chapter can be used to compensate the beam hardening effect, bad pixels and chip sensitivity uneven to get the data with true thickness of measured material.

First step is adding set of the reference images. Dirrect beam (or beam with filter that will be used in all the process) and flat plates with different thicknesses, optimally in whole range that you want measuring. Use the **pxcAddBHMask**. Reference images contains unsigned 32bit pixels. To create the reference pictures, recommended fold more acquired images to achieve pixel values aproximatelly tens of thousands or more (50,000 is good start). Don't forget corectly set their thickness.

If 2 or more reference images have been added, it is possible to determine which pixels are not usable. Check this using the **pxcGetBHBadPixelMatrix** function. If there are too many of these pixels, the **pxcApplyBHCorrection** function may lose accuracy and the **pxcInterpolateBadPixels** function will not work. In this case you can check:

- If the set-up is correct (condition of the reference plates; Is the whole area of the chip good irradiated and covered by the reference plates? Is there an obstacle in the beam?)
- If there is too much contrast between the references - a common problem, especially between the free beam and the first reference plate - you can use a filter throughout the process
- If some areas have too low values with high noise - increase the number of integrated frames, beam intensity, or acquisition time
- If exist areas with overexposition – decrease the time or beam intensity
- If the instability of the CdTe chip has manifested itself - increase the time by integrating the reference images
- If using CdTe and measuring start too soon afther chip init
- Try use other beam energy

After add sufficient number of reference images, you can measure image of the sample and use the **pxcApplyBHCorrection** function. Output is array of double, in units from reference thicknesses.

Now you can use the **pxcGetDeviceAndBHBadPixelMatrix** to get array of bad and other unusable pixels. Use this data with the **pxcInterpolateBadPixels**. If it doesn't work, you can use only device bad pixels from the **pxcGetDeviceBadPixelMatrix** function.

Example projects

MiniPixTpx3-Thickness-auto – Acquision time auto tuning for easy first experiments. Interactive control in console, text histograms and preview of images. But the code is relatively complex.

MiniPixTpx3-Thickness-bath – Bath process, using text config file. Simple code, but you need to set usable acquisition times.

3.4.1 pxcAddBHMask

Summary

Adds a new mask (frame) for Beam-Hardening calibration

Definition

PXCAPI int **pxcAddBHMask**(unsigned* data, unsigned size, double frameTime, double thickness)

Parameters

data – data of the frame tat will be used as BH mask
size – size of the data - number of pixels (width * height)
frameTime – acquisition time of the frame in seconds
thickness – thickness of the measured data

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.4.2 pxcBHMaskCount

Summary

Returns number of inserted Beam-Hardening masks (frames)

Definition

PXCAPI int **pxcBHMaskCount**()

Return Value

Number of masks if successful, otherwise the return value is a PXCERR_XXX code

3.4.3 pxcRemoveBHMask

Summary

Removes Beam-Hardening mask (frame)

Definition

PXCAPI int **pxcRemoveBHMask**(int index)

Parameters

index – index of the mask to remove

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.4.4 pxcApplyBHCorrection

Summary

Applies the Beam-Hardening correction to supplied frame

Definition

PXCAPI int **pxcApplyBHCorrection**(unsigned* inData, unsigned size, double frameTime, double* outData)

Parameters

inData – data of the frame that will be corrected
size – size of the data - number of pixels (width * height)
frameTime – acquisition time of the measured frame in seconds
outData – output data buffer where corrected data will be saved

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.4.5 pxcGetDeviceBadPixelMatrix

Summary

Gets the devices's matrix of bad pixels (bad = 1, good = 0). Bad pixels are pixels that are masked.

Definition

PXCAPI int **pxcGetDeviceBadPixelMatrix**(unsigned devIndex, unsigned* badPixelMatrix, unsigned size)

Parameters

deviceIndex - index of the device (indexing starting from 0)
badPixelMatrix – output data buffer where bad pixel matrix will be saved
size – size of the data - number of pixels (width * height)

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.4.6 pxcGetBHBadPixelMatrix

Summary

Gets the bad pixel matrix from Beam Hardening correction - pixels that cannot be corrected. (bad pixel = 1, good = 0). Must be called after all BH masks are added.

Definition

PXCAPI int **pxcGetBHBadPixelMatrix**(unsigned* badPixelMatrix, unsigned size)

Parameters

badPixelMatrix – output data buffer where bad pixel matrix will be saved
size – size of the data - number of pixels (width * height)

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.4.7 pxcGetDeviceAndBHBadPixelMatrix

Summary

Gets the combined bad pixel matrix from the device and BeamHardening (badpixel = 1, good = 0).

Definition

PXCAPI int **pxcGetDeviceAndBHBadPixelMatrix**(unsigned devIndex, unsigned* badPixelMatrix, unsigned size)

Parameters

deviceIndex - index of the device (indexing starting from 0)
badPixelMatrix – output data buffer where bad pixel matrix will be saved
size – size of the data - number of pixels (width * height)

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.4.8 pxcInterpolateBadPixels

Summary

Interpolates bad pixels in the image. Uses badPixelsMatrix as bad pixels (badPixel = 1, good = 0)

Definition

PXCAPI int **pxcInterpolateBadPixels** (unsigned char* badPixelsMatrix, double* data, unsigned width, unsigned height)

Parameters

badPixelMatrix – matrix of the bad pixels
data – data to be interpolated

width – width of the image
height – height of the image

Return Value

0 if successful, otherwise the return value is a PXCERR_XXX code

3.4.9 Example

```
int measureThicknessBH() { // ============================================================
    int rc;                         // return codes
    unsigned frameTmp[CHIP_pixels]; // measured data
    double frameOut[CHIP_pixels];   // BH corrected output data
    unsigned char badPixels[CHIP_pixels]; // combined bad and uncompensable pixels
    double frameTime;               // frame acquisition time
    double refThick;                // thickness of ref. plate
    char fileSet;                   // files setting: d data, p picture, b both, n none
    char bpSet;                     // bad px setting: c chip, b BH system
    string name;                    // line name/comment and used in output filenames
    unsigned size = CHIP_pixels; // buffer/chip pixel count, pxcMeasureSingleFrameTpx3
    string line, valStr;            // for config lines decoding
    fstream cfgFile("config.txt", ios::in);
    unsigned n, lineCnt;
    char fileOutName[20], fileOutName2[20], fileOutName3[20]; // output filenames

    rc = pxcSetTimepix3Mode(0, PXC_TPX3_OPM_EVENT_ITOT); // set mode on device 0
    if (errorCheck("pxcSetTimepix3Mode", rc, ABORT_ask)) return -1;
    lineCnt=1;
    while (getline(cfgFile, line, '\n')) {
        cout << "L " << lineCnt++ << " > " << line << endl;
        stringstream linestream(line);
        getline(linestream, valStr, ',');
        switch (valStr.at(0)) { // ---------------------------------------------------------
            case 'r': // reference
            case 'm': // measuring
                cout << "- Prepare this step and pres Y to do this, or press N to No: ";
                if (choiceKey('y', "Yes", 'n', "No")=='n') continue;
        }

        switch (valStr.at(0)) { // ---------------------------------------------------------
            case 'r': // reference ------------------------------------------------
                getline(linestream, valStr, ','); // ref thick
                refThick = stof(valStr);
```

```
        getline(linestream, valStr, ','); // acq time
        frameTime = stof(valStr);
        getline(linestream, valStr, ','); // num of expositions
        n = stoi(valStr);
        multiAcq(frameTime, n, frameTmp); // measure
        getline(linestream, valStr, ','); // file saving settings
        fileSet = trim(valStr).at(0);
        //cout << "fs >" << valStr << "<" << endl;
        getline(linestream, name, ',');   // name of the step and files
        name = trim(name);
        if (fileSet!='n') {
            cout << "save " << name << " set=" << fileSet << endl;
            strcpy_s(fileOutName, 20, name.c_str());
        }
        if (fileSet=='d') {         // data file
            saveToTXT(frameTmp, fileOutName, frameTime*(double)n, 1023*n);
        } else if (fileSet=='p') {  // picture file
            saveToBMP(frameTmp, fileOutName, frameTime*(double)n, 1023*n);
        } else if (fileSet=='b') {  // both files
            saveToBMP(frameTmp, fileOutName, frameTime*(double)n, 1023*n);
            saveToTXT(frameTmp, fileOutName, frameTime*(double)n, 1023*n);
        }
        rc = pxcAddBHMask(frameTmp, CHIP_pixels, frameTime*(double)n, refThick);
        if (errorCheck("pxcAddBHMask", rc, ABORT_ask)) return -1;
        break;
    case 'm': // measuring ------------------------------------------------
        getline(linestream, valStr, ','); // acq time
        frameTime = stof(valStr);
        getline(linestream, valStr, ','); // num of expositions
        n = stoi(valStr);
        multiAcq(frameTime, n, frameTmp); // measure
        getline(linestream, valStr, ','); // bad px settings
        bpSet = trim(valStr).at(0);
        getline(linestream, valStr, ','); // file saving settings
        fileSet = trim(valStr).at(0);
        getline(linestream, name, ',');   // name of the step and files
        name = trim(name);
        if (fileSet!='n') {
            cout << "save " << name << " set=" << fileSet << endl;
            strcpy_s(fileOutName,  20, (name + "-raw").c_str());
            strcpy_s(fileOutName2, 20, (name + "-bhc").c_str());
            strcpy_s(fileOutName3, 20, (name + "-bpc").c_str());
        }
        if (fileSet=='d') {         // data file raw
```

```
                saveToTXT(frameTmp, fileOutName, frameTime*(double)n, 1023*n);
            } else if (fileSet=='p') {  // picture file raw
                saveToBMP(frameTmp, fileOutName, frameTime*(double)n, 1023*n);
            } else if (fileSet=='b') {  // both files raw
                saveToBMP(frameTmp, fileOutName, frameTime*(double)n, 1023*n);
                saveToTXT(frameTmp, fileOutName, frameTime*(double)n, 1023*n);
            }

            rc = pxcApplyBHCorrection
                      (frameTmp, CHIP_pixels, frameTime*(double)n, frameOut);
            if (errorCheck("pxcApplyBHCorrection", rc, ABORT_ask)) return -1;

            if (fileSet=='d') {         // data file BH corrected
                saveToTXT(frameOut, fileOutName2, frameTime*(double)n, 1023*n);
            } else if (fileSet=='p') {  // picture file BH corrected
                saveToBMP(frameOut, fileOutName2, frameTime*(double)n, 1023*n);
            } else if (fileSet=='b') {  // both files BH corrected
                saveToBMP(frameOut, fileOutName2, frameTime*(double)n, 1023*n);
                saveToTXT(frameOut, fileOutName2, frameTime*(double)n, 1023*n);
            }

            if (bpSet=='d') { // bad px setting: d - device, b - device + BH system
                rc = pxcGetDeviceBadPixelMatrix(0, badPixels, CHIP_pixels);
                if (errorCheck("pxcGetDeviceBadPixelMatrix", rc, ABORT_ask)) return -1;
            } else {
                rc = pxcGetDeviceAndBHBadPixelMatrix(0, badPixels, CHIP_pixels);
                if (errorCheck("pxcGetDeviceAndBHBadPixelMatr", rc, ABORT_ask)) return -1;
            }

            rc = pxcInterpolateBadPixels(badPixels, frameOut, 256, 256);
            if (errorCheck("pxcInterpolateBadPixels", rc, ABORT_ask)) return -1;

            if (fileSet=='d') {         // data file BH+BP corrected
                saveToTXT(frameOut, fileOutName3, frameTime*(double)n, 1023*n);
            } else if (fileSet=='p') {  // picture file BH+BP corrected
                saveToBMP(frameOut, fileOutName3, frameTime*(double)n, 1023*n);
            } else if (fileSet=='b') {  // both files BH+BP corrected
                saveToBMP(frameOut, fileOutName3, frameTime*(double)n, 1023*n);
                saveToTXT(frameOut, fileOutName3, frameTime*(double)n, 1023*n);
            }
            break;
        default:
    }
}
```

```
    return 0;
}
int multiAcq(double frameTime, unsigned frameCount, unsigned *data) { // =================
    static double tmp[CHIP_pixels];
    static unsigned short frameTotEvent[CHIP_pixels];
    int rc;          // return codes
    unsigned fn, pn; // frame number, pixel number
    unsigned size = CHIP_pixels; //  pixel count, input/output pxcMeasureSingleFrameTpx3

    cout << "multiAcq time=" << frameTime << " cnt=" << frameCount << ": ";
    for (pn=0; pn<CHIP_pixels; pn++) data[pn]=0;

    for (fn=0; fn<frameCount; fn++) {
        cout << ".";
        rc = pxcMeasureSingleFrameTpx3(0, frameTime, tmp, frameTotEvent, &size, PXC_TRG_NO);
        if (rc!=0) return rc;
        for (pn=0; pn<CHIP_pixels; pn++) data[pn] += (unsigned)frameTotEvent[pn];
    }
    cout << endl;
    showHistogram(data);
    return 0;
}
```

This example using the config file named **config.txt** to configure steps of the process. File contents is like this:

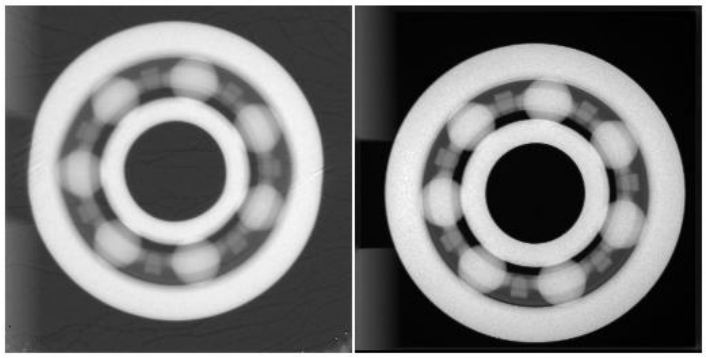
```
o, First: line type: o - comment, r - referrence plate, m - measuring
o, Ref. cfg. line: r, thickness [mm], acqTime [s], num of expositions, output files: d data / p picture / b both / n none, name
r, 0, 0.005, 200, d, dir_beam
r, 1, 0.006, 200, d, ref_1
r, 2, 0.010, 200, d, ref_2
r, 3, 0.015, 200, d, ref_3
o, Measuring cfg. line: m, acqTime [s], num of expositions, bad px setting: d device / b BH system, output files: d data / p
picture / b both / n none, name
m, 0.005, 200, d, b, test1
m, 0.005, 200, d, b, test2
m, 0.01, 200, b, b, test3
m, 0.02, 200, b, b, test4
```

Example program read the line, display it in the console, will ask if it should do it and wait for user response. If yes, program acquire required number of frames, view simple histogram in the console, do the processing and save images, if reqired.

This example is using many auxiliary functions. Complete code is in the example project named **MiniPixTpx3-Thickness-bath**. To get usable acquisition times do a experimental exposures in the **Pixet**

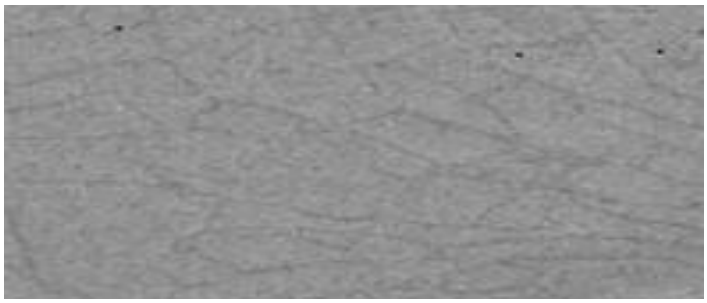
program, or get them using a second sample project named **MiniPixTpx3-Thickness-auto** that has auto-tuning and image preview in the console.

3.4.10 Some images



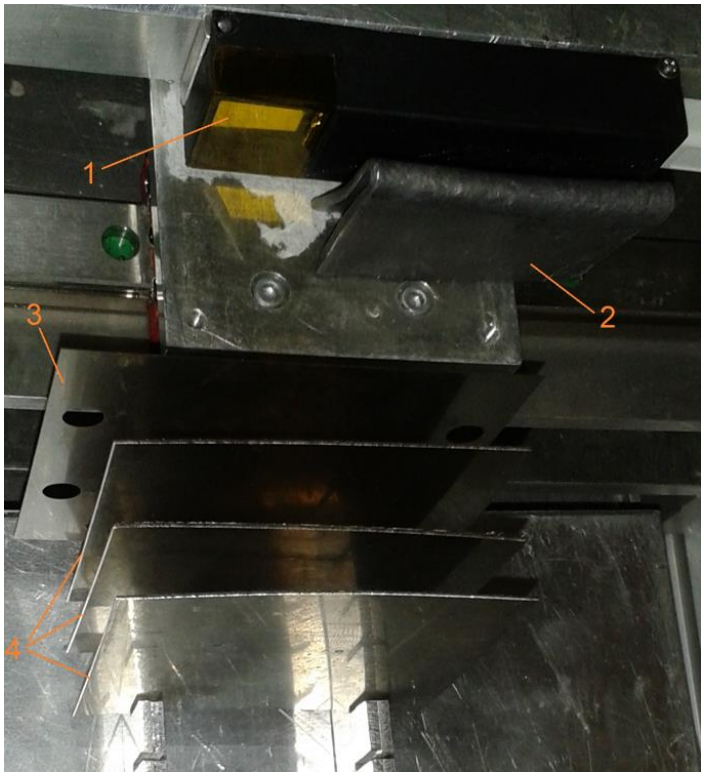
1. CdTe 2 mm (left) vs Si 0,3 mm (right)

CdTe advantages: More sensitive, orders of magnitude shorter acquisition times or lower beam intensities. Have lower energy consumption and and it less warming.
imgs:
CdTe: integrated 3.43 sec (25,6 ms subframes)
Si: integrated 27.7 sec (625 ms subframes)



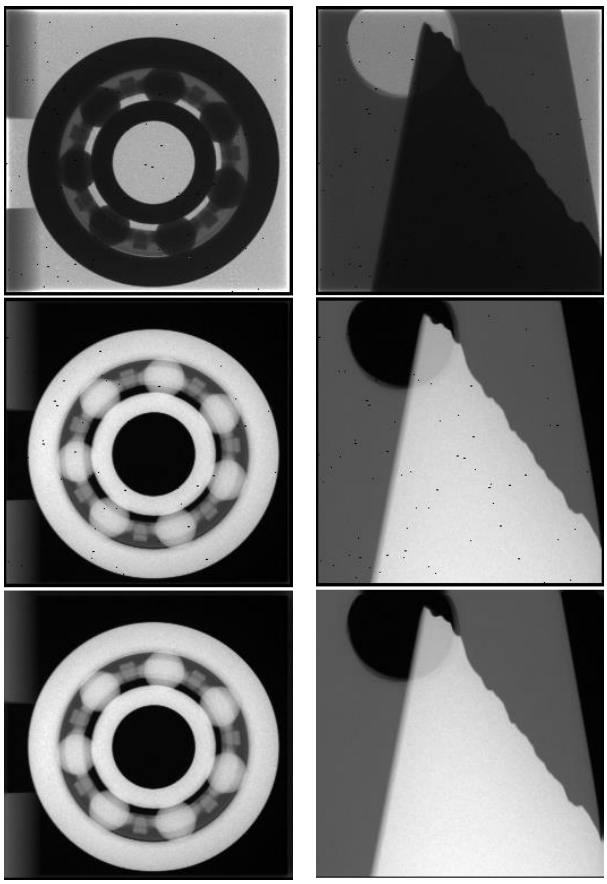
CdTe disadvantages: Uneven distribution of chip sensitivity. Slow instability can cause the need for a long total measurement time.
Specialy thick chips have a easily visible image distortion. Afther start-up, CdTe contains lot of free charges, they can cause problems during early measurements (use **pxcDoSensorRefresh**).

Si and CdTe have opposite bilas polarity. This can be used for type detection.



2. The beam hardening experimental setup

1 – The sensor
2 – Pb shielding of electronics (for sure, large intensities were tested in this experiment)
3 – several thin sheets, a total of 0.3 mm to reduce free beam / first ref. plate contrast
4 – 0.6 mm reference plates



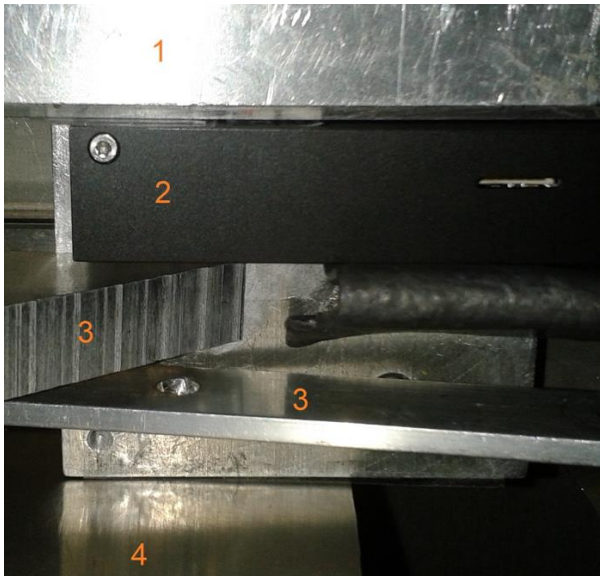
3. Test images

Left is Fe bearing, used reference plates and the setup from img. 2

Right is Al plates, 3 and 8 mm, reference plates was several 3mm sheets

Up – original images
Mid – afther pxcApplyBHCorrection
Down – mid afther pxcInterpolateBadPixels

(range in all images auto-resized to grayscale 0-255)



4. The Al plates experimental setup

1 – Cooler - Si chip used
2 – The Minipix
3 – Imaged aluminium plates
4 – Thin Al sheet to reduce free beam / first ref. plate contrast

3.5 The synchronizing

Sometimes it is necessary to synchronized with an external event. Or some instruments, Widepix-L for example, can be organized as set of more than 1 device in 1 box and more than 1 data cable go out. If you want start acquisition on whole instrument at exactly the sam time, must use synchronization by the hardware trigger.

3.5.1 Synchronizing basics

The acquisition functions have the TrgStg parameter. The 4th argument "trigger settings" of the pxcMeasureTpx3DataDrivenMode(), for example. This have options:

PXC_TRG_NO	No sync. Normally start immediatelly and stop after acqTime.
PXC_TRG_HWSTART	Wait and start the acq. on HW signal.
PXC_TRG_HWSTOP	Start the acq. immediatelly and stop on HW signal or after acqTime.
PXC_TRG_HWSTARTSTOP	Wait and start the acq. on HW signal and stop at next HW signal.
PXC_TRG_SWSTART	Wait and start the acq. on SW signal = pxcDoSoftwareTrigger().

Additionally, you can use the named parameters. You can read if the device is ready, is it master, etc or write some parameters. Details depends on the device type.

3.5.2 Synchronizing with an external source

Sync with the external source can depends on the device type. See the synchronization manual of your device type for hardware details.

Example: Way to use the external HW sync with the Minipix-Tpx3

1. Set some synchronization parameters: "TrgTimestamp", "TrgToSyncReset", "TrgOutLevel", "TrgOutEnable" using function to write the named parameters: pxcSetDeviceParameter.
2. Read some synchronization parameters: "TrgReady", "IsMaster" using the pxcGetDeviceParameter.
3. Start the pxcMeasure... with some PXC_TRG_HW...
4. Apply the sync signal to the input.

3.5.3 Synchronizing in multi-device instruments

Basic idea

1. Start the slave acquisition with trgStg = PXC_TRG_HWSTART
2. Start the master acquisition with trgStg = PXC_TRG_NO
3. Wait for both done
4. Get both frames

(3/4 can be realized in callback)

This can be simple if the continuous acquisition is used: This API function starts in separate thread:

```
rcs = pxcMeasureContinuous(slvIdx, bufCnt, acqTime, PXC_TRG_HWSTART, callback, (intptr_t)slvIdx);
rcm = pxcMeasureContinuous(masIdx, bufCnt, acqTime, PXC_TRG_NO, callback, (intptr_t)masIdx);
errorToList("pxcMeasureContinuous s,m", rcs, rcm);
```

But if using other acq. functions, you must start (1) in separate thread manually.
(see the Visual Studio example "Mpx3-2-sync")

Decision which device is master

```
isMaster = pxcGetDeviceParameter(devIdx, "TriggerMaster");
```

Output is 0 for slave, 1 for master or single, <0 if the device not supports it.
Note: For Timepix3 must use the `IsMaster` instead of the `TriggerMaster` .

4. Appendix

4.1 FitPIX device parameters

TriggerStg

settings of the trigger

Values:

- 0 - trigger reacts to logical 0 (0 V)
- 1 - trigger reacts to logical 1 (5 V)
- 2 - trigger reacts to rising edge
- 3 - trigger reacts to falling edge

TriggerWaitForReady

If the FitPIXes are connected in chain, the device waits for ready signal from the preceding device in the chain.

Values:

- 0 - does not wait
- 1 - waits

TriggerMaster

Sets the device to be the first one in the chain. If devices are connected in chain, master device has the external trigger connected.

Values:

- 0 - is not master device
- 1 - is master device

TriggerOutLevel

Sets the active level of Trigger Out pin

Values:

- 0 – logical 0 (0V)
- 1 – logical 1 (5V)

ADVACAM s.r.o.
U Pergamenky 1145/12, CZ 170 00 Praha 7
Czech Republic

Tel: +420-603-444112, 589854;
Email: info@advacam.com
www.advacam.com

For more information visit ADVACAM website at www.advacam.com
For any question send an email to info@advacam.com