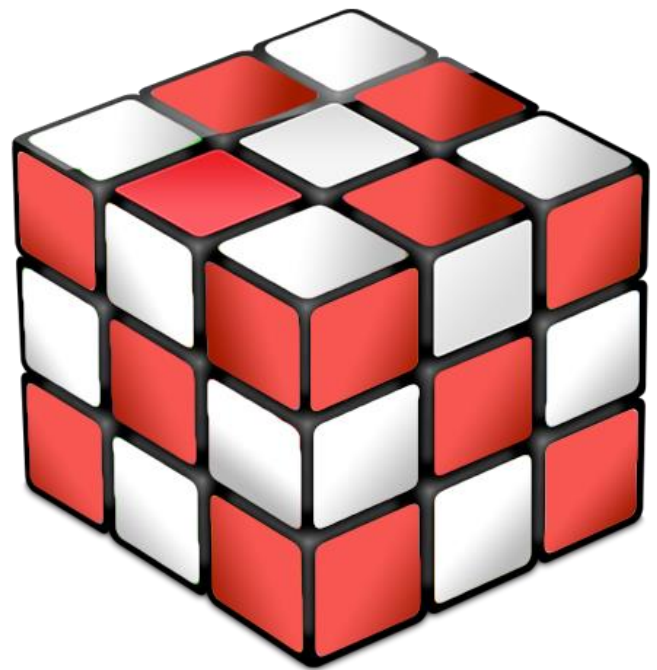


API Python

- Devices
- Processing
- File formats



*PIX*ET

Developer Documentation

1. Table of contents

1. Table of contents	2
2. Introduction.....	6
3. Requirements	6
3.1. Hardware.....	6
3.2. Software.....	6
3.2.1. Required files – in more detail.....	7
3.2.1.1. The Pixet core, python and additional libraries:.....	7
3.2.1.2. The pixet.ini file and the hwlibs	7
3.2.1.3. Additional files for some devices.....	7
3.2.1.4. The factory and the configs directories	8
3.2.1.5. Where to get these files?.....	8
4. Overview of the Python API	9
5. The pypixet object	10
5.1. Using	10
5.1.1. Example	10
6. The pixet object	11
6.1. Methods: General	11
6.2. Methods: Devices management	11
6.3. Methods: Directories informations	12
6.4. Methods: Other	12
6.5. Constants: Device and chip types.....	13
6.1. Constants: DevInfo types.....	13
6.2. Constants: Operation modes	14
6.3. Constants: File types, extensions and flags.....	15
6.3.1. File extensions and flags: TXT/PBF/PMF/BMF details	16
6.3.2. File extensions and flags: DSC details	18
6.3.3. File extensions and flags: CLOG and CLOG.IDX details	20
6.3.1. File extensions and flags: T3PA details	21
6.3.2. File extensions and flags: Pixel matrix configuration files	21
6.4. Constants: Acquisition types and modes, incl. triggers	22
6.5. Constants: Frame types	22
6.6. Constants: Settings types	22
6.7. Constants: Events.....	23
6.8. Constants: Data formats.....	24

6.9.	Constants: Threshold flags.....	24
6.10.	Constants: Pixels matrix config types	24
6.11.	Constants: Other	24
6.12.	Constants: Mpx3 settings.....	25
6.13.	Similar related to previous lists.....	26
6.13.1.1.	Data types	26
6.13.1.2.	Named parameters.....	26
6.13.1.3.	Error codes list	26
6.13.1.4.	Clustering constants and error codes.....	27
6.13.1.5.	Spectraimg constants and error codes.....	27
7.	Sensor device objects.....	28
7.1.	Methods: Device info	28
7.2.	Methods: Device config.....	29
7.2.1.	General loading and saving settings.....	29
7.3.	Methods: General	30
7.4.	Methods: Acquisition and related.....	31
7.4.1.	Simple and advanced acquisition	31
7.4.2.	Continuous acquisition	33
7.4.3.	Acquisition status	33
7.4.4.	Triggers	34
7.4.4.1.	Hardware trigger.....	34
7.4.4.2.	Software trigger	34
7.4.5.	Online data processing, events and callbacks	35
7.4.6.	Events: more details	35
7.5.	Methods: Other/Special	36
7.5.1.	Sensor refresh.....	36
7.5.2.	Threshold settings	36
7.5.3.	Bias settings	36
7.5.4.	Digital test.....	37
7.5.5.	DACs control	37
7.5.6.	Special.....	37
7.6.	Other sensor devices.....	38
7.6.1.	Timepix alias Medipix2	38
7.6.2.	Timepix2	38
7.6.3.	Medipix3	39
7.6.3.1.	Synchronizing multi-device Mpx3 instruments	41
8.	Other device objects	42

8.1.	The IDevMotor device	42
8.2.	The IDevXray device.....	42
9.	Accessing the named parameters	43
9.1.	The IParamMgr object and his methods.....	43
9.2.	The IParam object and his methods	43
9.3.	Parameters names lists by device types	44
9.3.1.	Tpx3 parameter names list	44
9.3.2.	Mpx2 parameter names list.....	45
9.3.3.	Mpx3 parameter names list.....	45
9.3.4.	Zest-wpxdev parameter names list	46
9.3.5.	Zem-wpx7dev parameter names list	46
10.	The pixels matrix configuration objects	47
10.1.	General.....	47
10.2.	IMpx2TpxPixCfg operation modes management.....	48
10.3.	Threshold correction bits management	48
10.4.	Masks management	49
11.	The IMpxDacs object.....	50
12.	The ISetting object	51
13.	The IMpxFrame and other frame objects	52
13.1.	Frame access and related methods	52
13.2.	Frame parameters methods.....	53
13.3.	Frame statistics and bulk mathematic methods.....	54
13.4.	Frame metadata and his handling methods.....	54
13.5.	Other frames methods	55
13.6.	Frame with subframes example	56
14.	The ITpx3Pixels and other pixels objects.....	57
14.1.	Basic pixels access and related methods.....	57
14.2.	Pixels other methods.....	58
14.3.	Pixels metadata methods	58
14.4.	The ITpx3Pixels example: After doAcquisition	59
14.5.	The ITpx3Pixels example: Callbacks.....	60
15.	The pypxproc object	61
16.	The Clustering object and related	62
16.1.	The Clustering methods	62
16.2.	The Clustering properties.....	63
16.3.	The Clusters object	63

16.4.	The Cluster object	63
16.5.	The Pixel object.....	63
16.6.	Clustering constants and error codes	64
16.7.	The Clustering examples	65
16.7.1.	Simple list of the clusters.....	65
16.7.1.	List of the clusters with pixels sample	66
16.7.2.	The clusters statistics using newClustersCallback	67
17.	The Spectralmg object	68
17.1.	Spectralmg auxiliary methods and properties	69
17.2.	Spectraimg measurement and related methods	69
17.3.	Spectraimg data output	70
17.4.	Spectraimg and data files.....	71
17.5.	Spectralmg constants and error codes.....	71
17.6.	Spectralmg examples.....	72
17.6.1.	Simple measuring and list of the spectrum (Tpx3 only)	72
17.6.2.	Begin and end for online mode examples.....	73
17.6.3.	Measuring and getFrameForEnergy	74
17.6.4.	Measuring and getFrameForEnergyRange	75
17.6.5.	Offline data processing.....	76
17.6.6.	Get a data for the offline processing	77
17.6.7.	Save processed data to BSTG file for future use	78
17.6.8.	Load and use previous saved BSTG data	78
17.6.9.	Measuring examples results (getFrameForEnergyRange).....	79
18.	Pixet plugin and associated objects.....	80
18.1.	The IDevControl object	80
18.2.	The IMmeasutils object	81
18.3.	The IClustering object.....	82
19.	The pygui object	83
20.	The Window and associated objects	84
20.1.	The GridLayout object and its methods	85
20.2.	The Widget, TabWidget and GroupBox objects.....	86
20.2.1.	The Widget object and its methods	86
20.2.2.	The GroupBox object and its methods	86
20.2.3.	The TabWidget object and its methods.....	86
20.3.	The PropertyTreeView object and its methods	87
20.4.	The MpxFrame object and its methods	88
20.4.1.	Timepix3 measuring with MpxFrame widget example	89

20.5.	The MpxFramePanel object and its methods.....	92
20.6.	The Plot object and its methods.....	93
20.7.	Other graphical objects and its methods	95

2. 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 python interface of the **PIXet** software. This interface consists of **pypixet.pyd** and **pypxproc.pyd**.

3. Requirements

3.1. Hardware

This API requires computer with x86 compatible architecture (no ARM, but can be on request), 64bit Windows or Linux and connected some Advacam hardware with imaging chip. Medipix3, Timepix, Timepix3, etc.

Some functions are universal for all hardwares (pypixet.start(), dev.doSimpleAcquisition(...), etc).

Some functions are not working with all (dev.setOperationMode(...)) is not working with devices that has configurable individual pixels).

Some functions are specialized for only one chip type (dev.setColorMode(...)) is Mpx3 only).

3.2. Software

The Pixet Python API can be used from the Python 2.7 interpreter integrated in the Pixet program or from commandline with external Python up to 3.x without the Pixet.

For starting from the Pixet Python scripting plugin are not need any special files.

If you want to run scripts without the Pixet, need additional files:

API functions using of **pypixet.pyd** and **pypxproc.pyd** and need **Python versions 2.7 to 3.x**, for Windows the Pixet core dlls: **pxcore.dll**, **pxproc.dll**, or linux **.so** equivalents. The library is **64bit only**.

The Pixet core need the **pixet.ini** file with proper hwlibs list inside, necessary hardware dll files (eq **minipix.dll**), subdirectory "**configs**" with config files for all present imaging chips (eq MiniPIX-I08-W0060.xml).

Pixet core on Windows need more Microsoft Visual Studio .NET standard dlls (vccorlib140.dll etc).

3.2.1. Required files – in more detail

See: https://wiki.advacam.cz/wiki/Files_and_directories_of_the_Pixet_and_SDK

3.2.1.1. The Pixet core, python and additional libraries:

pypixet.pyd (allways), **pypxproc.pyd** (clustering/spectral imaging)
pxcore.dll (allways), **pxproc.dll** (clustering/spectral imaging) or equivalent SO files on Linux

3.2.1.2. The **pixet.ini** file and the **hwlibs**

In the active directory must be the **pixet.ini** file. It must contain 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:	<pre>[Hwlibs] hwlibs\minipix.dll hwlibs\zest.dll</pre>	Example2:	<pre>[Hwlibs] minipix.dll zest.dll ;zem.dll</pre>
------------------	--	------------------	---

(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.

The **pixet.ini** can contains the **[Settings]** section. For API are relevant lines:

FileDevice=false If enabled, the FileDevice present as device[0] if no physical device connected.

UseAppDataDir=true If enabled, the configs and logs directories are located in user’s **appData** or **.config** on Linux.

Hwlibs list:	<pre>Minipix: minipix.dll Widepix with Eth: zest.dll Widepix without Eth: widepix.dll Advapix: zem.dll, okFrontPanel.dll</pre>
---------------------	--

3.2.1.3. Additional files for some devices

Device INI files list:	<pre>Widepix with Eth: zest.ini Widepix without Eth: widepix.ini</pre>
-------------------------------	---

Firmware images list:	<pre>Widepix-L: zestwpx.bit Widepix-F: zemwpxf.rbf Advapix-tpx3: zemtpx3.rbf Advapix-tpx3-quad: zemtpx3quad.rbf Advapix-Timepix: zemtpx.rbf</pre>
------------------------------	--

3.2.1.4. 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 or program can use it by the **loadFactoryConfig()** method.

The **configs** directory contain configuration XML files. The Pixet core try to use it while starting in **pixet.start()** method and automatically save the current settings to it, if the **pixet.exit()** method is used.

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

Note: The files must have proper names. "WidePIX-B04-W0535-10.xml" for example. If not, automatic load at start and by the **loadFactoryConfig()** method will not work. To obtain proper config name, you can do the **pixet.exitPixet()** and see saved filename in the configs directory. (see [Methods: Device config](#))

3.2.1.5. Where to get these files?

All need files are located in the Pixet directory:

drivers	composite.ini	msvcp140.dll	pypxproc.pyd	zem.ini
factory	dummy.ini	msvcr110.dll	python37.dll	zemmpx3.rbf
help	dxs.ini	msvcr120.dll	Qt5Core.dll	zemtpx.rbf
hwlibs	fitpix.ini	okFrontPanel.dll	Qt5Gui.dll	zemtpx2.rbf
libs	hvsrc.ini	Pixet.exe	Qt5Network.dll	zemtpx3.rbf
platforms	image.txt	pixet.ini	Qt5PrintSupport.dll	zemtpx3pix.rbf
plugins	lic.info	poscontroller.ini	Qt5Widgets.dll	zemtpx3quad.rbf
samples	mfc110.dll	pxcore.dll	usb.ini	zemwpxf.rbf
scripts	mfcloop.dll	pxproc.dll	vccorlib140.dll	zest.ini
styles	minipix.ini	pypixet.pyd	vcruntime140.dll	zestwpx.bit
bspulse.dll	msvcp120.dll	pypixetgui.pyd	widepix.ini	zip.exe

You can simply copy all selected files to your python project directory. If you want minimum files, you can copy only common files and files for your device. Example for Minipix:

drivers	composite.ini	msvcp140.dll	pypxproc.pyd	zem.ini
factory	dummy.ini	msvcr110.dll	python37.dll	zemmpx3.rbf
help	dxs.ini	msvcr120.dll	Qt5Core.dll	zemtpx.rbf
hwlibs	fitpix.ini	okFrontPanel.dll	Qt5Gui.dll	zemtpx2.rbf
libs	hvsrc.ini	Pixet.exe	Qt5Network.dll	zemtpx3.rbf
platforms	image.txt	pixet.ini	Qt5PrintSupport.dll	zemtpx3pix.rbf
plugins	lic.info	poscontroller.ini	Qt5Widgets.dll	zemtpx3quad.rbf
samples	mfc110.dll	pxcore.dll	usb.ini	zemwpxf.rbf
scripts	mfcloop.dll	pxproc.dll	vccorlib140.dll	zest.ini
styles	minipix.ini	pypixet.pyd	vcruntime140.dll	zestwpx.bit
bspulse.dll	msvcp120.dll	pypixetgui.pyd	widepix.ini	zip.exe

And You can delete unused files form **copied** hwlibs directory. For Minipix, only minipix.dll is needed.

4. Overview of the Python API

The python API can be used:

- directly in the system console, using a general python interpret
- in the Pixet program integrated python console

The base is **pypixet object**. It has methods for initialize and deinitialize, can create the pixet object. The **pixet object** have device list, can create device objects and allows access to global properties.

See: [The pypixet object](#) and [The pixet object](#)

A **device objects** have methods for acquisitions and allows access to device parameters.

See: [Sensor device objects](#) and [Other device objects](#)

The **pypxproc object** is intended for use to processing of a data.

See: [The pypxproc object](#), [The Clustering object and related](#) and [The Spectralmg object](#)

(and many objects for specialized purposes)

The **pygui object** allows you to create your own graphical interface.

It can be used only if a script is run from the Pixet program. See: [The pygui object](#)

A **plugins objects** allows you to acces a plugins of the Pixet Program.

It can be used only if a script is run from the Pixet program. See: [Pixet plugin and associated objects](#)

Small code example for using in the **system console** or other third-party environment, with the Python 3.7:

```
import pypixet

print("pixet core init...")
pypixet.start()
pixet=pypixet.pixet
devices = pixet.devicesByType(pixet.PX_DEVTYPE_TPX3)
dev = devices[0]
rc = dev.setOperationMode(pixet.PX_TPX3_OPM_EVENT_ITOT)
print("dev.setOperationMode:", rc, "(0 is OK)")

print("dev.doSimpleAcquisition (3 frames @ 1 sec) - start")
rc = dev.doSimpleAcquisition(3, 1, pixet.PX_FTYPE_AUTODETECT, "example.png")
print("dev.doSimpleAcquisition - end:", rc, "(0 is OK)")

pixet.exitPixet() # save settings, correct stop devices and core exit
pypixet.exit()   # both lines important if a third-party debug environment used
```

Small code example for using in the **Pixet python console** with integrated Python:

```
# do not create the pypixet and pixet, they exist by default
devices = pixet.devicesByType(pixet.PX_DEVTYPE_TPX3)
dev = devices[0]
rc = dev.setOperationMode(pixet.PX_TPX3_OPM_EVENT_ITOT)
print("dev.setOperationMode: %i (0 is OK)" % rc)
print("dev.doSimpleAcquisition (3 frames @ 1 sec) - start")
rc = dev.doSimpleAcquisition(3, 1, pixet.PX_FTYPE_AUTODETECT, "example.png")
print("dev.doSimpleAcquisition - end: %i (0 is OK)" % rc)
# do not execute the pixet.exitPixet(), it will cause whole the Pixet program to exit
```

5. The pypixet object

5.1. Using

start() Initialize the Pixet core.
If used from interpreter in the Pixet program, run next instance of Pixet.

Example: `pypixet.start()`

Warning: If `pypixet.start()` was used, and then it was used again, returns error code. Before repeat of the `pypixet.start()`, always must be used `pypixet.exit()`. See `exit()` for details.

Note: Unexpected charges may appear in the chip after power-up/initialization. The first measurement, especially Tpx3 data-driven, often generates meaningless data for a few seconds. Use a dummy acquisition before the first real measurement.

exit() Deinitialize the Pixet core. Save actual configuration to the default config file.
If used from interpreter in the Pixet program:

- and `pypixet.start()` was used, terminates both new and original Pixet instance.
- and `pypixet.start()` was not used, do nothing.

Example: `pypixet.exit()`

Warning: To correct end, the **`pixet.exitPixet()`** must be used before (if `pixet` is `pypixet.pixet`). This is not problem if the script is used by commandline like as `python myscript.py`, but can cause problems if debug environment used (in Spyder for example).

pixet The pixet object.

Example: `pixet = pypixet.pixet`

isrunning() Test when the Pixet core is running

Example: `print("Pixet core running:", pypixet.isrunning())`

startgui() Initialize the Pixet core. Unlike the **`start()`** launches the graphical interface.

5.1.1. Example

```
import pypixet      # Requires pxcore.dll/so, this requires hwlib(s) of device(s),
                   # the hwlib can require other file(s).
print("pixet core init...")
pypixet.start()
pixet=pypixet.pixet
devices = pixet.devices()
```

6. The pixet object

The pixet object provide methods to manage connected devices, global configuration and system constants for many purposes.

6.1. Methods: General

exitPixet() Destroy data structures and correctly terminates all devices. Recommended before exiting the program, otherwise it may not work properly the next time you start and the device will need to be disconnect and connected and Python must be restarted. Save actual configuration to the default config file, if not exist.

registerEvent (eventName, reserved, callbackFun) Register event to use the callback function

unregisterEvent (eventName, reserved, callbackFun) Unregister event to stop using callback

eventName See [Constants: Events](#)

reserved Reserved to future use, now it doesn't matter the value or the type, recommend use 0.

callbackFun Name of the function that can be called by the event.

Note: Events on the pixet object are not widely used. They are more often used on devices. See [Methods: Events](#)

6.2. Methods: Devices management

devices() Array of all connected devices. If no devices connected, there is the virtual "FileDevice 0"

devicesByType(type) Array of all connected devices of desired type see [Constants: Device type](#)

devicesMpx2() Array of all connected Medipix2 devices

devicesMpx3() Array of all connected Medipix3 devices

devicesTpx3() Array of all connected Timepix3 devices

deviceCount() All connected devices count. If no devices connected, count is 1 (see **devices()**)

refreshDevices() Refresh devices list. It can be used after connect new device while prg. is running.

DevInfo() Returns the **DevInfo** object to be used with **dev. deviceInfo**.

```
Example1:  devices = pixet.devices()
           if devices[0].fullName()=="FileDevice 0":
               print("No devices connected")
               pixet.exitPixet()
               pypixet.exit()
               exit()

           dev0 = devices[0] # first of connected devices
```

```
Example2:  devices = pixet.devicesByType(pixet.PX_DEVTYPE_TPX3)
           if len(devices)==0:
               print("No Tpx3 devices connected")
               pixet.exitPixet()
               pypixet.exit()
               exit()

           dev0 = devices[0] # first of connected Tpx3 devices
```

Example3:

```

di = pixet.DevInfo()
rc = dev.deviceInfo(di)
print("Last used dev info:")
print("  name", di.name)
print("  serial", di.serial)
print("  type", di.type)
print("  vendor", di.vendor)

```

Note: The type record contains device HW type. It can distinguish not only the chip type, but also the HWlib type.

6.3. Methods: Directories informations

appDir()	Full path to the running program
appDataDir()	Full path to auxiliary dirs default loading/saving (configs, logs, factory)
factoryDataDir()	Full path to the factory data – used by dev.loadFactoryConfig() method
logsDir()	Full path where logs are saved – created automatically at program start
configsDir()	Full path to configs loading/saving – created automatically at program start
licenseName()	Returns registration name of the Pixet core.
licenseSerial()	Returns registration serial of the Pixet core.

Example:

```

pixet=pypixet.pixet
print("pixet.appDataDir", pixet.appDataDir())
print("pixet.appDir", pixet.appDir())
print("pixet.configsDir", pixet.configsDir())
print("pixet.factoryDataDir", pixet.factoryDataDir())
print("pixet.logsDir()", pixet.logsDir())

```

6.4. Methods: Other

pixetVersion()	Returns the Pixet version string, like as "1.8.1 Build b24c5dec (240116)"
pixetAPIVersion()	Returns pxcore API version number, like as 1810
pixetType()	Returns Pixet type number. Normally it is PX_PIXETTYPE_PRO, in Pixet Lite is PX_PIXETTYPE_LITE or PX_PIXETTYPE_TIMEPIX3 (obsolete transitional version).
plugin(name)	Returns plugin object. See Pixet plugin and associated objects Example: plugin=pixet.plugin("name").privateInterface()
createSettings(filePath, stgType)	Returns the ISettings object. See: The ISetting object Example: pixet.createSettings("test.xml", pixet.PX_STGTYPE_XML)
createPluginSettings(name, IDev, stgType)	Returns the ISettings object associated with the plugin. Example: pixet.createPluginSettings("IDevControl", dev.asIDev(), pixet.PX_STGTYPE_XML)

6.5. Constants: Device and chip types

These constants are a device type indexes and can be used for example:

```
# list of all connected Timepix3 devices
devices = pixet.devicesByType(pixet.PX_DEVTYPE_TPX3)
# Warning: PX_DEVTYPE_... != PX_CHIPTYPE_... for the same devices
if dev.chipType() == pixet.PX_CHIPTYPE_TPX3: print("chipType OK")
```

Sensor devices types list:

```
PX_DEVTYPE_MPX2
PX_DEVTYPE_MPX3
PX_DEVTYPE_TPX3
PX_DEVTYPE_TPX2
PX_DEVTYPE_MPX4
PX_DEVTYPE_TPX4
```

Special devices types list:

```
PX_DEVTYPE_DATADEV
PX_DEVTYPE_MOTOR
PX_DEVTYPE_XRAY
PX_DEVTYPE_AUX
PX_DEVTYPE_HVSOURCE
```

Chip types list:

```
PX_CHIPTYPE_MXR
PX_CHIPTYPE_TPX
PX_CHIPTYPE_MPX3
PX_CHIPTYPE_TPX3
PX_CHIPTYPE_TPX2
PX_CHIPTYPE_MPX4
PX_CHIPTYPE_TPX4
```

6.1. Constants: DevInfo types

dev.deviceInfo(pixetDI).type See: [Methods: Devices management, Constants: DevInfo types](#)

```
PX_DEVINFO_TYPE_DUMMY
PX_DEVINFO_TYPE_FILEDEV
PX_DEVINFO_TYPE_MINIPIX
PX_DEVINFO_TYPE_WIDEPIX
PX_DEVINFO_TYPE_ZEM_ADVAPIX
PX_DEVINFO_TYPE_ZEM_WIDEPIX
PX_DEVINFO_TYPE_ZEM_ADVAPIX_TPX3
PX_DEVINFO_TYPE_ZEST_WIDEPIX_MPX3
PX_DEVINFO_TYPE_MINIPIX_EDU
```

Virtual dummy device generated by ini file
 Virtual file device generated if no devices found
 The MiniPIX device (minipix.dll hwlib)
 The WidePIX device with USB2 (obsolete)
 The AdvaPIX device, except Tpx3, with USB3 (zem.dll hwlib)
 The WidePIX device with USB3 (zem.dll hwlib)
 The AdvaPIX Tpx3 device with USB3 (zem.dll hwlib)
 The WidePIX Mpx3 device with GB Eth (zest.dll hwlib)
 The MiniPIX EDU device (education version – limited features)

6.2. Constants: Operation modes

These constants are a device operation mode indexes. For various types of devices existing various sets of operation modes and constants sets. It can be used for example:

```
# set the Timepix3 operation mode "Time of arrival"
dev.setOperationMode(pixet.PX_TPX3_OPM_TOA)
```

Medipix3 OPM list:

```
PX_MPX3_OPM_SPM_1CH
PX_MPX3_OPM_SPM_2CH
PX_MPX3_OPM_CSM
PX_MPX3_OPM_SPM_1CH_COLOR**
PX_MPX3_OPM_SPM_2CH_COLOR**
PX_MPX3_OPM_CSM_COLOR**
PX_MPX3_OPM_CUSTOM
```

Timepix2 OPM list:

```
PX_TPX2_OPM_TOT10_TOA18
PX_TPX2_OPM_TOT14_TOA14
PX_TPX2_OPM_CONT_TOT10_CNT4
PX_TPX2_OPM_CONT_TOT14
PX_TPX2_OPM_CONT_TOA10
PX_TPX2_OPM_CONT_TOA14
PX_TPX2_OPM_CONT_CNT10
PX_TPX2_OPM_CONT_CNT14
PX_TPX2_OPM_ITOT10_TOA18
PX_TPX2_OPM_ITOT14_TOA14
PX_TPX2_OPM_CONT_ITOT10_CNT4
PX_TPX2_OPM_CONT_ITOT14
```

Timepix3 OPM list:

```
PX_TPX3_OPM_TOATOT
PX_TPX3_OPM_TOA
PX_TPX3_OPM_EVENT_ITOT
PX_TPX3_OPM_TOT_NOTOA
```

Timepix modes list: *

```
PX_TPXMODE_MEDIPIX
PX_TPXMODE_TOT
PX_TPXMODE_1HIT
PX_TPXMODE_TIMEPIX
```

OPMs summary:

TOA	Time of arrival
TOT	Time over threshold
EVENT	Event count
ITOT	Integrated time over threshold
CONT	Continuous mode
CNT	Count (=Event count)
SPM	Single pixel mode (=Event count)
CSM	Charge summing mode
MEDIPIX	(=Event count)
TIMEPIX	(=ToA)
1HIT	The Timepix OPM bits undocumented combination (experimental only)
COLOR**	Spectral imaging using coupled pixels with more thresholds

*Timepix (IMpx2) modes

cannot be set by setOperationMode.

Use **pixcfg.setModeAll** or set the mode for each pixel.

See [IMpx2TpxPixCfg operation modes management](#)

****COLOR modes** working properly only with specially bonded chips with coupled pixels.

6.3. Constants: File types, extensions and flags

These constants are file types and extensions. It can be used for filenames testing or with acquisition methods.

Example:

```
# measure and save one 0.25 second frame to png file named "testFile.png"
dev.doSimpleAcquisition(1, 0.25, pixet.PX_FTYPE_PNG, "testFile")
dev.doSimpleAcquisition(1, 0.25, pixet.PX_FTYPE_AUTODETECT, "testFile.png")
```

File type constants

```
PX_FTYPE_NONE
PX_FTYPE_AUTODETECT
PX_FTYPE_ASCII_FRAME
PX_FTYPE_BINARY_FRAME
PX_FTYPE_MULTI_FRAME
PX_FTYPE_BINARY_MULTIFRAME
PX_FTYPE_TPX3_PIXELS
PX_FTYPE_TPX3_PIXELS_ASCII
PX_FTYPE_CLUSTER_LOG
PX_FTYPE_PIXEL_LOG
PX_FTYPE_PNG
PX_FTYPE_TPX3_RAW_DATA
PX_FTYPE_PIXET_RAW_DATA
PX_FTYPE_EXTERNAL
(description file saved automatically with pmf/txt)
(index file saved automatically with pmf/txt)
```

File extensions constants

```
(No direct file saving – data stored only in memory)
(FTYPE detected by extension in a filename)
PX_EXT_ASCII_FRAME          "txt"
PX_EXT_BINARY_FRAME         "pbf"
PX_EXT_MULTI_FRAME          "pmf"
PX_EXT_BINARY_MULTIFRAME    "bmf"
PX_EXT_TPX3_PIXELS          "t3p"
PX_EXT_TPX3_PIXELS_ASCII    "t3pa"
PX_EXT_CLUSTER_LOG          "clog"
PX_EXT_PIXEL_LOG            "plog"
PX_EXT_PNG                   "png"
PX_EXT_TPX3_RAW_DATA        "t3r"
PX_EXT_PIXET_RAW_DATA       "prd"
PX_EXT_FRAME_DESC           "dsc"
PX_EXT_INDEX                 "idx"
```

txt ASCII matrix: Text files with img lines converted to text lines with numbers separated by spaces.

pbf Simple binary files, numbers only.

pmf Multiple frames. Default is same as the txt, but multiple frames on top of each other. Can use BINARY flag.

t3pa Tpx3 pixels ASCII. Text format, tab-separated columns with the header in the first row. Biggest to saving.

t3p Tpx3 pixels. Binary format. Lower saved size, more complex to understand.

t3r Tpx3 raw data. Fastest to saving, difficult to understand, slow to processing and can cause processing errors.

bstg Binary settings file: Measured data with all configuration (see [Spectraimg methods and properties](#)).

clog, plog Clusters/pixels logs. Text files contains clusters separated to frames with pixels lists. Historic formats for saving a data with few hited pixels in a frames.

h5 HDF5, hierarchical data format 5. Used as one of multi-frame formats.

info Text file with "[FileInfo]" head and all metadata list (see [Frame metadata and his handling methods](#)).

dsc Text file with frame index first and all metadata list (see [Frame metadata and his handling methods](#)).

idx Binary index for multi-frame files. Usesfull for fast access to n-th frame.

Frame file saving flags (Use in **save** or **doAdvancedAcquisition** methods. Flags can be combined.)

Deault is set of separate subframes text files, with all pixels include zeros, each subframe with idx+dsc files:

file_ToT.pmf, file_ToT.pmf.dsc, file_ToT.pmf.idx, file_ToA.pmf, file_ToA.pmf.dsc, file_ToA.pmf.idx

```
PX_FRAMESAVE_BINARY          Use binary format in pmf.
PX_FRAMESAVE_SPARSEX         Index + non-zero pixels in file. # separates (sub)frs.
PX_FRAMESAVE_SPARSEXY        X, Y + non-zero pixel in file. # separates (sub)frames.
PX_FRAMESAVE_NODSC           Do not add dsc file.
PX_FRAMESAVE_NOSUBFRAMES     Do not use subframes, save main frame only.
PX_FRAMESAVE_SUBFRAMES_ONEFILE Save all subframes to a single file.
PX_FRAMESAVE_SUBFRAMES_SAVEMAINFRAME Save separate all subframes and main frame extra.
```

6.3.1. File extensions and flags: TXT/PBF/PMF/BMF details

Overview:

txt	Single frame in the text file.
pbf	Pixet Binary Frame - Single frame in the binary file.
pmf	Pixet Multi Frame - Multiframe file with text or binary format, depends on flags used with saving.
bmf	Binary Multi Frame - Multiframe file with binary format, depends on flags used with saving.
idx	Index for seeking - Binary array of 64b pointers to start of frames and subframes.
dsc	Description. List of all metadata for each frame and subframe. Actual device and acquisition parameters, data types, etc. The "Frame name" item can be helpful to orientation in pmf structure if the ONEFILE flag used. The Type= item is helpful to understanding the structure of data if the BINARY flag used.

Note: All the next examples are for Timepix3, single chip, opm = pixet.PX_TPX3_OPM_TOATOT

Multi-files names generation:

flags 0 (default), input filename: "name", acqCount = 1

name_ToA.txt, name_ToA.txt.dsc, name_ToT.txt, name_ToT.txt.dsc

acqCount = 6:

name_0_ToA.txt, name_0_ToA.txt.dsc, name_0_ToT.txt, ...

...

name_5_ToA.txt, name_5_ToA.txt.dsc, ...

PMF note: With each pmf generating .pmf.idx binary file, other is same as TXT with acqCount = 1.

TXT file data, default:

0 0 0 5 0 0 0 ... 256 numbers (int for non-calibrated values or float if the calibration used) and enter
0 872 0 0 0 ... 256 numbers (int for non-calibrated values or float if the calibration used) and enter
(256 lines)

PMF file data, default:

0.00000 78.65742 0.00000 ... 256 numbers (int for non-calibrated values or float if the calibration used) and enter
0.00000 0.00000 999785.5 ... 256 numbers (int for non-calibrated values or float if the calibration used) and enter
(256 lines * acqCount)

The **PMF.IDX** files generated beside the PMFs. Contains the simple binary array of little-endian qword addresses of the each frame start.

.pmf.idx example:

Pointers to frames at 0, 0x29, 0x52, 0x7b, 0xA4, 0xCD, ...

TXT file data, pixet.PX_FRAMESAVE_SPARSEX flag:

_ToA.txt file: px index, ToA _ToT.txt file: px index, ToT

```
-----
0      227212.500000          0      20
17     310685.937500          17     13
255    265487.500000          255    11
1274   105728.125000          1274    9
```

(lists of all hited pixels, int for non-calibrated data or float if the calibration used)

	0001	0203	0405	0607	0809	0A0B	0C0D	0E0F
000	0000	0000	0000	0000	2900	0000	0000	0000
010	5200	0000	0000	0000	7B00	0000	0000	0000
020	A400	0000	0000	0000	CD00	0000	0000	0000
030	F600	0000	0000	0000	3401	0000	0000	0000

TXT file data, pixet.PX_FRAMESAVE_SPARSEXY flag:

_ToA.txt file: X, Y, ToA

_ToT.txt file: X, Y, ToT

_ToA.txt file: X, Y, ToA			_ToT.txt file: X, Y, ToT		
247	3	189851.562500	247	3	16
250	4	140042.187500	250	4	12
5	9	317195.312500	5	9	5

PMF file data, pixet.PX_FRAMESAVE_SPARSEX(Y) flag:

Same as TXT, but containing single lines with only # to separate frames:

```

232 139 321620.312500
4 252 340231.250000
#
39 0 258270.312500
201 0 76593.750000
92 1 268642.187500

```

PX_FRAMESAVE_SUBFRAMES_ONEFILE:

All the data is in one file, subframes are placed one behind the other. If the measurement result has 10 frames with 2 subframes A/B, each _n TXT file contains 2 subframes and the PMF contains 20 frames in order:

sfr0A, sfr0B, sfr1A, sfr1B, ...

The exact order and names of type of (sub)frames is listed in the DSC file. The DSC have separate records [Fn] for all the items.

PX_FRAMESAVE_SUBFRAMES_SAVEMAINFRAME:

The group of the saved files contains the main frame and all subframes. Subframe files end in _sfrName, the main frame does not. In DSC file accompanying the TXT with main frame is not the "Frame name" item.

Not applicable if combined with the **ONEFILE** flag.

PX_FRAMESAVE_BINARY:

If the file type supports text and binary format, ex. PMF, save the binary.

Not applicable to TXT, must use PBF instead.

Data in the file are the simple array of non-calibrated 16 or 32b integers or calibrated doubles. See the DSC file for used data type.

BINARY + SPARSEXY examples:

	0001	0203	0405	0607	0809	0A0B	0C0D	0E0F	
000	2900	0000	0100	0000	0000	0000	1EA9	1541	X 32b
010	1D00	0000	0200	0000	0000	0040	ADC0	1541	Y 32b
020	FC00	0000	0300	0000	0000	0080	30FF	1341	data double

	0001	0203	0405	0607	0809	0A0B	0C0D	0E0F	
000	2900	0000	0100	0000	3A00	1D00	0000	0200	X 32b
010	0000	0A00	FC00	0000	0200	0000	0C00	EF00	Y 32b
020	0000	0300	0000	0A00	F700	0000	0300	0000	data 16b

6.3.2. File extensions and flags: DSC details

The first 3 lines have special functions:

1. File type selection
2. Index of the frame in this file
3. Arrangement and type of the data

Other lines are a list of metadata items separated by blank lines.

In txt.dsc and pbf.dsc, end of the frame is end of the file.

In the pmf.dsc, next item 2 and 3 and frame or subframe data follows.

Some example (PBF with BINARY and SPARSEXY – test_49_ToA.pbf.dsc):

B000000001	B=binary / A=ASCII and file type
[F0]	Index of frame in the file = 0
Type=double [X,Y,C] width=256 height=256	Pixel pos. X, Y and data with type double
"Acq Serie Index" ("Acquisition serie index"):	Some metadata item name and (description)
u32[1]	Type of the item data [number of values]
49	The value

(more metadata items separated by blank lines ...)

"Frame name" ("Frame name"):

char[3]

ToA

This is the ToA frame

(more metadata items separated by blank lines ...)

(end of the file)

Other example (PMF with BINARY+SPARSEX+ONEFILE – test.pmf.dsc):

B000000010	
[F0]	Start of the first subframe
Type=double [X,C] width=256 height=256	Pixel index and double type pixel data (ToA in ns)
"Acq Serie Index" ("Acquisition serie index"):	
u32[1]	
0	

(more metadata items separated by blank lines ...)

"Frame name" ("Frame name"):

char[3]

ToA

(more metadata items separated by blank lines ...)

[F1]	Start of the second subframe
Type=i16 [X,C] width=256 height=256	Pixel index and int16 pixel data (ToT counter value)
"Acq Serie Index" ("Acquisition serie index"):	
u32[1]	
0	

(and the ToT frame metadata, [F2] and ToA subframe, [F3] and ToT sfr, ... [Fn] and ToT sfr of (n/2)th frame)

Complete one frame DSC example (BINARY+SPARSEX – test_15_ToA.pbf.dsc):

B000000001

[F0]

Type=double [X,C] width=256 height=256

"Acq Serie Index" ("Acquisition serie index"):

u32[1]

15

"Acq Serie Start time" ("Acquisition serie start time"):

double[1]

1639059034.903085

"Acq time" ("Acquisition time [s]"):

double[1]

0.500000

"ChipboardID" ("Chipboard ID"):

char[9]

I08-W0060

"DACs" ("DACs"):

u16[19]

16 8 128 10 120 1301 501 5 16 8 16 8 40 128 128 128 256 128 128

"Frame name" ("Frame name"):

char[3]

ToA

"HV" ("High voltage [V]"):

double[1]

-500

"Interface" ("Readout interface"):

char[7]

MiniPIX

"Mpx type" ("Medipix type (1-MXR, 2-TPX, 3-MPX3, 4-TPX3, 5-TPX2)"):

i32[1]

4

"Pixet version" ("Pixet version"):

char[5]

1.7.8

"Start time" ("Acquisition start time"):

double[1]

1639059042.934810

"Start time (string)" ("Acquisition start time (string)"):

char[64]

Thu Dec 9 15:10:42.934809 2021

"Threshold" ("Threshold [keV]"):

double[1]

5.026744

6.3.3. File extensions and flags: CLOG and CLOG.IDX details

The CLOG format was developed to facilitate further processing of cluster data by the user programs. This is a text file divided to the frame records and the records can contain a clusters. Frames and clusters are separated by the line breaks. Frames can be separated by whole free line.

The record format:

Frame FN (frameStart, frameAcqTime s)

[x, y, energy, ToA] [x, y, energy, ToA] [x, y, energy, ToA] ...

FN	Frame index number. First 0 or 1.
frameStart	Start time of the frame. There are variants: <ol style="list-style-type: none"> 1. If it from measuring or from replay frame-based data with metadata available: Linux format, frame starting time from PC's getPrecisionTime. 2. If it from pixel-based data with metadata available (file.t3pa + file.t3pa.info): Linux format, acq. starting time from PC's getPrecisionTime with added time from data. 2. If it from replay data and metadata not available: Nanoseconds from the input data. Periodic increments if source is frame-based, random increments if source is data-driven.
frameAcqTime	Duration of the frame, float in seconds. Always 0.000000 in data from data-driven sources.
x, y	Position of the pixel.
energy*	Energy deposited in the pixel. Integer ToT counter value if not calibrated, float in keV if calibrated.
ToA*	Time of arrival, relative to frameStart. Integer in CLK ticks if ToA conversion is disabled, float in ns if ToA conversion is enabled.

*ToA+energy records can be created from source that supports combined ToA+ToT modes, like as OPM_TOATOT on the Timepix3. If the data source supports only single modes, only one value is in this position.

Clog from data-driven source not contains free frames.

Clog from frame-based source can contains free frames.

Example records (Timepix3, Frame2 with two clusters by 2 and 4 pixels, Frame3 with single 2-pixel cluster)

Frame 2 (273697060.937500, 0.000000 s)

[214, 195, 43.1598, 0] [220, 191, 20.6515, 7.8125]

[224, 182, 21.8018, 31.25] [223, 186, 4.58576, 31.25] [222, 183, 38.2381, 31.25] [226, 185, 14.7623, 34.375]

Frame 3 (371034565.625000, 0.000000 s)

[151, 33, 32.5745, 0] [151, 34, 13.8135, 17.1875]

Example records (Timepix)

Frame 6 (1639143482.765164, 0.200000 s)

[87, 134, 5.75352] [217, 58, 14.8396]

Frame 7 (1639143483.019154, 0.200000 s)

Frame 8 (1639143483.261158, 0.200000 s)

Frame 9 (1639143483.513150, 0.200000 s)

The **CLOG.IDX** files generated beside the CLOGs. Contains the simple binary array of little-endian qword addresses of the "F" at each record start.

.clog.idx example:

Pointers to records at 0, 0x29, 0x52, 0x7b, 0xA4, 0xCD, ...

	0001	0203	0405	0607	0809	0A0B	0C0D	0E0F
000	0000	0000	0000	0000	2900	0000	0000	0000
010	5200	0000	0000	0000	7B00	0000	0000	0000
020	A400	0000	0000	0000	CD00	0000	0000	0000
030	F600	0000	0000	0000	3401	0000	0000	0000

6.3.1. File extensions and flags: T3PA details

The T3PA is timepix3 data file in text format with lines and tabs. Contains the information line and data lines with record index, pixel index, Time of arrival, Time over threshold, Fine ToA and Overflow. Example:

Index	Matrix	Index	ToA	ToT	FToA	Overflow
0	1028	1918 14	22	0		
1	1028	3126 8	28	0		
2	1028	3778 5	23	0		
...						
156003		39793 98473646054	38	9	0	
156004		190 98492090610	19	3	0	

The **Index** is simple index of measurement line. This growing while measurement is running. If you append new measurement to existing file, new index is 0 again and again growing while new measurement is running.

The **Matrix Index** is index of the pixel. On the Minipix Tpx3 is 0 at the left-down, see image:

The **ToA** is time of arrival in units 25 ns, mod by limit, for example Minipix 2^{64} (14600y), Advapix-single 2^{30} (26s), Advapix-Quad 2^{28} (6.5s).

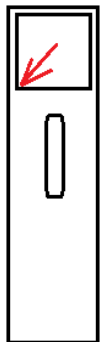
Note: The ToA on-chip implementation in the pixels is limited to 14 bits (409.6 μ s).

The ToA in T3PA is extended by device. But there is inherent uncertainty around the borders.

These values may be incorrectly assigned. Users not comfortable with our extension can apply AND with (uint64)16383 to extended ToA to get original ToA from the chip.

The **ToT** is time over threshold in units 25 ns.

The **FToA** stands for "fine ToA" and it is the finest step of the ToA measurement. To properly account for this step in the conversion of ToA to time, it is necessary to subtract the amount of counts of fToA in the following manner: **Time [ns] = 25*ToA - (25/16)*fToA**. The original range of this fToA value in the chip is 4 bits, or 16 values. This is extended in the post-processing of the data into 5 bits, or 32 values to include a correction for the delay of the clock propagation in the chip. The final value exported into t3pa files has a range of 5 bits, or 32 values, but the previous equation still stands.



The **Overflow** is sign of data transfer overflow. If the line has this 1:

index = 0x74: start of lost data

index = 0x75: end of lost data, toa is length of the missing time

(this can occurs with rates over megahits per seconds for Minipix)

Note: In data from **multichip devices**, there is not Overflow, replaced by **Chip index**.

6.3.2. File extensions and flags: Pixel matrix configuration files

Overview:

bpc	Binary Pixel Configuration	All PM config in one file, meaning of the bits depends on the chip.
txt	Ascii Mask Matrix	Text file with pixel mask
txt	Ascii Test Bit Matrix	Text file with test bits
txt	Ascii THL adj. bits Matrix	Text file with threshold values adjustment

```
QString BPC = "Binary Pixel Configuration (*.bpc)";
QString ASCII_MASK = "Ascii Mask Matrix (*.txt)";
QString ASCII_TEST = "Ascii Test Bit Matrix (*.txt)";
QString ASCII_THL = "Ascii THL adj. bits Matrix (*.txt)";
```

6.4. Constants: Acquisition types and modes, incl. triggers

These constants can be used with the **doAdvancedAcquisition** method, for example:

```
# acquisition for 2 seconds in data-driven mode
# without special triggering features, store data in the memory
dev.doAdvancedAcquisition(1, 2, pixet.PX_ACQTYPE_DATADRIVEN,
                           pixet.PX_ACQMODE_NORMAL, pixet.PX_FTYPE_NONE, 0, "")
```

Acquisition types:

```
PX_ACQTYPE_FRAMES
PX_ACQTYPE_TESTPULSES
PX_ACQTYPE_DATADRIVEN
```

Acquisition modes:

```
PX_ACQMODE_NORMAL
PX_ACQMODE_TRG_NO
PX_ACQMODE_TRG_HWSTART
PX_ACQMODE_TRG_HWSTOP
PX_ACQMODE_TRG_HWSTARTSTOP
PX_ACQMODE_TRG_SWSTART
PX_ACQMODE_COMPRESSED
PX_ACQMODE_TDI
PX_ACQMODE_CONTINUOUS
```

Modes description:

```
Start immediately on doAcq... method
Start immediately on doAcq... method
Start on trig HW signal edge
Stop on trig HW signal edge
Start and stop on trig HW signal edge
Start on software trigger
Save compressed stream
Time delayed integration mode
Continuous mode: Repeat acqs until abort
```

Note: Using DATADRIVEN option with frame-only devices can cause strange behavior without error. It is not guaranteed that the result will be the same with a different software/firmware version.

6.5. Constants: Frame types

Frames from frame acq.

```
PX_FRAMETYPE_I16
PX_FRAMETYPE_U32
PX_FRAMETYPE_DOUBLE
PX_FRAMETYPE_U64
```

Frames from pixels preview

```
PX_TPX3_FRAMETYPE_SUM
PX_TPX3_FRAMETYPE_FADE
```

6.6. Constants: Settings types

```
PX_STGTYPE_INI
PX_STGTYPE_XML
PX_STGTYPE_MCF
PX_STGTYPE_H5
PX_STGTYPE_BIN
PX_STGTYPE_EXT
```

6.7. Constants: Events

These constants can be used to register, use in callbacks and unregister various events. Example:

```
# register event "Acq finished" to use function "callbackFin"
dev.registerEvent(pixet.PX_EVENT_ACQ_FINISHED, 0, callbackFin)
```

Events used in acquisitions:

PX_EVENT_ACQ_MEAS_STARTED	Measurement was started (0 if start OK / error code)
PX_EVENT_ACQ_MEAS_FINISHED	Measurement was finished (0 if success / error code)
PX_EVENT_ACQ_SERIE_STARTED	Series of acquisitions was started (count of finished series =0)
PX_EVENT_ACQ_SERIE_FINISHED	Series of acquisitions was finished (count of finished series =1)
PX_EVENT_ACQ_FINISHED	One acquisition was finished (count of finished acqs) (<i>frame mode</i>)
PX_EVENT_ACQ_NEW_DATA	New data arrival (actual acq index) (<i>data-driven mode</i>)
PX_EVENT_ACQ_ABORTING	Aborting of acquisition is in progress
PX_EVENT_ACQ_ABORTED	Acquisition was aborted
PX_EVENT_ACQ_FAILED	Acquisition was failed (error code from the Pixet core)
PX_EVENT_ACQ_SWTRG_READY	Software trigger is ready

Typical order of events in the measurement process:

0. start of execution of a method like as do...Acquisition
1. ACQ_MEAS_STARTED (0)
2. ACQ_SERIE_STARTED (0)
3. DEV_STATUS_CHANGED (0)
- 4f. ACQ_FINISHED (1), ACQ_FINISHED (2), ACQ_FINISHED (3), ... (frame mode)
- 4d. ACQ_NEW_DATA (0), ACQ_NEW_DATA (0), ACQ_NEW_DATA (0), ... (data-driven mode)
5. ACQ_FINISHED (all frames count in frames / 1 in data-driven)
6. ACQ_SERIE_FINISHED (1)
7. ACQ_MEAS_FINISHED (0)
8. execution of acquisition method ends (if doContinuousConversion was used, this happened between 1 and 2)

Other events list:

PX_EVENT_DEV_CFG_CHANGED	PX_EVENT_MENU_CHANGED
PX_EVENT_DACs_CHANGED	PX_EVENT_NEW_DEVICE
PX_EVENT_BIAS_CHANGED	PX_EVENT_DEVICE_REMOVED
PX_EVENT_CFG_SAVING_PROGRESS	PX_EVENT_FILTERS_CHANGED
PX_EVENT_CFG_LOADING_PROGRESS	PX_EVENT_EXIT
PX_EVENT_PIXCFG_CHANGED	PX_EVENT_TPX2CLOCKS_CHANGED
PX_EVENT_TPX2CLOCK_CHANGED	PX_EVENT_TPX2STG_CHANGED
PX_EVENT_MPX3OPM_CHANGED	PX_EVENT_MOTOR_STATUS_CHANGED
PX_EVENT_TPX3STG_CHANGED	
PX_EVENT_LOCK_CHANGED	
PX_EVENT_DEV_MENU_CHANGED	
PX_EVENT_DEV_STATUS_CHANGED	

6.8. Constants: Data formats

```
PX_DATAFORMAT_NONE
PX_DATAFORMAT_FRAME
PX_DATAFORMAT_COMP_STREAM
PX_DATAFORMAT_TPX3_PIXELS
PX_DATAFORMAT_TPX_PIXELS
PX_DATAFORMAT_MULTIFRAME
PX_DATAFORMAT_TPX3_CLUSTERS
PX_DATAFORMAT_EXTERNAL
```

6.9. Constants: Threshold flags

See [Threshold settings](#)

PX_THLFLG_NONE	Raw DAC value.
PX_THLFLG_RELATIVE	Relative THL for more chips. Ref can be set by SET_RELATIVE.
PX_THLFLG_ENERGY	Absolute THL value in keV.
PX_THLFLG_RELATIVE_CURRENT	Relative THL to current value. Useful to increments etc.
PX_THLFLG_SET_RELATIVE	Only set the reference THL value for RELATIVE.

6.10. Constants: Pixels matrix config types

See [The pixels matrix configuration objects](#)

PX_PIXCFG_MXR	PX_PIXCFG_MASK
PX_PIXCFG_TPX	PX_PIXCFG_TEST
PX_PIXCFG_MPX3	PX_PIXCFG_THL
PX_PIXCFG_TPX3	PX_PIXCFG_THH
PX_PIXCFG_TPX2	PX_PIXCFG_TPXMODE
PX_PIXCFG_ALL	PX_PIXCFG_GAIN

6.11. Constants: Other

PX_CHIP_ALL	Chip index for indexing all the device chips. Applicable to setDac, setThreshold, ...
-------------	---

PX_TPX3_VCO_AUTO	Timepix3 superpixel FToA oscillator running if need.
PX_TPX3_VCO_ENABLED	Timepix3 superpixel FToA oscillator is forced enabled.
PX_TPX3_VCO_DISABLED	Timepix3 superpixel FToA oscillator is forced disabled.

PX_TPX3_GRAYCNT_AUTO	Timepix3 ToA Gray counter running if need.
PX_TPX3_GRAYCNT_ENABLED	Timepix3 ToA Gray counter is forced enabled.
PX_TPX3_GRAYCNT_DISABLED	Timepix3 ToA Gray counter is forced disabled.

See: **Sensor device objects – Methods: Other/Special – [Special](#)**

6.12. Constants: Mpx3 settings

Counter selection:

PX_MPX3_CNT_LOW
PX_MPX3_CNT_HIGH
PX_MPX3_CNT_BOTH

Counter depth*:

PX_MPX3_CNTD_1B
PX_MPX3_CNTD_6B
PX_MPX3_CNTD_12B
PX_MPX3_CNTD_24B
PX_MPX3_CNTD_COUNT

*Only 24b/not-24b is actually used

Thresholds:

PX_MPX3_TH0
PX_MPX3_TH1

Col. blocks:

PX_MPX3_COLB_ALL
PX_MPX3_COLB_0_31
PX_MPX3_COLB_32_63
PX_MPX3_COLB_64_95
PX_MPX3_COLB_96_127
PX_MPX3_COLB_128_159
PX_MPX3_COLB_160_191
PX_MPX3_COLB_192_223
PX_MPX3_COLB_224_255
PX_MPX3_COLB_0_63
PX_MPX3_COLB_64_127
PX_MPX3_COLB_128_191
PX_MPX3_COLB_192_255
PX_MPX3_COLB_0_127
PX_MPX3_COLB_128_255

Row blocks:

PX_MPX3_ROW_B_ALL
PX_MPX3_ROW_B_1
PX_MPX3_ROW_B_2
PX_MPX3_ROW_B_4
PX_MPX3_ROW_B_8
PX_MPX3_ROW_B_16
PX_MPX3_ROW_B_32
PX_MPX3_ROW_B_64
PX_MPX3_ROW_B_128

DAC indexes (some are reserved and do nothing):

Warning: Be carefull. Change DAC settings:

- will cause lost of calibration
- can cause some unexpected and strange problems
- can cause device damage

PX_MPX3_TH0
PX_MPX3_TH1
PX_MPX3_TH2
PX_MPX3_TH3
PX_MPX3_TH4
PX_MPX3_TH5
PX_MPX3_TH6
PX_MPX3_TH7
PX_MPX3_PREAMP
PX_MPX3_IKRUM
PX_MPX3_SHAPER
PX_MPX3_DISC
PX_MPX3_DISC_LS
PX_MPX3_SHAPER_TEST
PX_MPX3_DAC_DISC_L
PX_MPX3_DAC_TEST
PX_MPX3_DAC_DISC_H
PX_MPX3_DELAY
PX_MPX3_TP_BUFF_IN
PX_MPX3_TP_BUFF_OUT
PX_MPX3_RPZ
PX_MPX3_GND
PX_MPX3_TP_REF
PX_MPX3_FBK
PX_MPX3_CAS
PX_MPX3_TP_REFA
PX_MPX3_TP_REFB

See [Medipix3](#)

6.13. Similar related to previous lists

There is not pixet constants, but it is similar lists.

6.13.1.1. Data types

This is the numbers used to determine the datatype of the value. This can be copied to the user code.

DT_CHAR	= 0	DT_FLOAT	= 8
DT_BYTE	= 1	DT_DOUBLE	= 9
DT_I16	= 2	DT_BOOL	= 10
DT_U16	= 3	DT_STRING	= 11
DT_I32	= 4		
DT_U32	= 5		
DT_I64	= 6		
DT_U64	= 7		

Reversed List:

```
["Chr", "Byte", "i16", "u16", "i32", "u32", "i64", "u64", "Flo", "Dou", "Bool", "Str"]
```

6.13.1.2. Named parameters

Devices have named parameters. See [Parameters names lists by device types](#)

6.13.1.3. Error codes list

```
#define PXERR_NO_ERROR          ( 0)
#define PXERR_FRAME_NOT_FOUND  (-1000)
#define PXERR_EVENT_EXIST      (-1001)
#define PXERR_EVENT_NOEXIST    (-1002)
#define PXERR_EVENT_FUNC_NOEXIST (-1003)
#define PXERR_BUFFER_SMALL     (-1004)
#define PXERR_INVALID_ARGUMENT (-1005)
#define PXERR_FRAME_EMPTY      (-1006)
#define PXERR_ABORTED          (-1007)
#define PXERR_ACQ_FAILED        (-1008)
#define PXERR_READ_DATA_FAILED (-1009)
#define PXERR_FILE_OPEN         (-1010)
#define PXERR_FILE_READ         (-1011)
#define PXERR_FILE_WRITE        (-1012)
#define PXERR_FILE_BADDATA      (-1013)
#define PXERR_FILE_SEEK         (-1014)
#define PXERR_MEMORY_ALLOC      (-1015)
#define PXERR_ITEM_NOT_FOUND    (-1016)
#define PXERR_INVALID_DATA_TYPE (-1017)
#define PXERR_UNEXPECTED        (-1018)
#define PXERR_FRAME_LOCKED      (-1019)
```

```
#define PXERR_PLUGIN_EXISTS      ( -1020)
#define PXERR_PLUGIN_INITERROR  ( -1021)
#define PXERR_MENUITEM_EXISTS   ( -1022)
#define PXERR_CANNOT_CONVERT    ( -1023)
#define PXERR_LOCK_TIMEOUT      ( -1024)
#define PXERR_NOT_LOCK_OWNER    ( -1025)
#define PXERR_UNSUPPORTED       ( -1026)
#define PXERR_NOT_FOUND         ( -1027)
#define PXERR_EXISTS            ( -1028)
#define PXERR_REFCOUNT_NONZERO  ( -1029)
#define PXERR_DEVICE_ERROR      ( -1030)
#define PXERR_NO_ITEMS          ( -1031)
#define PXERR_NO_MORE_ITEMS     ( -1032)
#define PXERR_INVALID_FILE_TYPE ( -1033)
#define PXERR_AUTODETECT_FAILED ( -1034)
#define PXERR_BUFFER_FULL       ( -1035)
#define PXERR_ACQ_RUNNING       ( -1036)
#define PXERR_TEMPERATURE_ERROR ( -1037)
#define PXERR_INVALID_DATA      ( -1038)

#define PXERR_ACQFAIL_ERRORS_START ( -5000)
#define PXERR_ACQFAIL_COMMUNICATION ( -5001)
#define PXERR_ACQFAIL_TEMPERATURE_ERROR ( -5002)
#define PXERR_ACQFAIL_SAVING_DATA ( -5003)
#define PXERR_ACQFAIL_ERRORS_END ( -5999)
```

6.13.1.4. Clustering constants and error codes

See [Clustering constants and error codes](#)

6.13.1.5. Spectraimg constants and error codes

See [Spectraimg constants and error codes](#)

7. Sensor device objects

Objects in this chapter: **IDevTpx3** (Timepix3), **IDevMpx2** (Medipix2 = Timepix), **IDevMpx3** (Medipix3).

This chapter is **by default about the IDevTpx3**.

(see notes for other devices [Other sensor devices](#))

Example:

```
pixet=pypixet.pixet
dev = pixet.devices()[0]
```

The dev is IDevTpx3 object created by pixet's device() method or other his device...() methods if connected device with index 0 have the Timepix3 chip.

7.1. Methods: Device info

deviceType()	Returns type of the device (see Constants: Device types)
fullName()	Returns full name of the device. Example print: MiniPIX I08-W0060
deviceID()	Returns device ID. This is same as first chip serial number. Example print: I08-W0060
deviceInfo(devInfo)	Sets device informations to the DevInfo object created by the pixet object. See Pixet: Methods: Devices management , Constants: DevInfo types

Note: Methods described above have all devices, not only sensors.

chipIDs()	Returns array of all installed chip IDs. Example print: ['I08-W0060']
chipIDSummary()	Returns chip ID summary. This is same as first chip serial number. Example: I08-W0060
chipLayout(w, h, o, a)	Gets the chips layout/order informations. See example bellow. w, h: pointers to store width and height in chip count as units. o, a: pointers to store chip orders and chip angles lists.
supportedAcqModes()	Returns OR of all acq. modes (see Constants: Acquisition types and modes, incl. triggers).
supportedAcqTypes()	Returns OR of all acq. types (see Constants: Acquisition types and modes, incl. triggers).
supportedDataFormats(list)	Sets list of the supported data formats, returns 0 if OK (see Constants: Data formats).
acqTimeMin()	Returns shortest allowed acquisition time in seconds.
acqTimeMax()	Returns longest allowed acquisition time in seconds.
sensorThickness(idx)	Returns sensor chip thickness in micrometers of the indexed chip.
sensorPitch(idx)	Returns pixels pitch in micrometers of the indexed chip.
sensorType(idx)	Returns sensor type string of the indexed chip. Example print: CdTe (Multi-devices: some returns "Si" on all slaves)
width()	Returns width of the sensor in pixels.
height()	Returns height of the sensor in pixels.
pixelCount()	Returns count of pixels in the sensor (width*height).
chipType()	Returns index of chip type (see Constants: Device and chip types).
chipCount()	Returns number of sensor chips in the device.
adcCount()	Returns count of ADCs available in the device.
adcValue(idx)	Returns value from indexed ADC.

```
w = [int(0)] # The dev.chipLayout example, chipAngles working same
h = [int(0)]
o = [int(0)] * 4
a = [int(0)] * 4
rc = dev.chipLayout(w, h, o, a)
print("RC, Width [chips], Height [chips], Chips order, angles:" rc, w[0], h[0], o, a)
```

7.2. Methods: Device config

Sensor devices have lot of settings, the **values of which are necessary for proper function**. These settings are stored in the **XML config file**. The Pixet core have mechanism for **automatic loading and saving** the config files. Files contains tree with detector settings, matrix configuration, including masked pixels, chip order of multichip devices, calibration data and other settings. Programmer can use the user settings, it stored here too.

When the Pixet core starting, it initializing devices and searching for config file for each device in the **“configs” directory** (For example, If device is Minipix with chip s.n. K05-W0103, search name is “MiniPIX-K05-W0103.xml”). If file found, Pixet core loads it. If not, next search is in the **“factory” dirrecotry**.

When the Pixet core exiting, it closing all devices and automatically saving actual settings of each device to the “configs” directory, including overwriting of last file (see [The factory and the configs directories](#)).

Programmer can invoke loading or saving settings by next described methods.

7.2.1. General loading and saving settings

loadFactoryConfig()	Loads factory default config from the “factory” directory.
defaultConfigFileName()	Returns default full path for device config file.
loadConfigFromSettings(set)	Loads config from the ISettings object. See The ISetting object
saveConfigToSettings(set)	Saves config to the ISettings object. See The ISetting object
loadConfigFromFile(path)	Loads config from the specified file.
saveConfigToFile(path)	Saves actual config to the specified file.
isConfigInDeviceSupported()	Returns 1 if store of configuration data in the device is supported or 0 if not.
hasConfigInDevice()	Returns 1 if configuration data are stored in the device.
loadConfigFromDevice()	Loads config stored in the device.
saveConfigToDevice()	Stores actual config in the device (use only with certainty of the correct config)
hasDefaultConfig()	Returns 1 if the pixCfg have 0 in all maskBit, testBit and thl items, or 0 if not. This means, if 1 returned, the device has no config (but 0 not means proper config).
userSettings()	Returns the ISettings object to access the user settings . See The ISetting object

7.3. Methods: General

lastError()	Returns last error msg. (useful with return codes testing or ACQ_FAILED callback)
deviceStatusText()	Returns device status text
isConnected()	Returns 1 if device is connected or 0 if not
reconnect()	Reinitializes the device, same as plug/unplug the cable As with power-up/init, even now unexpected charges may appear in the chip.
isBusy()	Returns 1 if device is busy or 0 if not. Note: IDevTpx3 allways returns 0.
isDeviceLocked()	Returns 1 if device is locked or 0 if not
lockDevice()	Sets device as locked
unlockDevice(bool force)	Sets device as unlocked. Use force=true to force unlock.

parameters()	Returns the IParamMgr object (see Accessing the named parameters)
asIDev()	Returns the IDev object. This can be used to connect the device to other objects, pypxproc for example (see The pypxproc object).

Note: Methods described above have all devices, not only sensors.

operationMode()	Returns actual device operation mode * (see Constants: Operation modes)
setOperationMode(opm)	Sets the device operation mode * (see Constants: Operation modes) * If device haven't this, use pixcfg. setModeAll . (see The pixels matrix configuration objects)

enableInterpolationOfMaskedPixels(bool enable, flags)	Enables/disables interpolation of masked pixels. Use flags 0 or PX_INTERPOL_FLAG_BAD_COLUMNS to enable interp. of columns.
isInterpolationOfMaskedPixelsEnabled()	Returns 1 if interpolation of masked pixels is enabled or 0 if not.
setConvertToaTime(bool convert)	If enabled, in ToA subframes will be counter and fine counter converted to the nanosecs, if not, two separate subframes will be created:

1. ToA named, with ToA counter raw values
2. FTOA named, with fine ToA counter raw values

(see [Frame access and related methods](#))

isConvertToaTimeEnabled()	Returns 1 if ToA convert is enabled or 0 if not.
setCalibrationDataAbct(chipIdx, dataa, datab, datac, datat)	Associates calibration arrays to indexed chip.
calibrationDataAbct(chipIdx, dataa, datab, datac, datat)	Reads calibration data of indexed chip to the arrays.
useCalibration(bool enable)	Enable/disable ToT calibration (convert to energy) of a data in frame mode.
isUsingCalibration()	Returns 1 if calibration of a frame data is enabled or 0 if not.
hasCalibration()	Returns 1 if device has calibration (default after manuf. expedition) or 0 if not.
calibrateFrame(frame)	Apply calibration to the frame. (Tpx3 only)
calibratePixelsAndFilter(pixels, timeWalkEn, eneMin, eneMax)	Calibrate and filter pixels. (Tpx3 only, add 2023)

Pixels: See: [The ITpx3Pixels and other pixels objects](#)

timeWalkEn: Use timewalk correction (signe chip devices only)

eneMin, eneMax: Energy range for filtering. Use <=0 to disable.

Output: [[idx], [toa], [tot], OutputPxCount>0 / 0 / errCode<0]

Using: `pixelsCalib = dev.calibratePixels(pixels, timeWalkEn, eneMin, eneMax)`

`len = pixelsCalib[3]`

pixCfg()	Returns the pixel matrix config object (see The pixels matrix configuration objects)
createCopyOfPixCfg()	Returns copy of the pixel matrix object. Useful to backup the pixCfg settings.
refreshPixelCfg()	Sets the pixel matrix configuration from the pixCfg memory to the device.
setPixCfg(pixcfg)	Sets the pixel matrix configuration from the used pixCfg object to the device.

7.4. Methods: Acquisition and related

7.4.1. Simple and advanced acquisition

Note: Unexpected charges may appear in the chip after power-up/initialization. The first measurement, especially Tpx3 data-driven, often generates meaningless data for a few seconds. Use a dummy acquisition before the first real measurement. It can often be enough to simply wait, which is why it rarely occurs in GUI.

doSimpleAcquisition(count, time, fileType, fileName) Acquire desired number of frames in normal/frame mode, to be used each separately.

doSimpleIntegralAcquisition(count, time, fileType, fileName) Acquire desired number of frames in normal/frame mode and integrate all to single frame.

doAdvancedAcquisition(count, time, acqType, acqMode, fileType, fileFlags, fileName)

Frame mode: Acquire desired number of frames to be used each separately.

Data-driven mode: Acquire pixels for desired time.

Support acq. type and mode set.

doAdvancedIntegralAcquisition(count, time, acqType, acqMode, fileType, fileFlags, fileName)

count	Number of acquisitions to do (in data-driven mode exact value ignored, only must be over 0)
time	Each acquisition time [sec]. 0 can be used for infinite in data-driven mode.
fileType	Type of file to be saved, or <code>pixet.PX_FTYPE_NONE</code> to only store data in memory Using <code>FTYPE_NONE</code> along with a valid file name often works, but can lead to unexpected results
fileName	Filename if files used or "" if not. See Constants: File types, extensions and flags
fileFlags	Flags for file saving. See Constants: File types, extensions and flags
acqType	Acquisition type (frames / test pulses / data-driven)
acqMode	Acquisition mode (normal / triggered / ...) See Constants: Acquisition types and modes, incl. triggers

The above methods start the measurement and wait for it's end. If file specified, this save single or more files. Depends on operation mode, more files can be created from each acquisition.

Example:

```
import pypixet

print("pixet core init...")
pypixet.start()
pixet=pypixet.pixet
devices = pixet.devicesByType(pixet.PX_DEVTYPE_TPX3)
dev = devices[0]
dev.setOperationMode(pixet.PX_TPX3_OPM_EVENT_ITOT)

print("dev.doSimpleAcquisition (2 frames per 1 sec) - start")
rc = dev.doSimpleAcquisition(2, 1, pixet.PX_FTYPE_AUTODETECT, "test.png")
print("dev.doSimpleAcquisition - end:", rc)

pixet.exitPixet()
```

This example save files: test_0_Event.png, test_0_iToT.png, test_1_Event.png, test_1_iToT.png

If filename "" specified, or fileType NONE, data is stored only in memory. After each acquisition, data can be read via **lastAcqFrameRefInc()** in frame mode, or via **lastAcqPixelsRefInc()** in data-driven mode. This can be used after end, if count is 1, or via callbacks. Event **ACQ_FINISHED** in frame mode or **ACQ_NEW_DATA** data-driven mode.

To use events with callback, must use **registerEvent** device method before start of measurement. After process ends, can be used **unregisterEvent** to stop this (see [Methods: Events](#)). Even more events can be used in connection with the measurement (see details in [Constants: Events](#)).

Example: # first is same code as in the previous example, up to dev.setOperat...

```
def clbACQ_FINISHED(cnt):
    print("ACQ_FINISHED", cnt)
    frame = dev.lastAcqFrameRefInc()
    data = frame.data()
    print("  Px val min:", min(data), ", max:", max(data))
    frame.destroy()

dev.registerEvent(pixet.PX_EVENT_ACQ_FINISHED, 0, clbACQ_FINISHED)

print("dev.doSimpleAcquisition (5 frames per 1 sec) - start")
rc = dev.doSimpleAcquisition(5, 1, pixet.PX_FTYPE_NONE, "")
print("dev.doSimpleAcquisition - end:", rc)
```

This example acquires 5 frames and display minimum and maximum pixel value from each frame.

Note: Most device types have subframes and frame.data() contains only processing artefacts. See [Frame access and related methods](#) and [Frame with subframes example](#)

7.4.2. Continuous acquisition

There are three features:

1. Endless measuring. Number of repeats is not limited.
2. Can do true continuous measuring. If the device has the Continuous feature, measuring time is longer than data transfer time and other setting not colliding with it, sets of counters alternate, one sending data while the other measures.
3. Working in separate thread. The function terminates immediately.

doContinuousAcquisition(buffCount, time, acqMode)

buffCount Number of buffers. After number of acquired frames reach buffCount, event DEV_STATUS_CHANGED occurs, frame count resets and start overwriting of previous frames stored in the memory.

This method is "callbacks-only". Unlike the previously described methods, this terminates immediately after the "ACQ_MEAS_STARTED" event. The process continues to run in the background and generating callbacks. In this time, program can do anything else or simply waiting, like as input("Press enter to end").

Warnings:

Currently, for devices **other than Mpx3, this feature is somewhat experimental**. For some devices, the acqMode parameter value has no effect. For future compatibility use the `pixet.PX_ACQMODE_CONTINUOUS`. On Tpx3 abort may not working and this process cannot be terminated correctly.

7.4.3. Acquisition status

isAcquisitionRunning()	Returns 1 if acq. is now running or 0 if not
isAcquisitionAborting()	Returns 1 if now aborting of a acq. or 0 if not
abortOperation()	Aborts a running operation. Can be used to stop the ContinuousAcquisition.
getShutterStartTime()	Returns shutter start time. Without HW trigger is always =0.
getShutterEndTime()	Returns shutter end time. Without HW trigger is always =acqTime.

7.4.4. Triggers

Sensor devices can start measurement immediately on command execution or can wait for trigger. This can be enabled using `acqMode` option in `doAdvancedAcquisition`, `doAdvancedIntegralAcquisition` or `doContinuousAcquisition`.

`doAdvancedAcquisition(count, time, acqType, acqMode, fileType, fileFlags, fileName)`

For example, if the `pixet.PX_ACQMODE_TRG_HWSTART` used as `acqMode`, starting of acquisition waits for hardware trigger signal. (see [Constants: Acquisition types and modes, incl. triggers](#))

7.4.4.1. Hardware trigger

The hardware trigger can be from external source to synchronize with a particle accelerator bunches, scanning motors for CT-like applications, etc, or from interconnecting signals within parts of a multi-device like as Widedix-L with two eth. cables. If programmer want using HW trigger, must set-up low level device parameters related to the HW trigger. See the **Synchronization guide** for the device.

Example for Advapix-Tpx3 external sync:

Note: Advapix has several subversions with and they have slightly different sets of parameters. See **Synchronization guide** of Your device.

```
pars = dev.parameters()
print("IsMaster .getBOOL", pars.get("IsMaster").getBOOL())
print("Device for synchronizing must be slave")
rc = pars.get("TriggerStg").setByte(3)
print("TriggerStg .setByte", rc)
rc = pars.get("TrgMulti").setBOOL(False)
print("TrgMulti .setBOOL", rc)
rc = pars.get("TrgT0SyncReset").setBOOL(False)
print("TrgT0SyncReset .setBOOL", rc)
rc = pars.get("TrgTimestamp").setBOOL(False)
print("TrgTimestamp .setBOOL", rc)
rc = pars.get("TrgReady").setBOOL(True)
print("TrgReady .setBOOL", rc)
rc = pars.get("TrgCmos").setBOOL(False)
print("TrgCmos .setBOOL", rc)
```

(see [Synchronizing multi-device Mpx3 instruments](#) for Widedix-L Mpx3)

7.4.4.2. Software trigger

isReadyForSoftwareTrigger() Returns 1 if device is ready to start acq. by the software trigger or 0 if not
doSoftwareTrigger(flags) Do the the software trigger now. Flags is reserved for future use, recommended 0

This can be used in acquisition mode **PX_ACQMODE_TRG_SWSTART** (other TRG_SW... modes are reserved for future use). See [Constants: Acquisition types and modes, incl. triggers](#)

After use start acquisition method in this mode, event occurs **PX_EVENT_ACQ_SWTRG_READY** and sw. trigger is ready. From this time, acquisition starts immediately with `dev.doSoftwareTrigger` used. Note: "immediately" in Python, understand with exaggeration.

7.4.5. Online data processing, events and callbacks

Online data processing is possible by two ways:

- A) Do acquisition(s) without files and process the data after it ends
- B) Do acquisition(s) without files and process the data in callback functions while acq. running
 - In frame mode, the **PX_EVENT_ACQ_FINISHED** occurs after each frame.
 - In data-driven mode, the **PX_EVENT_ACQ_NEW_DATA** occurs after some pixels arrival. Maximum event rate is approximately twice per second. Maximum pixel rate is about 1.4 million pixels per event, depending on the speed of the computer, condition of the data path and other circumstances.

acqDataCount() Returns number of data blocks or frames waiting in memory (both modes)

acqFrameCount() Returns number of frames waiting in memory (frame mode)

acqFrameRefInc(idx) Gets the frame object from indexed acquisition (see [The frame object](#))

acqDataRefInc(idx) Gets the pixels or frame object from indexed acquisition

lastAcqFrameRefInc() Gets the frame object from last acquisition (see [The frame object](#)) (frame mode)

lastAcqPixelsRefInc() Gets the pixels object from last acquisition (see [The pixels object](#)) (data-driven mode)

lastAcqDataRefInc() Gets a universal object, can contains pixels or frame, depending on OPM.

7.4.6. Events: more details

registerEvent (eventName, reserved, callbackFun) Register general event to use the callback function

unregisterEvent (eventName, reserved, callbackFun) Unregister general event to stop using callback

eventName See [Constants: Events](#)

reserved Reserved to future use, now it doesn't matter the value or the type, recommend use 0.

callbackFun Name of the function that can be called by the event.

registerBeforeSaveDataEvent (reserved, callbackFun)

unregisterBeforeSaveDataEvent (reserved, callbackFun)

Register/unregister the “before save” event to use (stop using) callbackFun. This is special event, which occurs while measurements with file-saving is running, before start saving each file. Parameter of callback function is frame object with data to be saved. It can be used for add attributes or preprocessing. In data-driven mode, this event occurs very often. Be carefully for CPU load if use it.

7.5. Methods: Other/Special

7.5.1. Sensor refresh

The sensor refresh is used to clean the sensor of free charges. Process containing sequence of bias changes. It was designed primary for the CdTe chips.

isSensorRefreshSupported()	Returns 1 if the device support sensor refresh process or 0 if not.
isSensorRefreshEnabled()	Returns 1 if the sensor refresh process enabled on the device or 0 if not.
enableSensorRefresh(en, time)	Enable/disable automatic sensor refresh. time: 0 = refresh before every meas. start / >0 refresh every time seconds.
doSensorRefresh()	Do the sensor refresh now.
sensorRefresh()	Returns the sensor refresh sequence text.
setSensorRefresh(text)	Sets the sensor refresh sequence text. Suitable values depend on chip manufacturing technology details.

The refresh string defines steps with pairs of times [sec] and bias coefficients [1=100%]. Physical bias values limited to min/max chip properties, values >1 has same effect as =1.

Example: `# ("time1, coef1; time2, coef2; time3, coef3; ...")`
 `dev.sensorRefreshString("0.5,0; 0.75,1.4; 0.43,0.96; 0.4,1")`

7.5.2. Threshold settings

threshold(ch, flags) Returns the threshold value from indexed chip (flags: see [Constants: Threshold flags](#))
setThreshold(ch, val, flags) Sets the threshold value on indexed chip (flags: see [Constants: Threshold flags](#))

setMinThresholdNoise(ch, thresholdKev)	Sets threshold energy of bound of noising on the chip.
isThresholdCalibrated(ch)	Returns 1 if the chip calibrated threshold or 0 if not.
setThresholdCalibCoeffs(ch, gainMode, bitDepth, double* calibParams, size)	Sets threshold calib coefficients for the specified parameters.
thresholdCalibCoeffs(ch, gainMode, bitDepth, double* calibParams, size)	Gets threshold calib coefficients for the specified parameters.

Note: Some device types have individual pixel thresholds. See [Thresholds management](#)
 The **Medipix3** has additional second parameter after chip index – thIdx = threshold index. See [Medipix3](#)
 Example: Tpx3: **threshold(ch, flags)** / Mpx3: **threshold(ch, thIdx, flags)** – thIdx: 0/1 or 0-7 in color mode

7.5.3. Bias settings

bias()	Returns actual bias set value.
biasMin()	Returns minimal allowed bias value.
biasMax()	Returns maximal allowed bias value.
setBias(voltage)	Sets the bias voltage.
isPolarityPositive(idx)	Returns 1 if polarity of the bias voltage on indexed chip is positive or 0 if not.
IsBiasSenseSupported()	Returns 1 if device support bias sensing or 0 if not.
biasVoltageSense()	Returns actual measured bias voltage in V.
biasCurrentSense()	Returns actual measured bias current in μ A.

7.5.4. Digital test

doDigitalTest() Do the digital test. Writes the test data to the device, read and verify. Returns return code, good pixels count, bad pixels count and frame with it's map (1 bad / 0 good).

```
Example: rc, good, bad, frame = dev.doDigitalTest()
if rc==0:
    print("good pixels", good, "bad pixels", bad)
else:
    print("Digital test failed")
```

7.5.5. DACs control

Many sensor devices contain digital-to-analog converters. These are used to set some internally used voltages and currents. Refer to the device/chip datasheets for details on their meaning.

Warning: This is very low-level thing, direct driving of DACs is not needed for normal device using. Using the wrong settings can lead to strange behaviour, sometimes even damaging the device.

dacs()	Returns the IMpxDacs object to control the devices DACs. See The IMpxDacs object
createCopyOfDacs()	Returns the IMpxDacs object to store all the device DACs settings.
setDacs(dacs)	Sets all the device DACs settings by used IMpxDacs object.
refreshDacs()	Sets all the device DACs settings to the DACs outputs.

7.5.6. Special

VCO()	Returns the Timepix3 superpixel FTOA oscillator enable setting value.
setVCO(en)	Sets the Timepix3 superpixel FTOA oscillator enable setting value. piket.PX_TPX3_VCO_... – AUTO/ENABLE/DISABLE – See: Constants: Other

grayCounter()	Returns the Timepix3 gray counter enable setting value.
setGrayCounter(en)	Sets the Timepix3 gray counter enable setting value. piket.PX_TPX3_GRAYCNT_... – AUTO/ENABLE/DISABLE – See: Constants: Other

timeWalkCorrCoeffs(ch, c, d)	Copies timewalk coefficients of the indexed chip to list-type variables.
setTimeWalkCorrCoeffs(ch, c, d)	Sets timewalk coefficients of the indexed chip. This is the ToA correction for charge transfer speed dependence in the detector chip.

7.6. Other sensor devices

7.6.1. Timepix alias Medipix2

Has **no setOperationMode**. All pixels can be configured individually.

Use the `pixcfg.setModeAll` or other `pixcfg.setMode...` methods (see [The pixels matrix configuration objects](#))

It has **no subframes**, producing only one type of data at a time.

timepixClock()	Returns actual Timepix clock value in MHz.
setTimepixClock(clock)	Sets Timepix clock divider for clock value nearest the desired MHz. Warning: Conversion ToT to energy is calibrated only for factory default clock.
afterSetDacDelay()	Returns actual delay time [s], that will be used after set of DAC.
setAfterSetDacDelay(delay)	Sets delay time [s], that will be used after set of DAC.

7.6.2. Timepix2

Has **setOperationMode**

Has subframes, it can simultaneously produce two types of data.

totClock()	Returns actual ToT clock value in MHz.
setTotClock(totClock)	Sets ToT clock divider for clock value nearest the desired MHz. Warning: Conversion ToT to energy is calibrated only for factory default ToT clock.
toaDivider()	Returns ToA clock divider index.
setToaDivider(toaDiv)	Sets ToA clock divider index. 0: disable clk, 1: no div, 2-30: div. by 2^{n-1} .
toaClock()	Returns actual ToA clock value in MHz.
setAdaptiveGain(on)	Enables or disables the adaptive gain feature (+bias only). Warning: Conversion ToT to energy is calibrated only for AG disabled.
isAdaptiveGainOn()	Returns 1 if adaptive gain enabled or 0 if not.
setAnalogMasking(on)	Enables or disables the analog masking feature.
isAnalogMasking()	Returns 1 if analog masking enabled or 0 if not.

7.6.3. Medipix3

The Medipix3 sensor (IDevMpx3 object) is very specific. It has a large amount of settings.

Has **setOperationMode** but the operating mode can be adjusted in more detail via **pixcfg**.

Can has subframes, it can simultaneously produce two sets of data with different settings include thresholds.

Threshold settings (see [Threshold settings](#)) has threshold index parameter – **thlIdx**: 0/1 or 0 to 7 in color mode.

All of **setThl...**, **incThl...**, **thl** and **maxThl** methods of the **IMpx3PixCfg** has variants with **...thh...** for correction bits of the **high threshold** (See [Threshold correction bits management](#))

Has no ToT and ToA features.

resetMedipix()	Resets the Medipix chip.
thresholdCalibCoeffNames()	Returns list of names. Example: ['A', 'B', 'C', 'D']
isColorModeSupported()	Returns 1 if color mode is supported or 0 if not.
isColorModeEnabled()	Returns 1 if color mode is enabled or 0 if not.
setColorMode(enabled)	Set color mode ON (1) or OFF (0).
isChargeSummingEnabled()	Returns 1 if charge summing mode is supported or 0 if not.
setChargeSumming(enabled)	Set charge summing mode ON (1) or OFF (0).
isContinuousReadWriteEnabled()	Returns 1 if CRW mode is enabled or 0 if not.
setContinuousReadWrite(enabled)	Enables (1) or disables (0) the CRW mode (See Mpx3 manual).
counterDepth()	Returns the counter bits depth value (pixet.PX_MPX3_CNTD... constants).
setCounterDepth(depth)	Sets the counter bits depth. Use the pixet.PX_MPX3_CNTD... constants.
selectedCounter()	Returns selected counter(s).
setSelectCounter(counter)	Select counter(s). Use the pixet.PX_MPX3_CNT... constants.
columnBlock()	Returns index of active column block(s).
setColumnBlock(colBlock)	Sets index of active column block(s). Use pixet.PX_MPX3_COLB... constants. Note: The setOperationMode will reset this.
rowBlock()	Returns index of active row block(s).
setRowBlock(rowBlock)	Sets index of active row block(s). Use pixet.PX_MPX3_ROW... constants. Note: The setOperationMode will reset this.
isEqualizeEnabled()	Returns 1 if equalize is enabled or 0 if not.
setEqualize(enabled)	Enables (1) or disables (0) equalization.
isDiscCsmSpmEnabled()	Returns 1 if CSM/SPM discrimination is enabled or 0 if not.
setDiscCsmSpm(enabled)	Enables (1) or disables (0) CSM/SPM discrimination (See Mpx3 manual).

Mode related settings: Copying the setting variant according to the selected OPM from config to DACs/PixCfg

dacsMode(opMode)	Returns the IMpxCACs object with the opMode set.
setDacsMode(opMode, dacs)	Sets opMode of the IMpxDacs object.
createCopyOfDacsMode(opMode)	Returns copy of the IMpxDacs object. (see The IMpxDacs object)
pixCfgMode(opMode)	Returns the IMpx3PixCfg object with the opMode set.
createCopyOfPixCfgMode(opMode)	Returns copy of the IMpx3PixCfg object.
setPixCfgMode(opMode, pixCfg)	Sets opMode of the IMpx3PixCfg object. (see The pixels matrix configuration objects)

setMinThresholdNoise(ch, thlIndex, thresholdKev)	Sets threshold energy of bound of noising on the chip.
isThresholdCalibrated(ch, thlIndex)	Returns 1 if the chip calibrated the threshold or 0 if not.
setThresholdCalibCoeffs(ch, thlIndex, gainMode, bitDepth, double* calibParams, size)	Sets threshold calib coefficients for the specified parameters.
thresholdCalibCoeffs(ch, thlIndex, gainMode, bitDepth, double* calibParams, size_t size)	Gets threshold calib coefficients for the specified parameters.

gain() Returns the active gain option of the charge convertor.
setGain(gain) Sets the gain option. Use `pixet.PX_MPX3_GAIN...` constants.

(high-low ticks per pulse)		(narrow-broad max pulse width)
<code>PX_MPX3_GAIN_SUPERHIGH</code>	alias	<code>PX_MPX3_GAIN_SUPER_NARROW</code>
<code>PX_MPX3_GAIN_HIGH</code>	alias	<code>PX_MPX3_GAIN_NARROW</code>
<code>PX_MPX3_GAIN_LOW</code>		
<code>PX_MPX3_GAIN_SUPERLOW</code>	alias	<code>PX_MPX3_GAIN_BROAD</code>

`PX_EVENT_MPX3OPM_CHANGED` Event of operation mode changed: Useful with `..Mode` methods.

See: [Constants: Mpx3 settings](#)

7.6.3.1. Synchronizing multi-device Mpx3 instruments

Sometimes it is necessary to synchronized with an external event. Or some instruments, Widedpix-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.

Decision if device is master example

```
tm = pars.get("TriggerMaster").getBOOL()
print("    TriggerMaster:", tm)
```

Master setup example

```
pars = devices[masterIdx].parameters()
rc = pars.get("TriggerStg").setByte(0)
print("    TriggerStg .setByte", rc)
rc = pars.get("TriggerWaitForReady").setBOOL(True)
print("    TriggerWaitForReady .setBOOL", rc)
rc = pars.get("TriggerOutLevel").setByte(1)
print("    TriggerOutLevel .setByte", rc)
```

Slave setup example

```
pars = devices[slaveIdx].parameters()
rc = pars.get("TriggerStg").setByte(2)
print("    TriggerStg .setByte", rc)
rc = pars.get("TriggerWaitForReady").setBOOL(True)
print("    TriggerWaitForReady .setBOOL", rc)
rc = pars.get("TriggerOutLevel").setByte(0)
print("    TriggerOutLevel .setByte", rc)
```

Synchronized acquisition example

```
# start slaves before master
rc = devices[slaveIdx].doContinuousAcquisition
    (bufCount, frameTime, pixet.PX_ACQMODE_TRG_HWSTART)
print("slave dev", slaveIdx, "doContinuousAcquisition start", rc, "(0 is OK)")
# start master after all slaves
rc = devices[masterIdx].doContinuousAcquisition
    (bufCount, frameTime, pixet.PX_ACQMODE_TRG_HWSTART)
print("master dev", masterIdx, "doContinuousAcquisition start", rc, "(0 is OK)")
```

Note If the acquisition time is long, approximately from 100 ms, the run is irregular and there may be long pauses (1 to 5 * acqTime) between acquisitions. This can be suppressed if master acqTime is approximately over 10 % longer.

The Widedpix-L Medipix3's TriggerStg and TriggerOutLevel are linked in the following way:

Function	Master TriggerStg	Master TriggerOutLevel	Slave TriggerStg	Slave TriggerOutLevel
Sync 1:1	0	0	3	X
Alternating sync	0	0	2	X
Alternating sync	0	1	3	X
Sync 1:1	0	1	2	X

X - the value doesn't matter

8. Other device objects

Devices in this chapter are accessories for detector devices and can be used if have a hwlib dynamic link library file compatible with the Pixet core. These devices can be supplied at the customer's request, or customer's devices can be integrated on request.

These devices have standard device config methods, with config in device supported and general device methods, exclude detector-only methods.

(see [Methods: Device info](#), [Methods: Device config](#) and [Methods: General](#))

8.1. The IDevMotor device

moveAbsolute(dest, wait)	Moves motor to position [units from settings], if wait, waits for end.
moveAbsoluteSteps(dest, wait)	Moves motor to position [steps], if wait, waits for end.
moveRelative(diff, wait)	Moves motor position by [units from settings], if wait, waits for end.
moveRelativeSteps(diff, wait)	Moves motor position by [steps], if wait, waits for end.

The move... methods returns 0 if done, -1 if destination > high limit or -2 if destination < low limit.

isRunning()	Returns 1 if motor is running or 0 if not.
stop()	Stops motor if running.
setZero()	Sets value of motor position to zero. Only if you are sure the motor is in zero position.
position()	Reads the actual motor position in units from settings.
positionSteps()	Reads the actual motor position in steps of motor.

8.2. The IDevXray device

turnXrayOn()	Turns on the x-ray source.
turnXrayOff()	Turns off the x-ray source.
setVoltage(voltage)	Sets x-ray tube voltage in kV.
setCurrent(current)	Sets x-ray tube current in μ A.
voltage()	Returns actual value of the x-ray tube voltage.
current()	Returns actual value of the x-ray tube current.
minVoltage()	Returns minimal allowed x-ray tube voltage in kV.
maxVoltage()	Returns maximal allowed x-ray tube voltage in kV.
minCurrent()	Returns minimal allowed x-ray tube current in μ A.
maxCurrent()	Returns maximal allowed x-ray tube current in μ A.
temperature()	Returns actual temperature value.
isXrayOn()	Returns 1 if x-ray source is on, or 0 if off.
isInterlockOn()	Returns 1 if interlock contact is closed, or 0 if not.

9. Accessing the named parameters

The counter-based imaging devices have large amount of parameters and majority of them have not own separate set/get method. In addition, many such parameters are specific to different types of devices. Solution is text named parameters. The device parameter objects are intended for manipulate with the named parameters. Many of this parameters are read-only from the device (TemperatureCpu) therefore it is widely named “readout parameters”. But also existing writable (TrgStg) or named parameters of the system (DDBlockSize).

Example:

```
pars = dev.parameters()
print("CPU Temp:", pars.get("TemperatureCpu").getDouble(), end=' ')
print(", Chip Temp:", pars.get("TemperatureChip").getDouble())
```

9.1. The IParamMgr object and his methods

This is the interface to access IParam, named parameters objects in the device.

count()	Returns number of named parameters of the device.
params()	Get array of the the IParam objects.
hasParam(name)	Returns 1 if the device has a parameter with the name or 0 if not.
get(name)	Get single IParam object with the name.

9.2. The IParam object and his methods

This is object containing name, description, flags and the parameter value.

name()	Returns text name of the data in the object.
byteSize()	Returns size of data in bytes.
description()	Returns description text for data.
flags()	Returns flags of the data.
setFlags()	Set flags of the data.
type()	Returns index of type of the data (see Data types).

getBOOL, getByte, getChar, getDouble, getFloat, getI16, getI32, getI64, getU16, getU32, getU64, getString

Returns value of the data. Works properly only if the correct data type is used.

setBOOL, setByte, setChar, setDouble, setFloat, setI16, setI32, setI64, setU16, setU32, setU64, setString

Set the data value. Works properly only if the correct data type is used.

Note: Majority of a named parameters are read only.

9.3. Parameters names lists by device types

These are lists collected from the Pixet C++ source code. Use the strings. Names and notes can sometimes be helpful.

9.3.1. Tpx3 parameter names list

```
#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
```

9.3.2. Mpx2 parameter names list

```
#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
```

9.3.3. Mpx3 parameter names list

```
#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"
```

9.3.4. Zest-wpxdev parameter names list

```
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";
```

9.3.5. Zem-wpx7dev parameter names list

#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"

10. The pixels matrix configuration objects

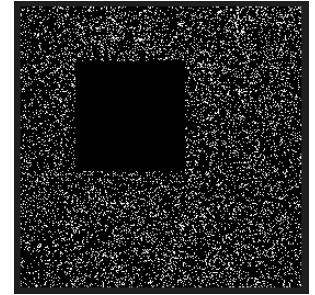
The imaging devices can create this objects to configure some pixel matrix parameters. Some devices have some parameters directly global accessible, some devices have similar parameters accessible via pixels matrix config and some have both. For example, ITpx3Dev have the setOperationMode, but IMpx2Dev have not and the pixCfg.setModeAll must be used instead. The ITpx3Dev devices creates **ITpx3Cf**g object, the IMpx2Dev (Medipix2-Timepix) devices creates **IMpx2TpxPixCf**g and the IMpx3Dev (Medipix3) creates **IMpx3PixCf**g.

Example1:

```
pixCfg = dev.pixCfg() # Set mode for all pixels on Timepix
pixCfg.setModeAll(pixet.PX_TPXM0DE_TOT)
```

Example2:

```
pixCfg = dev.pixCfg() # Mask rectangle at 50,50, size 100,100
print("pixCfg", pixCfg)
print("pixCfg.maskRect", pixCfg.maskRect(50, 50, 100, 100, True))
```



10.1. General

byteSize()	Returns byte size of the object data in the memory.
size()	Returns number of pixels in the matrix.
width()	Returns pixel width of the matrix.
height()	Returns pixel height of the matrix.
pixCfgType()	Returns type of the matrix configuration. (see Constants: Pixels matrix config types)
loadFromFile(path, type)	Loads pixels matrix configuration. (see Constants: File types, extensions and flags)
saveToFile(path, type)	Saves pixels matrix config to file. (see Constants: File types, extensions and flags)
loadFromSettings(settings)	Loads pixels matrix config from the ISettings object (see The ISetting object)
saveToSettings(settings)	Saves pixels matrix config to the ISettings object (see The ISetting object)
setToDevice()	Sets the all the pixel matrix configuration from the object memory to the device.
setDefault()	Sets all the pixel matrix config to maskBit, testBit and thl =0, reloads dev maskbits.
destroy()	Destroys the object.

Note: The **IMpx3Dev** have additional special pixCfg management with sets for various OPMs. See [Medipix3](#)

10.2. IMpx2TpxPixCfg operation modes management

Notes: Matrix mode settings **not available at all chips**. Timepix yes, Timepix3 no, for example.
See [Constants: Operation modes](#)

setMode(px, mode)	Set the operation mode of the indexed pixel.
setModeAll(mode)	Set the operation mode of whole device.
setModeColumn(col, mode)	Set the operation mode of whole indexed column.
setModeColumnChip(col, ch, mode)	Set the operation mode of whole indexed column in the indexed chip.
setModeRow(row, mode)	Set the operation mode of whole indexed row.
setModeRowChip(row, ch, mode)	Set the operation mode of whole indexed row in the indexed chip.
setModeRect(x, y, w, h, mode)	Set the operation mode of whole rectangle with XY position and WH dimension.
setModeMatrix(buff, isChipByChip)	Set the operation mode of whole device by buffer contents.
setModeMatrixChip(ch, buff)	Set the operation mode of whole indexed chip by buffer contents.
mode(px)	Returns operation mode of the indexed pixel.
modeAll()	Returns operation mode of all the device, where was set by setModeAll. Note: If it was later changed on part of the matrix, modeAll returns average.
modeMatrix(buff, isChipByChip)	Read whole device mode values to the buffer.
modeMatrixChip(buff, ch)	Read whole indexed chip mode values to the buffer.
maxMode()	Returns maximum usable value of operation mode in the device.

10.3. Threshold correction bits management

setThl(px, thl)	Set the single threshold correction value in the single indexed pixel.
setThlAll(thl)	Set the single threshold correction value in whole device.
setThlAllChip(ch, thl)	Set the single threshold correction value in whole indexed chip.
setThlChip(px, ch, thl)	Set the single threshold correction value in the single indexed pixel on indexed chip.
setThlColumnChip(col, ch, thl)	Set the single threshold correction value in whole column of the indexed chip.
setThlMatrix(buff)	Set all the threshold correction values in whole device from the buffer.
setThlMatrixChip(ch, buff)	Set all the threshold correction values in whole indexed chip from the buffer.
setThlRect(x, y, w, h, thl)	Set the threshold correction value in whole rectangle on XY position with WH size.
setThlRow(row, thl)	Set the threshold correction value in whole indexed row.
incThlColumn(row, inc, mOfr)	Increment threshold corrections in the indexed column by the inc value.
incThlRow(row, inc, mOfr)	Increment threshold corrections in the indexed row by the inc value. mOfr = true: Mask pixels if result out of range / mOfr = false: No mask, only limit.
incThlCountingOver(frame, val, inc, mOfr)	Increment threshold by the inc value if the corresponding pixel in the frame is over the val value.
thl(px)	Returns the threshold correction value in the single indexed pixel.
thlChip(px, ch)	Returns the threshold correction value in the single indexed pixel on indexed chip.
thlMatrix(buff, isChipByChip)	Read whole device threshold correction values to the buffer.
thlMatrixChip(buff, ch)	Read whole indexed chip threshold correction values to the buffer.
averageThl()	Returns the average threshold correction value in the device.
averageThlChip(ch)	Returns the average threshold correction value in the indexed chip.
maxThl()	Returns max possible value of the threshold correction in the device.

Mpx3 note: All setThl..., incThl..., thl and maxThl has variants with **...thh...** for corr. bits of the **high threshold**

10.4. Masks management

mask(px, mask)	Mask (mask = true) or unmask (mask = false) the indexed single pixel.
maskAll(mask)	Mask (mask = true) or unmask (mask = false) whole device.
maskAllChip(ch, mask)	Mask (mask = true) or unmask (mask = false) whole indexed chip.
maskChip(px, ch, mask)	Mask (mask = true) or unmask (mask = false) single pixel on the indexed chip.
maskColumn(col, mask)	Mask (mask = true) or unmask (mask = false) whole indexed column in the device.
maskRow(row, mask)	Mask (mask = true) or unmask (mask = false) whole indexed row in the device.
maskRect(x, y, w, h, mask)	Mask/unmask the rectangle at XY position with WH dimensions.
maskColumnChip(col, ch, mask)	Mask (mask = true) or unmask (mask = false) whole column on the indexed chip.
maskRowChip(row, ch, mask)	Mask (mask = true) or unmask (mask = false) whole row on indexed chip.
setMaskMatrix(buff, isChipByChip)	Set all the mask matrix in the device by the buffer data.
setMaskMatrixChip(ch, buff)	Set all the mask matrix on the indexed chip by the buffer data.
maskMatrix(isChipByChip)	Returns list with whole the mask matrix from the device.
maskMatrixChip(buff, ch)	Read whole the mask matrix on the indexed chip to buffer.
maskedCount()	Returns number of masked pixels in the device.
maskedCountChip(ch)	Returns number of masked pixels on the indexed chip.
isMasked(px)	Returns 1 if indexed pixel is masked or 0 if not.
isMaskedChip(px, ch)	Returns 1 if indexed pixel on indexed chip is masked or 0 if not.
isMaskedColumn(col)	Returns 1 if all pixels on indexed column is masked or 0 if not.
isMaskedColumnChip(col, ch)	Returns 1 if all pixels on indexed column on indexed chip is masked or 0 if not.
isMaskedRow(row)	Returns 1 if all pixels on indexed row is masked or 0 if not.
isMaskedRowChip(row, ch)	Returns 1 if all pixels on indexed row on indexed chip is masked or 0 if not.
setSpacingPixCfg(spac, subldx)	Sets mask with spacing. The chip is divided into squares of size spac . In each square, only the pixel of the selected subldx is left active.

The pixcfg change in **setSpacingPixCfg** is preceded by:

1. If it is used for the first time, creates backup copy of original pixcfg.
2. If the backup exists, restores the backup.
3. If spacing<2 sends restored data to the device and returns.

11. The IMpxDacs object

Many sensor devices containing digital-to-analog converters. These are used to set some internally used voltages and currents Refer to the device/chip datasheets for details on each channel meaning. The devices also have an ADCs to check real output values (don't confuse with **dev.adc...** functions. see [Methods: Device info](#) for it).

Notes: This is very low-level thing, direct driving of DACs is **not needed for normal device using**. Change of some DAC channels can have influence to the output data = lost of calibration.

Warnings: Using the wrong settings can cause to strange behaviour, sometimes even **damaging the device**. To correct strange behavior after experimenting with DAC values, it may be necessary to physicaly disconnect the power supply, wait a few seconds to a few tens of seconds, and reconnect.

Create: **dev.dacs()** Returns the IMpxDacs object to control the devices DACs.
dev.createCopyOfDacs() Returns the IMpxDacs object to store all the device DACs settings.
 See [DACs control](#)

Methods:

chipType()	Returns chip type on which is the DAC.
count()	Returns count of channels in whole the device.
singleChipCount()	Returns count of channels one sensor chip.
names()	Returns list of all channels names.
namesList(list)	Sets list of all channels names.
maxValue(dacIdx)	Returns max value of indexed DAC channel.
dac(dacIdx, chipIdx)	Returns value of the indexed DAC channel on the indexed chip.
setDac(dacIdx, chipIdx, value)	Sets value of the indexed DAC channel on the indexed chip.
setToDevice()	Apply DAC parameters stored in memory to the device.
analogDac(dacIdx, chipIdx, cnt)	Returns measured average output value of cnt samples or -1 if failed.
setDacAndGetAnalog(dacIdx, chipIdx, val, cnt)	Sets value, apply, measures real output and returns result.
setExternalDac(eDacIdx, chipIdx, val)	Sets external DAC. Note: Many devices have not ext. DACs.
setDefault()	Fills all of the DACs settings by defaults and apply to the device.
loadFromFile(fileName)	Loads all the DACs settings from file and apply to the device.
saveToFile(fileName)	Saves all the DACs settings to the file.
loadFromSettings(settings)	Loads all the DACs settings from ISettings and apply to the device.
saveToSettings(settings)	Stores actual DACs settings to the ISettings object.
copyFromDacs(dacs)	Copies DACs settings from other IMpxDacs object.
destroy()	Destroys the IMpxDacs object. Only if created by the dev.createCopyOfDacs() .

Note: The **IMpx3Dev** have additional special DACs management with sets for various OPMs.
 See [The IDevMpx3 object differences](#)

12. The ISetting object

This object can be used to access settings in XML, INI, and other settings or container files. The ISettings objects can be created by Pixet core methods **createSettings(filePath, stgType)**, **createPluginSettings(name, IDev, stgType)** (see [Methods: Other](#)) and device method **userSettings()** (see [Methods: Device config](#)).

A device has the **userSettings()** method. It returns the **ISettings** object. Programmer can add your settings to settings tree for future use. This is maintained as other device settings include automatic loading/saving to config. Notes: The userSettings section is not affected by the **loadFactoryConfig**. In combination with the **dev.saveConfigToSettings** it can cause continual growing of the config by saving copy of all settings to user settings, but copied settings contains userSettings too...

getPathItems(group)	Returns list of items names at the specified group/path starting with the string.
itemExists(group, key)	Returns 1 if specified item exists in the specified group/path or 0 if not.
destroy()	Clears all. It will causes delete the UserSettings section from config at exit core.
setString(group, key, value)	Sets or creates specified string value of specified key in specified group/path. Note: "grp" or "a\\b\\c" creates groups named "grp" or "a\b\c" "a/b/c" creates tree structure with "a", "b" and "c" dirs and subdirs. Warning: The \ in a names/groups can cause problems.
getString(group, key, default)	Returns string value of specified key in specified group or default value if not exist.
setBool, setBuffer, setDouble, setString	Working like as setString.
getBool, getBuffer, getDouble, getString	Working like as getString.

Example:

```
uset = dev.userSettings()
print(uset.getString("myGroup", "myStr", "(default)"))
print(uset.setString("myGroup", "myGroup", "myVal"))
print(uset.getString("myGroup", "myStr", "(default)"))
# First start: (default) 0 myVal / Second start: myVal 0 myVal
```

```
<A\B\C>
  <S>test</S>
</A\B\C>
<a>
  <b>
    <c>
      <s>test</s>
    </c>
  </b>
</a>
<myGroup>
  <myStr>myVal</myStr>
</myGroup>
```

Programmer can use **pixet.createSettings(filePath, stgType)** to obtain general ISetting object with own config file.

Example: `pixet.createSettings("test.xml", pixet.PX_STGTYPE_XML)`

Or **createPluginSettings(name, IDev, stgType)** to obtain settings associated with the Pixet plugin and the device. This settings can be loaded/saved from/to "configs" directory with name like as "IDevControl_I08-W0060.xml".

Example: `pixet.createPluginSettings("IDevControl", dev.asIDev(), pixet.PX_STGTYPE_XML)`

See [Constants: Settings types](#) for PX_STGTYPE_... constants list

getFilePath()	Returns full path of the settings file (except user settings).
setFilePath()	Sets filename or full path of the settings file (except user settings).
load()	Loads the settings from the associated file (except user settings).
save()	Saves the settings to the associated file (except user settings).
getLastError()	Returns last error string from the plugin (plugins only).

13. The IMpxFrame and other frame objects

This object can be used in frame based mode to access acquired frames. Depending on device type and operation mode, frame can contain single data block (old Medipixes), or contains subframe(s) with data blocks. Number of a subframes depending on operation mode and other device settings.

Example: `dev.doSimpleAcquisition(3, 1, pixet.PX_FTYPE_NONE, "")`
 `frame = dev.lastAcqFrameRefInc()`

13.1. Frame access and related methods

subFrameCount()	Returns number of subframes in the frame. 0: Frame contains valid data block (IMpxNDev) or something is wrong. 1 or more: Frame contains a data blocks with remnants of data parsing (ITpxNDev). Valid data blocks are in the subframes.
subFrames()[idx]	Get array of the subframes. Working if subFrameCount is over zero.
duplicateFrame()	Get a copy of the frame.
copyFromFrame(source)	Copy source frame to the frame.
asIData()	Access the frame as IData object.
destroy()	Destroy the frame object. Do this at end of work with the frame.
addSubFrame(string name)	Add named subframe to the frame.
removeSubFrame(idx)	Remove indexed subframe.
removeAllSubFrames()	Remove all subframes.
init(width, height, type)	Init the frame. Do this after addSubFrame (see Constants: Frame types).
initFromFile(path, index)	Init the frame. Do this after addSubFrame (use like as load)
save(path, fType, flags)	Save the frame data to the specified file. path: File name or path. fType, flags: See Constants: File types, extensions and flags
load(path, index)	Load the frame data from the specified file. Index is the subframe index.
lock(timeout)	Lock the frame to prevent access from other threads.
unlock()	Unlock the frame.
isLocked()	Returns 1 if the frame is locked or 0 if not.
data()	Gets the data array. Note: Valid data is usually in subframes.
setData(array)	Sets the data array.
byteSize()	Returns size of the data array in bytes.
copyDataFromFrame(source)	Copy data from source frame to the frame.

Data array in the ToA subframe

If **ConvertToaTime** is **enabled** on the device, this array contains the Time Of Arrival values in nanoseconds. On Tpx3 devices, values are computed from 14bit ToA counter with 40 MHz clock and refined with small fractional counter with higher clock. Maximal value is about 409,600 ns. ToA counter is resets at start of the acquisition, but fractional counter not reset. This leads to a minimal values is not zero, but can reach small negative values about - 25 ns. Pixels that was not hit, contains zero.

If **ConvertToaTime** is **not enabled** on the device, this array contains only raw value of ToA counters.

Data array in the FTOA subframe

This subframe exist only if the operation mode supports ToA and **ConvertToaTime** is **not enabled** on the device. Contains raw values of the ToA fine counter.

Data array in the ToT subframe

If **useCalibration** is **enabled** on the device, this array contains energy in keV absorbed in the pixels.

If **useCalibration** is **not enabled** on the device, this array contains the Time Over Threshold values. On Tpx3 devices, this is raw values from 10bit counter. Min is 0, max 1022.

Data array in the Event subframe

This array contains Event counts. On Tpx3 devices, this is raw values from 10bit counter.

Data array in the IToT subframe

This array contains Integrated Time Over Threshold values. On Tpx3 devices, this is raw values from 14bit counter. Min is 0, max 16,383.

13.2. Frame parameters methods

frameName()	Returns name of the frame. In root frame is void, in subframe contain output name.
frameType()	Returns type of data in the frame (see Constants: Frame types)
width()	Returns width of the frame in pixels.
height()	Returns height of the frame in pixels.
size()	Returns number of pixels in the frame.
device()	Get device object with source device parameters.
isDeviceFrame()	Returns 1 if the frame is a device frame or 0 is not.
isSubFrame()	Returns 1 if the frame is a subframe or 0 is not.
startTime()	Returns the system time of start acquisition.
acqTime()	Returns the acquisition time in seconds.
dataFormat()	Returns index of data format in the object (see Constants: Data formats).

setAcqTime() setStartTime() setDevice() setFrameName() setFrameType() Set the parameters listed above.

13.3. Frame statistics and bulk mathematic methods

min(), max(), mean(), median(), sum()	Returns basic statistics of pixel data in the frame.
stdDev()	Returns standard deviation.
minNonZero()	Returns minimum non-zero value from pixel data in the frame.
nonZeroCount()	Returns number of non-zero values in pixel data in the frame.
aboveValueCount(value)	Returns number of values above specified.
multiplyWithFrame(frame)	Multiply all pixel data in the frame by specified frame pixel data.
divideWithFrame(frame)	Divide all pixel data in the frame by specified frame pixel data.
addToFrame(frame)	Add all pixel data from specified frame to the frame pixel data.
subtractFromFrame(frame)	Subtracts all pixel data in the frame from specified frame pixel data.
multiplyWithValue(value)	Multiply all pixel data in the frame by specified value.
fillWithValue(value)	Fill the frame data with the value.
fillWithZeros()	Fill the frame data with the zeros.

13.4. Frame metadata and his handling methods

Every frame can contain some metadata. This can be used in postprocessing. Metadata can be saved automatically with saving data to the auxiliary files (eq. DSC with TXT files). Programmer can manage it before saving files. If file saving is automatic, metadata can be managed using the beforeDataSave event (see [Methods: Events](#)).

Metatadata in DSC file example:

```
"Frame name" ("Frame name"):
char[5]
Event
```

Every metadata item contains strings Name and Description, number of the DataType and the data.

addMetaData(name, description, type, value)	Add metadata item to the frame. (see Related constants: Data types)
removeMetaData(name)	Remove metadata item from the frame.
removeAllMetaData()	Remove all metadata from the frame and his subframes.
hasMetaData(name)	Returns 1 if the frame has metadata with the name.
metaDataCount()	Returns number of metadata items in the frame.
metaDataNames()	Get the array of a metadata names in the frame.
metaData(name)	Access the metadata: metaData(name).data()

List of all frame's metadata example:

```
print("metaDataCount:", frame.metaDataCount())
for mdn in frame.metaDataNames():
    print("  ", mdn, frame.metaData(mdn).data())
```

13.5. Other frames methods

convertToChipByChip()

Converts multi-chip frame to arrangement like as single horizontal line of chips.

convertFromChipByChip()

Converts multi-chip frame from single line of chips to configured arrangement.

Note: The **chip-by-chip format** is used in low level functions

Example: `dev.pixCfg().setMaskMatrix(data_chipByChip)`

13.6. Frame with subframes example

```

dev.setOperationMode(pixet.PX_TPX3_OPM_TOATOT)
print("dev.doSimpleAcquisition - start")
# count, time, fileType, fileName
rc = dev.doSimpleAcquisition(count, time, pixet.PX_FTYPE_AUTODETECT, "")
print("dev.doSimpleAcquisition - end", rc)
print("acqFrameCount", dev.acqFrameCount())
for n in range(dev.acqFrameCount()):
    frame = dev.acqFrameRefInc(n)
    data = frame.data()
    print("Frame idx:", n)
    if frame.subFrameCount()>0:
        for sfr in frame.subFrames():
            print("  Subframe:", sfr.frameName())
            print("    min", sfr.min(), 'mean', sfr.mean(), 'max', sfr.max())
            data = sfr.data()
            print("    data sample:")
            print("      ", end="");
            for i in range(256*120+100, 256*120+110):
                print(data[i], end=" ")
            print()
            sfr.save("test-{}".format(n)+sfr.frameName()+".png",
                    pixet.PX_FTYPE_AUTODETECT, 0)

            sfr.destroy()
        #for sfr in frame.subFrames()
    else:
        print("  No subframes")
    #if frame.subFrameCount()>0
    frame.destroy()
#for n in range(dev.acqFrameCount())

```


14. The ITpx3Pixels and other pixels objects

This objects can be used in data-driven mode to direct access the measured data. Object have methods to get raw pixels data arrays or prepared data arrays and few methods to get parameters.

Example: `dev.doAdvancedAcquisition(1, 5, pixet.PX_ACQTYPE_DATADRIVEN, ...`
`pixels = dev.lastAcqPixelsRefInc()`

14.1. Basic pixels access and related methods

totalPixelCount()	Returns total pixels count over all the measurement.
pixelCount()	Returns count of pixels where actually can be processed.
pixels()	Get the array containing pixel position index, ToA in ns and ToT counter value.
rawPixels()	Get the array containing pixel position index, cyclic ToA count, reserved, fine ToA and ToT.

Example:

```
pixels = dev.lastAcqPixelsRefInc()
pxCntTot = pixels.totalPixelCount() # All acquired pixels count
pxCnt = pixels.pixelCount()         # Pixels to be read now
pixelsData = pixels.pixels()
pixelsRaw = pixels.rawPixels()

for n in range(pxCnt):
    ind = pixelsData[0][n] # The pixel index
    toa = pixelsData[1][n] # ToA in nanoseconds
    tot = pixelsData[2][n] # Raw ToT counter value

    Rind = pixelsRaw[0][n] # The pixel index
    Rtoa = pixelsRaw[1][n] # cyclic ToA counts value
    Rres = pixelsRaw[2][n] # reserved (=0, weakly other)
    Rfto = pixelsRaw[3][n] # Raw fine ToA counter value
    Rtot = pixelsRaw[4][n] # Raw ToT counter value
```

ToA from the **pixels** method is time in nanoseconds. It is limited to $2^{31}-1$ ($0x7fffffff = 2,147,483,647$ ns = **2.147 sec**) and overflow is indicated by value -2^{31} ($-0x80000000 = -2,147,483,648$).

ToA from the **rawPixels** method is time in ToA base ticks 40 MHz (25 ns/tick). Max value is $2^{31}-1$ (2,147,483,647 ticks = **53.68 sec**). After overflow, value jump to -2^{31} and continues to rise for next 107,37 sec. Counting of this overflows can be used to get unlimited time.

Example with endless ns:

```
if rawToaLastSgn==0 and Rtoa<0: # was +, now -
    rawToaLastSgn=1;
    rawToaAdd += 2**32;
elif rawToaLastSgn==1 and Rtoa>0: # was -, now +
    rawToaLastSgn=0;
ToA_ns = (Rtoa + rawToaAdd)*25 - 25/16*Rfto
```

14.2. Pixels other methods

device()	Get device object with source device parameters.
dataFormat()	Returns index of data format (see Constants: Data formats) normally is TPX3_PIXELS .
destroy()	Destroy the object.
previewMpxFrame()	Create a frame with preview of the pixels data.
fadeWindowSize()	Return size of the time window in ns for use in previewMpxFrame .
setFadeWindowSize(time)	Set the fadeWindowSize .
save(path, fType, flags)	Save the data to the specified file. path: File name or path. fType, flags: See Constants: File types, extensions and flags
load(path, index)	Load the data from the specified file. Index is the subframe index.
lock(timeout)	Lock the object to prevent access from other threads.
unlock()	Unlock the frame.
isLocked()	Returns 1 if the object is locked or 0 if not.

14.3. Pixels metadata methods

addMetaData, removeMetaData, removeAllMetaData
hasMetaData, metaDataCount, metaDataNames, metaData

All metadata methods in the pixels objects are same us frame metadata methods.
 For details see: [Frame metadata and his handling methods](#)

14.4. The ITpx3Pixels example: After doAcquisition

```
import pypixet

print("pixet core init...")
pypixet.start()
pixet=pypixet.pixet
dev = pixet.devicesByType(pixet.PX_DEVTYPE_TPX3)[0]

dev.setOperationMode(pixet.PX_TPX3_OPM_TOATOT)

print("doSensorRefresh start")
rc = dev.doSensorRefresh()
print("doSensorRefresh end:", rc)

print("doAdvancedAcquisition start")
rc = dev.doAdvancedAcquisition(1, 2, pixet.PX_ACQTYPE_DATADRIVEN,
                               pixet.PX_ACQMODE_NORMAL, pixet.PX_FTYPE_AUTODETECT, 0, "")
print("doAdvancedAcquisition end:", rc)

pixels = dev.lastAcqPixelsRefInc()
pxCnt = pixels.pixelCount()
pixelsData = pixels.pixels()
print("pixels count:", pxCnt)

for n in range(pxCnt):
    ind = pixelsData[0][n] # The pixel index
    toa = pixelsData[1][n] # ToA in nanoseconds
    tot = pixelsData[2][n] # Raw ToT counter value
    print(" ind ToA ToT:", ind, toa, tot)
    if n>100:
        break

pixels.destroy()
print("pixet core exit...")
pixet.exitPixet()
pypixet.exit()
```

14.5. The ITpx3Pixels example: Callbacks

```
import pypixet

print("pixet core init...")
pypixet.start()
pixet=pypixet.pixet
dev = pixet.devicesByType(pixet.PX_DEVTYPE_TPX3)[0]
dev.setOperationMode(pixet.PX_TPX3_OPM_TOATOT)

print("doSensorRefresh start")
rc = dev.doSensorRefresh()
print("doSensorRefresh end:", rc)

def clbNewData(unused):
    # parameter "unused" is allways zero
    pixels = dev.lastAcqPixelsRefInc()
    pxCnt = pixels.pixelCount()
    pxCntTot = pixels.totalPixelCount()
    pixelsData = pixels.pixels()
    print("New data: pxCnt", pxCnt, "Total", pxCntTot, "-----")

    i = 0
    for n in range(pxCnt):
        ind = pixelsData[0][n] # The pixel index
        toa = pixelsData[1][n] # ToA in nanoseconds
        tot = pixelsData[2][n] # Raw ToT counter value
        x = ind % 256
        y = ind // 256
        if x>5 and x<250 and y>5 and y<250:
            print(" XY:", x, y, "ToA", toa, "ToT", tot)
            i += 1
            if i>10:
                break

    pixels.destroy()

dev.registerEvent(pixet.PX_EVENT_ACQ_NEW_DATA, 0, clbNewData)
print("doAdvancedAcquisition start")
rc = dev.doAdvancedAcquisition(1, 2, pixet.PX_ACQTYPE_DATADRIVEN,
                               pixet.PX_ACQMODE_NORMAL, pixet.PX_FTYPE_AUTODETECT, 0, "")
print("doAdvancedAcquisition end:", rc)
dev.unregisterEvent(pixet.PX_EVENT_ACQ_NEW_DATA, 0, clbNewData)
print("pixet core exit...")
pixet.exitPixet()
pypixet.exit()
```

15. The pypxproc object

This object can create data processing objects that can work with clusters from data that comes best from data-driven mode.

Clustering(dev) Get the Clustering object. It can processing a single clusters to looking for interesting single particles and the like.

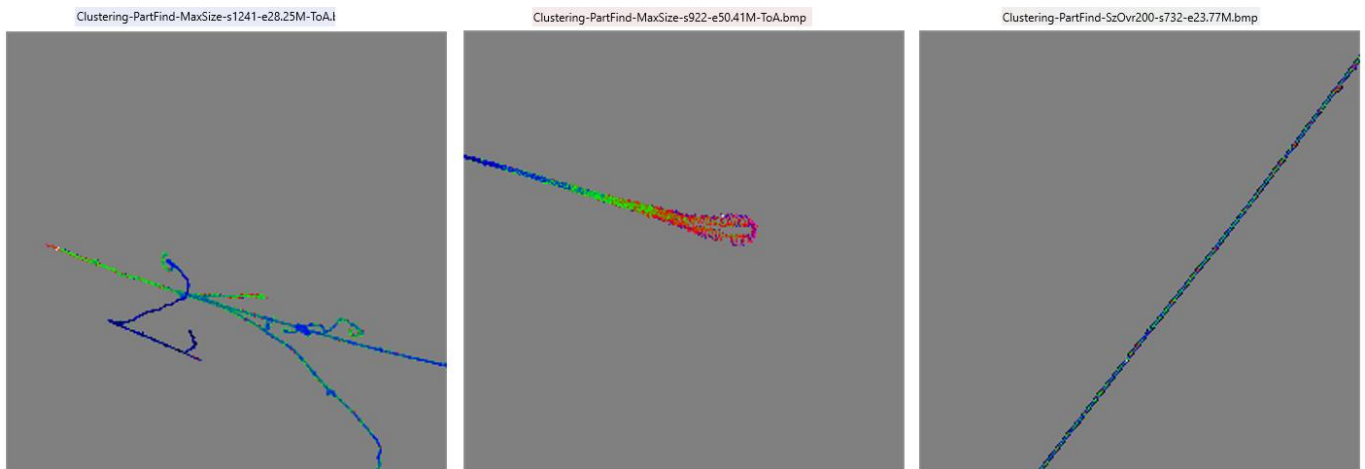
SpectralImaging(dev) Get the Spectralimg object. It can process many clusters into pixels in images, select clusters by energy range, or sort to energy channels..

Syntax note: The name is from Spectra Imaging, not SpectraL magnig)

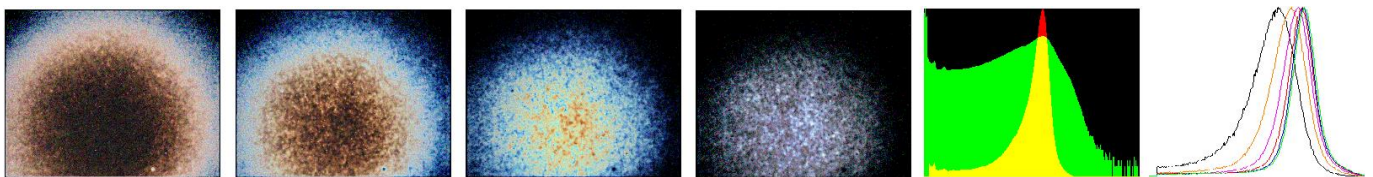
The methods listed above can be used with or without parameter. If not used, only offline processing is possible.

Example1: `si = pypxproc.SpectraImaging()` # offline proces only

Example2: `devices = pixet.devices()`
 `dev = devices[0]`
 `si = pypxproc.SpectraImaging(dev.asIDev())` # can use the device



Clustering - Sample output from an experiment



Spectralimg - Sample output from an experiment

16. The Clustering object and related

The Clustering object is designed for easy work with clusters of the pixels. Typically used to determine energy and other parameters of high energy particles.

16.1. The Clustering methods

loadCalibrationFromDevice() Load the calibration data from IDev connected to the Cl.
loadCalibrationFromFiles(path) Loads ABCT calibration from single XML file. Device config file can be used.
loadCalibrationFromFiles(pathA|pathB|pathC|pathT) Load ABCT calibration data from set of text files.
isCalibrationLoaded() Returns 1 if calibration is loaded or 0 if not.

replayData(pathIn, pathOut, blocking) Process data from the input file.
 Input can be tpx3 pixels: t3pa, t3r, t3p.
 Process the data and calls the corresponding callbacks for further cluster processing.
 If calibration is loaded, energy values will be calibrated.
 If the output path defined and ending with .clog, cluster log will be saved.
 If the blocking is true, program wait to process end, if false, processing is started in a separate thread.

Example: `cl.replayData("input.t3pa", "output.clog", True)`

startMeasurement(acqTime, measTime, pathOut) Start the measurement. Only if the SI connected to the IDev.
 Measurement works in the background. Use while-isRunning() to wait for end, if need it.
measTime: Total time [s] of the measurement. Use 0 to endless measurement (progress will always 100%).
acqTime: Primary for frame-based devices (Medipixes, Timepix, no Tpx3): This is single frame time.
 Use a short enough time to prevent clusters overlapping. Too short time can cause too many losses between frames. On data-driven devices (Timepix3, no Timepix), this is the ToA limit. After exceeds, ToA is resets and acqIndex in the newClusters... callbacks is incremented.

pathOut: Output file path. For data-driven devices (eq Timepix3) required pixel files: t3pa, t3r, t3p
 For Frame-based devices (eq Medipix, Timepix) required cluster log files: clog

processData: True/false, enable/disable online processing.
Warning: Online processing can cause data loss due to insufficient computing power.

isRunning() Returns 1 if clustering process is running or 0 if not.

Example:

```
print("starting measurement...")
rc = cl.startMeasurement(1, 100, "")
while cl.isRunning():
    pass
print("meas. end, rc:", rc, "(0 is OK)")
```

16.2. The Clustering properties

messageCallback	Name of the callback function for message receiving (errCode, messageString).
progressCallback	Name of the callback function for process progress monitoring (progPercent, finishedNum). Occurs approximately twice per second.
acqStartedCallback	Name of the callback function for acquisition started (acqIndex).
acqFinishedCallback	Name of the callback function for acquisition finished (acqIndex).
newClustersCallback	Name of the callback function for new clusters parameters using (clusters, acqIndex).
newClustersWithPixelsCallback	Name of callback function for new clusters data processing (clusters, acqIndex).

The callback parameter “clusters” get the **Clusters** object. This can be simply used like us array to get the **Cluster** object. The Cluster object from the newClustersWithPixelsCallback can be used to get array of the **Pixel** objects.

16.3. The Clusters object

The **Clustering** object have the callback functions named **newClustersCallback** and **newClustersWithPixelsCallback**. His first parameter can be used to get the **Clusters** object containing a clusters. Normally is used as an array of the **Cluster** objects.

Example:

```
def newClustersCb(clusters, acqIndex):
    for i in range(len(clusters)):
        cluster = clusters[i]
```

16.4. The Cluster object

The Cluster object contains a single cluster data. Total size, total energy, position, roundness and can include list of his pixels.

id	Order number of the cluster.
toa	Time of arrival of first cluster's pixel.
x, y	Position of the cluster. This is not normal integer position of one pixel, it's computed cluster center.
size	Number of pixels in the cluster.
height	Pixel size of the cluster.
roundness	1 for an ideal circle with area equal to cluster.size, decreasing by a number related to the ratio of the inscribed circle vs. circumscribed circle.
e	Total energy in the cluster. If the calibration loaded, e is energy in keV, if not, e is sum of pixels ToTs.
pixels	Array of the Pixel objects. Only if the cluster is from the newClustersWithPixelsCallback .

16.5. The Pixel object

Single pixel data from the cluster which was obtained from the **newClustersWithPixelsCallback**.

toa	Time of arrival of first cluster's pixel.
x, y	Position of the pixel.
e	Energy absorbed in the pixel. If the calibration loaded, e is energy in keV, if not, e is the ToT value.

16.6. Clustering constants and error codes

```
// Contants:
#define CL_API_VERSION 1
#define CL_INVALID_HANDLE 0
#define CL_NO_DEVICE -1

// Error Codes:
#define CL_ERR_CANNOT_LOAD_PIXET -1
#define CL_ERR_PIXET_NOT_LOADED -2
#define CL_ERR_INVALID_HANDLE -3
#define CL_ERR_INVALID_DEVICE_INDEX -4

#define CL_ERR_INVALID_ARGUMENT -100
#define CL_ERR_CALIB_DIMENSION_MISMATCH -101
#define CL_ERR_CANNOT_OPEN_FILE -102
#define CL_ERR_CANNOT_READ_FRAME -103
#define CL_ERR_CANNOT_MEASURE -104
#define CL_ERR_LOCK_TIMEOUT -105
#define CL_ERR_DIMENSION_MISMATCH -106
#define CL_ERR_NO_DATA -107
#define CL_ERR_INVALID_RECT -108
#define CL_ERR_CANNOT_READ_FILE -109
#define CL_ERR_CANNOT_SAVE_FILE -110
#define CL_ERR_CANNOT_MASK_PIXEL -111
```

For Pixet core constants see [Constants: Device and chip types](#) and nexts.

16.7. The Clustering examples

16.7.1. Simple list of the clusters

This example only shows sample lists of the clusters with it's parameters using the **newClustersCallback**:

```
import pypixet, pypxproc

def messageCb(error, msg): # =====
    print("*** ErrCode: {}, msg: {}".format(error, msg))

def progressCb(finished, progress): # =====
    print("*** Progress: {:.2f} %, finished={}".format(progress, finished))

def newClustersCb(clusters, acqIndex): # =====
    print("*** NewClusters: cnt={}, acqIndex={}".format(len(clusters), acqIndex))
    for i in range(len(clusters)):
        cluster = clusters[i]
        print("{}: x={}, y={}, e={:.2f}, toa={:.2f}, size={}, height={:.2f}".format(
            i, int(cluster.x), int(cluster.y), cluster.e, cluster.toa, cluster.size,
            cluster.height))
        if i > 20:
            break

print("pixet core init...")
pypixet.start()
pixet=pypixet.pixet
devices = pixet.devicesByType(pixet.PX_DEVTYPE_TPX3)
dev = pixet.devices()[0]

cl = pypxproc.Clustering(dev.asIDev())
cl.messageCallback = messageCb
cl.progressCallback = progressCb
cl.newClustersCallback = newClustersCb

rc = cl.loadCalibrationFromDevice()
print("Load calib", rc)

# startMeasurement(acqTime, measTime, outputFilePath)
rc = cl.startMeasurement(2, 30, "")
print("startMeasurement", rc)

while cl.isRunning():
    pass

print("Exit pixet...")
pixet.exitPixet()
pypixet.exit()
```

16.7.1. List of the clusters with pixels sample

This example only shows sample lists of the clusters with it's parameters and lists of it's pixels, using the `newClustersWithPixelsCallback`:

```
import pypixet, pypxproc

def messageCb(error, msg): # =====
    print("*** ErrCode: {}, msg: {}".format(error, msg))

def progressCb(finished, progress): # =====
    print("*** Progress: {:.2f} %, finished={}".format(progress, finished))

def newClustersWithPixelsCb(clusters, acqIndex): # =====
    print("*** NewClustersWPx: cnt={}, acqIndex={}".format(len(clusters), acqIndex))
    for i in range(len(clusters)):
        cluster = clusters[i]
        print("{}: x={}, y={}, e={:.2f}, toa={:.2f}, size={}, height={:.2f}".format(
            i, int(cluster.x), int(cluster.y), cluster.e, cluster.toa, cluster.size,
            cluster.height))
        for pix in cluster.pixels:
            print("  [ {}, {}, {:.2f}, {:.2f} ]".format(
                int(pix.x), int(pix.y), pix.e, pix.toa))
        if i > 20:
            break

print("pixet core init...")
pypixet.start()
pixet=pypixet.pixet
devices = pixet.devicesByType(pixet.PX_DEVTYPE_TPX3)
dev = pixet.devices()[0]

cl = pypxproc.Clustering(dev.asIDev())
cl.messageCallback = messageCb
cl.progressCallback = progressCb
cl.newClustersWithPixelsCallback = newClustersWithPixelsCb

rc = cl.loadCalibrationFromDevice()
print("Load calib", rc)

rc = cl.startMeasurement(2, 30, "")
print("startMeasurement", rc)

while cl.isRunning():
    pass

print("Exit pixet...")
pixet.exitPixet()
pypixet.exit()
```

16.7.2. The clusters statistics using newClustersCallback

This can be used in program like as previous example. Display some statistics of clusters in each set and global histogram lists after the process ends.

```
def newClustersCb(clusters, acqIndex):
    try:
        lc = len(clusters)
        print("\n*** NewClusters: count={}, acqIndex={}".format(lc, acqIndex))
        e = 0
        em = 0
        s = 0
        sm = 0
        for cluster in clusters:
            eChan = int(cluster.e / newClustersCb.chansEneStep)
            sChan = int(cluster.size / newClustersCb.chansSizStep)
            newClustersCb.chansEne[eChan] += 1
            newClustersCb.chansSiz[sChan] += 1
            e += cluster.e
            if cluster.e > em: em = cluster.e
            s += cluster.size
            if cluster.size > sm: sm = cluster.size
        print(" Sum sizes", s, "Sum E {:.2f} MeV,".format(e/1000),
              "clust.S: avg {:.2f} px/cl, max {:.2f},".format(s/lc, sm),
              "clust.E: avg {:.2f} keV/cl, max {:.2f}M".format(e/lc, em/1000))
    except:
        traceback.print_exc()

newClustersCb.chansEneStep = 30000/250
newClustersCb.chansSizStep = 4
newClustersCb.chansEne = [0] * 256
newClustersCb.chansSiz = [0] * 256
# def newClustersCb

# ... (initialize, load calibration, measuring or replay data)

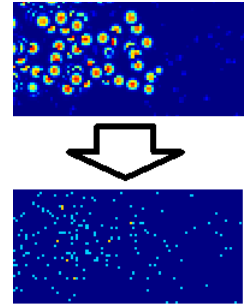
while cl.isRunning():
    pass

print("newClustersCb.chansEne", newClustersCb.chansEne)
print("newClustersCb.chansSiz", newClustersCb.chansSiz)
```

17. The Spectralmg object

The Spectralmg is designed for easy working with an energy spectras. It can work with previous saved data using the **replayData** method or with physical device. For offline mode create the Spectralmg using **pypxproc.SpectralImaging()**. For online mode, the device must be normally initialized by the Pixet object, convert to IDev using device's **asIDev()** method and connected using **pypxproc.SpectralImaging(iddev)**. Pixels in the generated frames have values from clusters detected in the input data.

Warning: Online processing can cause data loss due to insufficient computing power.



Example (online):

```
import pypixet, pypxproc

print("pixet core init...")
pypixet.start()
pixet=pypixet.pixet
devices = pixet.devicesByType(pixet.PX_DEVTYPE_TPX3)
dev = devices[0]
si = pypxproc.SpectraImaging(dev.asIDev())
```

Example (offline):

```
import pypxproc
si = pypxproc.SpectraImaging()
```

Steps for using this object in the online mode:

1. Initialize the Pixet core and create the device object (skip core init if starting from the Pixet program).
2. Optionally use the sensor refresh or dummy acq.
3. Create the Spectralmg object using IDev object converted from the device.
4. Set-up the callbacks.
5. Load device calibration (**loadCalibrationFromDevice** or **loadCalibrationFromFiles**).
6. Set measurement parameters using the **setMeasParams** and possibly **setXrfParams** methods.
7. Start the measurement using **startMeasurement**.
8. Wait for measurement and processing is complete (and display the progress) using **while-isRunning**.
9. Use some get/save... method and use the data.
10. Deinitialize the Pixet core.

Steps for using this object in the offline mode:

1. Create the Spectraimg object using empty parentheses.
2. Set-up the callbacks.
3. Load device calibration using **loadCalibrationFromFiles**.
4. Set measurement parameters using the **setMeasParams** and possibly **setXrfParams** methods.
5. Use the **replayData** instead of a measurement and process data from the t3pa file.
6. Wait for processing complete (and display the progress) using **while-isRunning**.
(If goal is only get the clog file, this is end)
7. Use some get/save... method and use the data.

Using BSTG files to save processing time:

1. After processing is complete (end of waiting steps above), use the **saveToFile("file.bstg")** method.
2. Anytime use the **loadFromFile("file.bstg")** method and continue using the data as it was processed.

17.1. Spectralmg auxiliary methods and properties

loadCalibrationFromDevice() Loads the calibration data from the device that is connected to the SI as an IDev.
(see [The pypxproc object](#))

loadCalibrationFromFiles(path) Loads ABCT calibration data from single XML file. Device config file can be used.

loadCalibrationFromFiles(pathA|pathB|pathC|pathT) Load ABCT calibration data from set of text files.

isCalibrationLoaded() Returns 1 if calibration is loaded to the Spectralmg object or 0 if not.

messageCallback Name of callback function for message receiving (errCode, messageString).
Occurs at start of a processing and on errors.

progressCallback Name of callback function for process progress monitoring (progPercent, finishedNum).
Occurs every approximately 1 second.

Examples: `si.messageCallback = messageClibFun`
`si.loadCalibrationFromFiles("cal_a.txt|cal_b.txt|cal_c.txt|cal_t.txt")`

measuredPixelsPerSecond() Returns actual measuring speed in pixels per second while measuring.

processedPixelsPerSecond() Returns actual data processing speed in pixels per second while measuring.

17.2. Spectraimg measurement and related methods

setMeasParams(from, to, step, maskNP, doSPC) Sets spectral parameters of future data processing.
No return value.

from: Spectral range start [keV] (int only).

to: Spectral range end [keV] (int only).

step: Step width in the spectrum [keV] (float, but values significantly <1 do not work correctly).
The from/to/step are source of energy indexes used in get... methods listed below.

maskNP: Mask (true) or not mask (false) noisy pixels. (Measure only, ignored in replay)

doSPC: Perform (true) or not perform (false) subpixel correction. (influence to sumFrame)

setXrfParams(minVol, maxVol, distance, toaDiff, reserved, correctXrf) Set X-ray fluorescence corr. parameters.
No return value.

minVol: Minimum XRF cluster volume (energy).

maxVol: Maximum XRF cluster volume (energy).

distance: Fluorescent pixels maximal distance.

toaDiff: Fluorescent pixels time difference (depends on chip material and thickness).

reserved: Reserved for future use: enable Remove XRF events.

correctXrf: Enable (true) or disable (false) XRF correction. Detected XRF energy will be replaced by 25 keV.

startMeasurement(acqTime, measTime, pathOut, processData) Start the measurement (physical device).

Measurement works in the background. Use `while-isRunning()` to wait for end, if need it.

measTime: Total time of the measurement.

acqTime: Single frame time. Used only on frame-based devices (Timepix using, Timepix3 ignoring).

Use a short enough time to prevent clusters overlapping. Too short time can cause too many losses.

pathOut: Output file path. Must end with `.clog`.

processData: True/false, enable/disable online processing.

Warning: Online processing can cause data loss due to insufficient computing power.

replayData(pathIn, pathOut, reserved) Use data from the input file like as in measuring.

Input can be pixels: t3pa, t3r, t3p, frames: pmf, txt, h5, bmf, plog, or clog.

Processes the data from input file and calls the corresponding callbacks.

If calibration is loaded, energy values will be calibrated.

If the output path defined and ending with .clog, cluster log will be saved.

The process works in the background. Use while-isRunning() to wait for end, if need it.

Example: `si.replayData("input.t3pa", "output.clog", 0)`

isRunning() Returns 1 if process is running or 0 if not.

abort() Abort the meas/process.

Data processing note:

Measurement (with processData=True) and relapy data first identify a clusters, using XRF parameters, next divides clusters into groups according to MeasParams.

17.3. Spectraimg data output

getFrameForEnergy(energyIndex, sumFrame, normalize, zoom) Gets image from selected energy range.
Returns the return code and the frame.

Returned frame is 2-dimimensional array with sizes of the sensor multiplied by the zoom factor.

energyIndex: Index of energy range. Ranges must be set using the **setMeasParams** method.

sumFrame: Gets summary frame (True) and ignore index or gets the single energy range (False).

normalize: Enable normalize of the image (True/False).

zoom: Zoom factor of sumFrame. Usable values are 1 and 2.
Single energies not zoomed, but allowed values are 1/2 too.

Example: `rc, frame = si.getFrameForEnergy(13, False, False, 1)`

getFrameForEnergyRange(energyFromIndex, energyToIndex, normalize) Gets image from selected energy range.
Returns the return code and the frame.

Returned frame is 2-dimimensional array with sizes of the sensor with sum from the selected ranges.

energyFromIndex: Index of first energy range. Ranges must be set using the **setMeasParams** method.

energyToIndex: Index of last energy range. Ranges must be set using the **setMeasParams** method.

normalize: Enable normalize of the image (True/False).

getGlobalSpectrum() Gets the global energy spectrum. Returns return code, the spectrum array and step size.
Array size and step size [keV] is depends on previous used **setMeasParams** method.

Example: `rc, spectrum, step = si.getGlobalSpectrum()`

getGlobalSpectrumInRect(left, top, right, bottom) Same us the **getGlobalSpectrum()** but in the rectangle
defined by corners positions.

Example: `rc, spectrum, step = si.getGlobalSpectrumInRect(x1, y1, x2, y2)`

Note: The smaller the rectangle, the more data is needed.

17.4. Spectraimg and data files

Output data files:

- saveSumFrame**(path, zoom, correct) Save all data to one text frame. Zoom 1/2/3. Sub-pixels correction True/False.
The txt ASCII matrix file with space separated decimal float numbers at lines.
- saveDataAsFramesToFile**(filePath, oneFile) Save all spectrum bins data to the frame file(s).
Multi-frame one file format can be used or more filenames.
- saveDataAsSpectrumToFile**(path) Save text file with tab-separated cols for all energy steps and lines for all pixels.
The step size [keV] and count depends on previous used **setMeasParams** method.

BSTG (Binary SeTTinGs and data) files:

Save and reuse a processed data: See the BSTG workflows in SI introduction [The Spectralmg object](#)

- loadFromFile**(path) Load the previous measured+processed data and device configuration from the BSTG file.
- saveToFile**(path) Save the measured and processed data and device configuration to the BSTG file.

(See [Save processed data to BSTG file for future use](#) and [Load and use previous saved BSTG data](#))

17.5. Spectralmg constants and error codes

```
// Contants:
#define SI_API_VERSION 1
#define SI_INVALID_HANDLE 0
#define SI_NO_DEVICE -1

// Error Codes:
#define SI_ERR_CANNOT_LOAD_PIXET -1
#define SI_ERR_PIXET_NOT_LOADED -2
#define SI_ERR_INVALID_HANDLE -3
#define SI_ERR_INVALID_DEVICE_INDEX -4

#define SI_ERR_INVALID_ARGUMENT -100
#define SI_ERR_CALIB_DIMENSION_MISMATCH -101
#define SI_ERR_CANNOT_OPEN_FILE -102
#define SI_ERR_CANNOT_READ_FRAME -103
#define SI_ERR_CANNOT_MEASURE -104
#define SI_ERR_LOCK_TIMEOUT -105
#define SI_ERR_DIMENSION_MISMATCH -106
#define SI_ERR_NO_DATA -107
#define SI_ERR_INVALID_RECT -108
#define SI_ERR_CANNOT_READ_FILE -109
#define SI_ERR_CANNOT_SAVE_FILE -110
```

For Pixet core constants see [Constants: Device and chip types](#) and nexts.

17.6. Spectralmg examples

17.6.1. Simple measuring and list of the spectrum (Tpx3 only)

```
import pypixet
import pypxproc

print("pixet core init...")
pypixet.start()
pixet=pypixet.pixet
devices = pixet.devicesByType(pixet.PX_DEVTYPE_TPX3)
dev = devices[0]

si = pypxproc.SpectraImaging(dev.asIDev())

print("Load calibration...")
rc = si.loadCalibrationFromDevice()
print("rc", rc , "0 is OK")

# setMeasParams(from, to, step, maskNP, doSPC)
si.setMeasParams(0, 300, 10, True, False)

print("Start measurement...")
# si.startMeasurement(acqTime (frame-only devs), measTime, outFile, processData)
rc = si.startMeasurement(1, 10, "", True)
print("rc", rc , "0 is OK")
print("measuring...")

while si.isRunning():
    pass # It is the simplest, but has twice the CPU usage than time.sleep(...)

print("Get global spectrum...")
rc, spect, step = si.getGlobalSpectrum()
print("rc={} Spectrum in {} chans, with step {} keV:".format(rc, len(spect), step))
print(spect)

pixet.exitPixet()
pypixet.exit()
```


17.6.2. Begin and end for online mode examples

```
import pypixet, pypxproc, time, sys, os
from saveBMP import *

def messageCb(error, msg): # =====
    print("** ErrCode:", error, "Message:", msg)

def progressCb(finished, progress): # =====
    if finished==0:
        if progressCb.prevProc == -1 or progressCb.prevProc < progress-10:
            print("\nProgress: {:.2f} %".format(progress), end=" ")
            progressCb.prevProc = progress
        else:
            sys.stdout.write('.')
            sys.stdout.flush()
    else:
        print("\nFinished")

progressCb.prevProc = -1

try: # create dirrectory for file output
    os.makedirs("out-files")
except FileExistsError:
    pass # already exists

print("pixet core init...")
pypixet.start()
pixet=pypixet.pixet
devices = pixet.devices()
if devices[0].fullName()=="FileDevice 0":
    print(" No devices connected")
    pixet.exitPixet()
    exit()

dev = devices[0]

si = pypxproc.SpectraImaging(dev.asIDev())

si.messageCallback = messageCb
si.progressCallback = progressCb

rc = si.loadCalibrationFromDevice()
print("Load calib, rc", rc)

# here place the working code ←-----

print("pixet core exit...")
pixet.exitPixet()
pypixet.exit()
```

17.6.3. Measuring and getFrameForEnergy

```
# (insert between the "Begin and end for online...")
# setMeasParams(from, to, step, maskNP, doSPC)
si.setMeasParams(0, 5000, 250, True, False)
# The clusters will be sorted into 250 keV wide bands, first at 0, last at 5000,
# noisy pixels masked, subpixel correction Off

print("Start measurement...")
# si.startMeasurement(acqTime (ignore at data-driven devs), measTime, outFile, processData)
rc = si.startMeasurement(1, 40, "", True)
print("rc", rc, "(0 is OK)")
print("measuring...")

while si.isRunning(): # wait for end of measuring and processing
    time.sleep(0.1)

print("Get spectrum...")
rc, spect, step = si.getGlobalSpectrum()
chans = len(spect)
print("rc={}. Spectrum in {} channels, with step {} keV:".format(rc, chans, step))
print(spect)
print()

print("Get frame for energy (sum frame)")
# getFrameForEnergy(energyIndex, sumFrame, normalize, zoom)
rc1, fr = si.getFrameForEnergy(2, True, False, 1) # get the sum frame
print(" rc", rc1, "(0 is OK)")

# saveWithLogNormFrameToFile(framesRGB, fName, fIdx=-1, desc="")
saveWithLogNormFrameToFile([fr, fr, fr], "out-files/SpectraImg-GFFE-sum")
# (save function from python examples environment: saveBMP.py)

for n in range(0, chans-2, 3):
    # generate triplets of a single channel frames to use in RGB images
    info = "E_{:.1f}-{:.1f}_keV".format(n*step, (n+2)*step)
    print("Get frame for energy: idx", n, info)
    # getFrameForEnergy(energyIndex, sumFrame, normalize, zoom)
    rc1, fb = si.getFrameForEnergy(n+0, False, False, 1)
    rc2, fg = si.getFrameForEnergy(n+1, False, False, 1)
    rc3, fr = si.getFrameForEnergy(n+2, False, False, 1)
    print(" rc", rc1, rc2, rc3, "(0 is OK)")

    # saveWithLogNormFrameToFile(framesRGB, fName, fIdx=-1, desc="")
    saveWithLogNormFrameToFile([fr, fg, fb], "out-files/SpectraImg-GFFE", n, info)
```

(this example is part of the **tpx3-spectrum-gFFE-bmp.py** example file)

17.6.4. Measuring and getFrameForEnergyRange

```
# (insert between the "Begin and end for online...")
# setMeasParams(from, to, step, maskNP, doSPC)
si.setMeasParams(0, 5120, 20, True, False)
# The clusters will be sorted into 20 keV wide bands, first at 0, last at 5120,
# noisy pixels masked, subpixel correction ON

print("Start measurement...")
#si.startMeasurement(acqTime (ignore at data-driven devs), measTime, outFile, processData)
rc = si.startMeasurement(1, 40, "", True)
print("rc", rc, "(0 is OK)")
print("measuring...")

while si.isRunning():
    time.sleep(0.1)

for n in range(0, chans-15, 15):
    print("Get frames for energy range:", n)
    # getFrameForEnergyRange(energyFromIndex, energyToIndex, normalize)
    rc1, fb = si.getFrameForEnergyRange(n+0, n+4, True)
    rc2, fg = si.getFrameForEnergyRange(n+5, n+9, True)
    rc3, fr = si.getFrameForEnergyRange(n+10, n+14, True)
    print(" rc", rc1, rc2, rc3, "(0 is OK)")

    # saveWithLogNormFrameToFile(framesRGB, fName, fIdx=-1)
    saveWithLogNormFrameToFile([fr, fg, fb], "out-files/SpectraImg-gFfER", n)
    # (save function from python examples environment: saveBMP.py)
```

(this example is part of the **tpx3-spectrum-gFfER-bmp.py** example file, adapted for online processing)

17.6.5. Offline data processing

```
import pypixet, pypxproc, sys, time, os
from saveBMP import *

def messageCb(error, msg): # =====
    print("** Message callback: ErrCode:", error, "Message:", msg)

def progressCb(finished, progress): # =====
    if finished==0:
        if progressCb.prevProc == -1 or progressCb.prevProc < progress-10:
            print("\nProgress: {:.2f} %".format(progress), end=" ")
            progressCb.prevProc = progress
        else:
            sys.stdout.write('.')
            sys.stdout.flush()
    else:
        print("\nFinished")

progressCb.prevProc = -1

print("pixet core init...")
pypixet.start()
pixet=pypixet.pixet
devices = pixet.devices()
si = pypxproc.SpectraImaging(dev.asIDev())

si.messageCallback = messageCb
si.progressCallback = progressCb

print("Load calibration...")
rc = si.loadCalibrationFromDevice()
print("rc", rc, "(0 is OK)")

# setMeasParams(from, to, step, maskNP, doSPC)
si.setMeasParams(0, 256*15, 15, True, True) # 0-3840 keV, step 15

print("replay data start")
# replayData(pathIn, pathOut, reserved)
rc = si.replayData("out-files/SpectraImg-data.t3pa", "", 0)
print("rc", rc, "(0 is OK)")
print("processing...")

while si.isRunning():
    time.sleep(0.1)

# Now you can use processed data by si.getGlobalSpectrum, getFrameForEnergy, ...
```

Notes:

The device is not required for offline processing. In this example it's only as source of calibration data.

Alternative start with no device is:

```
si = pypxproc.SpectraImaging()
print("Load calibration...")
rc = si.loadCalibrationFromFiles("config.xml") # the device config file
print("rc", rc, "(0 is OK)")
```

You can also use

```
si.loadCalibrationFromFiles("cal_a.txt|cal_b.txt|cal_c.txt|cal_t.txt")
```

to load only ABCT configuration data instead of whole config.xml.

(The offline data processing is used in more Spectralmg examples: **tpx3-spectrum-gFfER-bmp.py**, **tpx3-spectrum-gGSiR-bmp.py**, ...)

17.6.6. Get a data for the offline processing

```
import pypixet, os

try: # create dirrectory for file output
    os.makedirs("out-files")
except FileExistsError:
    pass # already exists

print("pixet core init...")
pypixet.start()
pixet=pypixet.pixet
devices = pixet.devices()
dev = devices[0]
dev.setOperationMode(pixet.PX_TPX3_OPM_TOATOT)

dataFileName = "out-files/example-data.t3pa"

print("doAdvancedAcquisition...")
#doAdvancedAcquisition(acqCount, acqTime, acqType, acqMode, fileType, fileFlags, fName)
rc = dev.doAdvancedAcquisition(1, acqTime, pixet.PX_ACQTYPE_DATADRIVEN,
                               pixet.PX_ACQMODE_NORMAL, pixet.PX_FTYPE_AUTODETECT, 0, dataFileName)
print(" rc", rc, "(0 is OK)")

print("Exit pixet...")
pixet.exitPixet()
pypixet.exit()
```

Note: If your device not support data-driven mode, frame mode and frame files must be used. Set short acqTime to prevent clusters overwriting and more acqCount to obtain sufficient amount of data.

17.6.7. Save processed data to BSTG file for future use

```
# (insert between the "Begin and end for online...")
# setMeasParams(from, to, step, maskNP, doSPC)
si.setMeasParams(0, 5000, 250, True, False)

print("Start measurement...")
#si.startMeasurement(acqTime (ignore at data-driven devs), measTime, outFile, processData)
rc = si.startMeasurement(1, 40, "", True)
print("rc", rc, "(0 is OK)")
print("measuring...")

while si.isRunning(): # wait for end of measuring and processing
    time.sleep(0.1)

print("Save bstg...")
rc = si.saveToFile("out-files/SpectraImg.bstg")
print("rc", rc, "(0 is OK)")
```

17.6.8. Load and use previous saved BSTG data

```
import pypxproc
from saveBMP import *

print("Starting pypxproc.SpectraImaging...")
si = pypxproc.SpectraImaging()

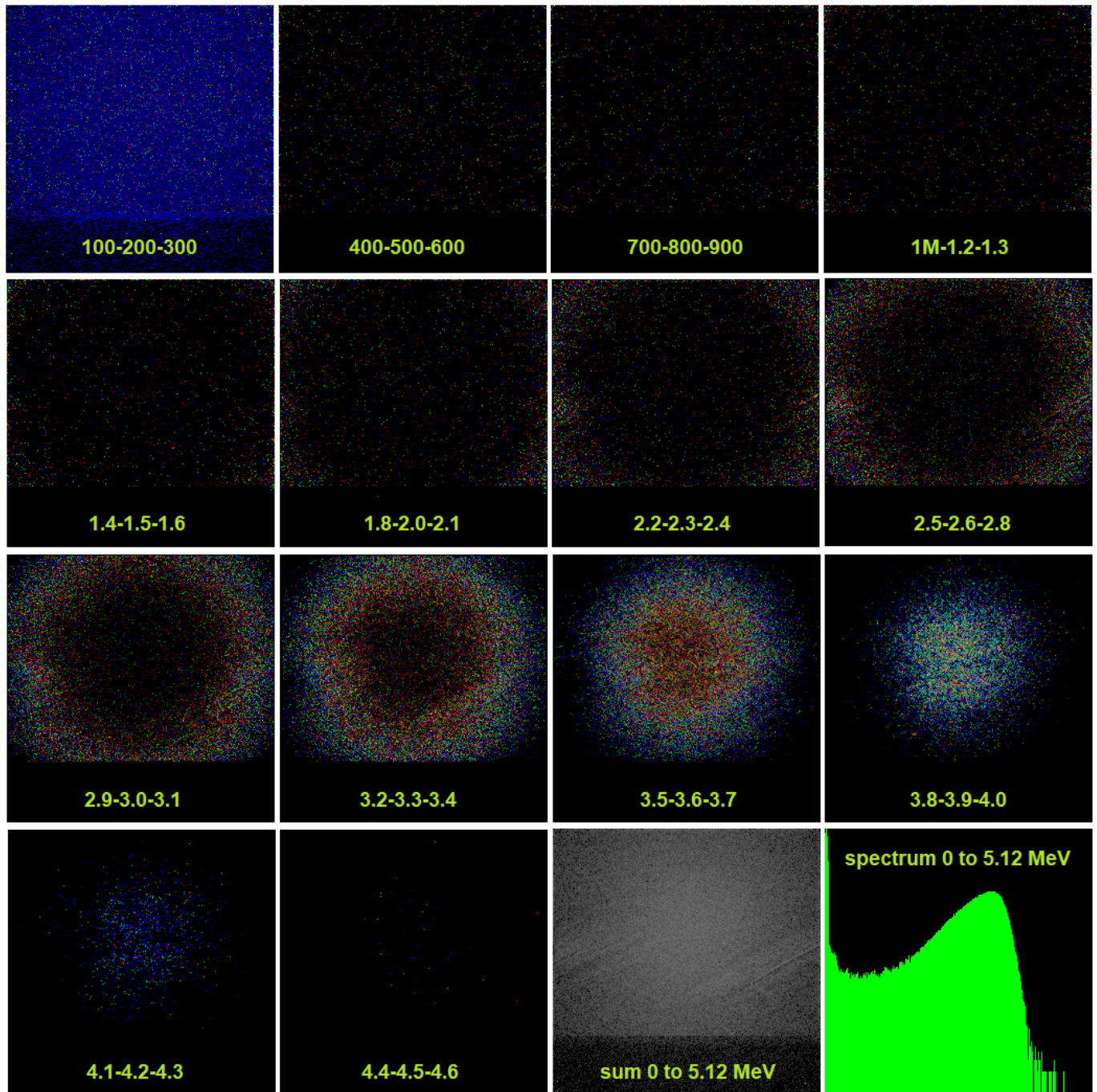
print("Load bstg...")
rc = si.loadFromFile("out-files/SpectraImg.bstg")
print("rc", rc, "(0 is OK)")

# Now you can use processed data as it was saved

print("Get spectrum...")
rc, spect, step = si.getGlobalSpectrum()
chans = len(spect)
print("rc={}. Spectrum in {} channels, with step {} keV:".format(rc, chans, step))
print(spect)
print()

print("pixet core exit...")
pixet.exitPixet()
pypixet.exit()
```

17.6.9. Measuring examples results (getFrameForEnergyRange)



²⁴¹Am from smoke detector, located 3 mm above the chip, MinipixTpx3, CdTe 2 mm, acq. time 45 seconds.

Settings used: `setMeasParams(0, 5120, 20, True, False) # (from, to, step, maskNP, doSPC)`

Each RGB color channel in images was generated from sum of 5 energy ranges (100 keV), using the **getFrameForEnergyRange** method, $\log_2(\text{val}+0.5)$ applied and normalized to 0-255. Order is blue-green-red. The blue at the first image is gamma byproduct 59.5 keV and noise in the first band. The images with MeV ranges shows a 5.48 MeV alpha particles attenuated in air.

Summary frame shows all hits in the set range. It was get using the **getFrameForEnergy** method with `sumFrame=True`. A “scratches” on the picture: Spectralmg is very sensitive to small differences in px sensitivity.

The spectrum on last image was get using the **getGlobalSpectrum** method and processed like us pixels data, include using \log_2 . A gaps on the right are caused by no hits.

18. Pixet plugin and associated objects

The plugin objects can control plugins of the Pixet GUI from the Pixet Python interpreter.

Example for start using plugin in the Pixet interpreter:

```
import pymeasutils
idev = pixet.devices()[0].asIDev()
mu = pixet.plugin("measutils").privateInterface()
```

18.1. The IDevControl object

This object can control in the Pixet program the **Device control GUI window** of the device selected by **IDev**.

showFrame(idev, frame)	Shows the frame.
setOutputFile(idev, fileName, outputEnabled)	Sets the output file name and enables/disables file output.
setAcqParams(idev, acqCount, acqTime)	Sets the acquisition params.
setAcqType(idev, acqType)	Sets acq. type (see Constants: Acquisition types and modes, incl. triggers).
setAcqMode(idev, mode)	Sets the acq. mode (see Constants: Operation modes).
setRepeat(idev, repeatCount, repeatEnabled)	Sets and enables/disables repeats for repeated or integral acq.
start(idev)	Starts the acquisition.
stop(idev)	Stops the acquisition.
isRunning(idev)	Returns 1 if acquisition is running or 0 if not.
setColorMap(idev, index)	Sets index of the color map used in the frame view.
setRange(idev, min, max)	Sets range of values in the frame view.
setCurrentSubFrame(idev, index)	Switchs view to the indexed subframe.

Example:

```
import pydevcontrol

# get pointer to devcontrol object
devctrl = pixet.plugin("devcontrol").privateInterface()

# load some frame from disk and show it in first device preview window:
dev = pixet.devices()[0]
frame = pixet.dataMgr().loadFrame("/tmp/frame.pmf", 0);
devctrl.showFrame(dev.asIDev(), frame)
frame.destroy()

# change acquisition parameters in main window - acqCount=100, acqTime=0.5 s:
devctrl.setAcqParams(dev.asIDev(), 100, 0.5)

# simulate clicking on start button on main window:
devctrl.start(dev.asIDev())

# simulate clicking on stop button on main window:
devctrl.stop(dev.asIDev())
```


18.2. The IMmeasutils object

The IMmeasutils controlling the **measutils plugin** in the Pixet GUI. This is alternative access to the Spectral Imaging. (see [The Spectralimg object](#) for details, it is very similar)

siSetParameters (idev, spectraFrom, spectraTo, binCount, subPixFactor, correction)	Sets spectral imaging pars. See Spectraimg measurement and related methods
siSetMaskNoisyPixels (idev, mask)	Sets mask of noisy pixels.
siStart (idev, acqTime, measTime, outputFilePath, processData)	Starts the measurement.
siStop (idev)	Stops the measurement.
siIsRunning (idev)	Returns 1 if processing is running or 0 if not.
siReplayData (idev, filePath, outputFilePath)	Starts processing of data in the file.
siSaveFrames (idev, filePath, int flags)	Save frames from processed data. Flags: 0 multiple files, 1 single
siSaveSumFrame (idev, filePath, zoomFactor, correction)	Saves the sum frame.
siSaveSpectra (idev, filePath)	Saves spectrum to text file. Every pixel has a line with spectrum.

Example:

```
import pymeasutils
import time

idev = pixet.devices()[0].asIDev()
mu = pixet.plugin("measutils").privateInterface()
print(mu)

# sets spectra from=0, to 300, bin count=300, subpixel factor=1, correction True:
mu.siSetParameters(idev, 0, 300, 300, 1, True)

# starts the measurement and save data to file
mu.siStart(idev, 0.5, 3, "/tmp/test.t3r", False)

# wait till measurement finishes:
isRunning = True
while isRunning:
    isRunning = mu.siIsRunning(idev)
    print(isRunning)
    time.sleep(0.5)

#mu.siStop(idev)
#mu.siSaveSumFrame(idev, "/d/xs/tmpdata/sum.txt", 1, False)
#mu.siSaveSpectra(idev, "/d/xs/tmpdata/spectra.txt")
#mu.siSaveFrames(idev, "/d/xs/tmpdata/frames.pmf", 1)
# pmf flags: 0 - multiple files, 1 - frames in single file
```

18.3. The IClustering object

The IClustering object controlling the **Clustering plugin** in the Pixet GUI.

(see [The Clustering object and related](#) for details, it is very similar)

start (idev, acqTime, measTime, outputPath, processData)	Starts the measurement.
stop (idev)	Stops the measurement.
isRunning (idev)	Returns 1 if processing is running or 0 if not.
replayData (idev, filePath)	Starts processing of data in the file.
saveFrames (idev, frameIndex, filePath, flags)	Save frames from processed data. Flags: 0 frames, 1 spectrum
clearData (idev)	Clears processed data. Note: Next processing adds new data to end of previous.
setLogOKClusterLogFile (idev, clusterLogFilePath)	Sets file path for saving OK clusters.
setLogAllClusterLogFile (idev, clusterLogFilePath)	Sets file path for saving all clusters.
setLogBadClusterLogFile (idev, clusterLogFilePath)	Sets file path for saving bad clusters.

Example:

```
import pyclustering

# get pointer to devcontrol object
clustering = pixet.plugin("clustering").privateInterface()

# replay data with clustering from first device:
dev = pixet.devices()[0]
clustering.setLogAllClusterLogFile(dev.asIDev(), "/tmp/output.clog")
#clustering.setLogOKClusterLogFile(dev.asIDev(), "/tmp/output.clog")
#clustering.setLogBadClusterLogFile(dev.asIDev(), "/tmp/output.clog")

#clustering.replayData(dev.asIDev(), "/tmp/data.t3pa")
#clustering.replayData(dev.asIDev(), "/tmp/data.t3r")
clustering.replayData(dev.asIDev(), "/tmp/somefile.clog")
```

19. The pygui object

This object provides a graphical environment. If the script using it, must be run from the Pixet program's Python plugin. The pygui using .UI files with XML format, which is compatible with UI files from the Qt Creator. Graffical items can be also generated by code, withour an UI file.

loadFromUiString(xmlstring) Load window or widget definition from XML string. Returns the [Window object](#).
loadFromUiFile(uiFilePath) Load window or widget definition from XML file. Returns the Window object.
MainWindow() Create the new Window object for future use with code-generated components.

showMessage(title, message) Shows the system messagebox, info type. Returns none.
showWarning(title, message) Shows the system messagebox, warning type. Returns none.
showError(title, message) Shows the system messagebox, error type. Returns none.
showQuestion(title, message) Shows the system messagebox, question Yes/No type. Returns 1 or 0.

showItemSelection(title, message, itemsArray, defaultIndex, editable)
 Shows the Select Item dialog to select from array of string items.
 User can optionally edit selected item.
 Returns the selected string or default string item if the Cancel clicked.
inputText(title, label) Shows the Input type dialog. Returns string or None if the Cancel clicked.

getOpenFileName(name, filter) Shows the File Open dialog. Returns the file path or None if the Cancel clicked.
getExistingDirectory(name) Shows the Select Folder dialog. Returns the file path or None if the Cancel clicked.
getSaveFileName(name, filter) Shows the Save as... dialog. Returns the file path or None if the Cancel clicked.
getSaveFileNameAndFilter(name, filter) Some as getSaveFileName, but returns array with path and selected filter.

File filters variants:

1. Single filter: Only filtering string "*.txt"
2. Single filter with description: Description (filter) "Text files (*.txt)"
3. Multiple filters in single select item: Space separated list "*.txt *.csv"
4. Multiple filter items to select: List separated by the ;; (double ;) "*.txt;*.csv"
5. Combined example: "Images (*.png *.xpm *.jpg);;Text files (*.txt);;XML files (*.xml)"

Examples:

```
import pygui

pygui.showMessage("Window title", "Message text")
pygui.showWarning("Window title", "Message text")
pygui.showError("Window title", "Message text")
print(pygui.showQuestion("Window title", "Question text"))
print(pygui.showItemSelection("Title", "Message", ["item0", "item1"], 0, False))
print(pygui.showItemSelection("Title", "Message", ["item0", "item1"], 0, True))

print(pygui.inputText("Window title", "Question text"))
print(pygui.getOpenFileName("", "*.txt"))
print(pygui.getExistingDirectory(""))
print(pygui.getSaveFileName("test.txt", "*.txt"))
print(pygui.getSaveFileNameAndFilter("test.txt", "*.txt;*.csv"))

window = pygui.MainWindow()
print(window)
window.show()
```

20. The Window and associated objects

This is graphical window object created by the [pygui](#) object, using methods **loadFromUiString**, **loadFromUiFile** or empty object from **MainWindow**.

The Window methods:

show()	Shows the window.
hide()	Hides the window.
setTitle(title)	Sets title of the window.
setPosition(x, y)	Sets position of the window.
setSize(width, height)	Sets size of the window.
processEvents()	Process waiting events associated with the window.
setLayout(layout)	Sets layout of the window, using the GridLayout object.
newGridLayout()	Creates and returns a GridLayout object.
Label(title)	Creates and returns a Label object.
Button(title)	Creates and returns a PushButton object.
ToolButton()	Creates and returns a ToolButton object.
CheckBox(title)	Creates and returns a CheckBox object.
RadioButton()	Creates and returns a RadioButton object.
ComboBox()	Creates and returns a ComboBox object.
SpinBox()	Creates and returns a SpinBox object.
DoubleSpinBox()	Creates and returns a DoubleSpinBox object.
LineEdit()	Creates and returns a LineEdit object.
TextEdit()	Creates and returns a TextEdit object.
Slider()	Creates and returns a Slider object.
ProgressBar()	Creates and returns a ProgressBar object.
PropertyTreeView()	Creates and returns a PropertyTreeView object.
ListWidget()	Creates and returns a ListWidget object.
Widget()	Creates and returns a Widget object.
GroupBox()	Creates and returns a GroupBox object.
MpxFrame()	Creates and returns a MpxFrame object.
MpxFramePanel()	Creates and returns a MpxFramePanel object.
Plot()	Creates and returns a Plot object.

Using of the Window object via **pygui.MainWindow()**:

1. Create the window using some as **window = pygui.MainWindow()**
2. Create the objects that you plan to use in the window using some as **lab = window.Label("Text")**
3. Create the GridLayout using some as **gl = window.newGridLayout()**
4. Add objects created in 2 to layout using some as **gl.addWidget(lab, 0, 0, 1, 1)**
5. Set your layout to the window using some as **window.setLayout(gl)**
6. Show the window using some as **window.show()**

The GroupBox and Widget objects can be used similarly as a Window:

Create new graphical objects using methods of the Window object, create the GroupBox, create GridLayout, add objects to the GridLayout and set GridLayout to the GroupBox. Now you can use the GroupBox as other objects in the Window.

20.1. The GridLayout object and its methods

The GridLayout object are using to set layout of the objects in the Window object.

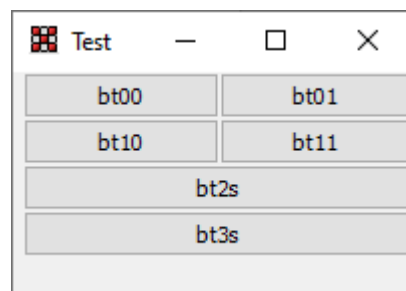
setSpacing(pixels)	Sets pixel spacing size in the grid.
setVerticalSpacing(pixels)	Sets vertical spacing between objects in the GridLayout.
setHorizontalSpacing(pixels)	Sets horizontal spacing between objects in the GridLayout.
setContentsMargins(left, top, right, bottom)	Sets margins around all of the GridLayout's contents.
setRowStretch(row, stretch)	Sets stretch factor* of the row.
setColumnStretch(column, stretch)	Sets stretch factor* of the column.
spacing()	Returns spacing in the grid.
verticalSpacing()	Returns vertical spacing in the grid.
horizontalSpacing()	Returns horizontal spacing in the grid.
addWidget(widget, row, col, rowspan, colspan)	Adds widget-type object to the GridLayout to the row/col position. The rowSpan and colSpan is number of coupled rows or columns occupied by the object.
rowCount()	Returns count of rows in the grid.
columnCount()	Returns count of columns in the grid.

* The **stretch factor** is relative to the other columns in this grid. Columns/rows with a higher stretch factor take more of the available space.

Example:

```
import pygui
```

```
window = pygui.MainWindow()
window.setTitle("Test")
bt00 = window.Button("bt00")
bt01 = window.Button("bt01")
bt10 = window.Button("bt10")
bt11 = window.Button("bt11")
bt2s = window.Button("bt2s")
bt3s = window.Button("bt3s")
gl = window.newGridLayout()
gl.addWidget(bt00, 0, 0, 1, 1)
gl.addWidget(bt01, 0, 1, 1, 1)
gl.addWidget(bt10, 1, 0, 1, 1)
gl.addWidget(bt11, 1, 1, 1, 1)
gl.addWidget(bt2s, 2, 0, 1, 2)
gl.addWidget(bt3s, 3, 0, 1, 2)
gl.setSpacing(0)
gl.setContentsMargins(5, 0, 0, 20)
window.setLayout(gl)
window.show()
```



20.2. The Widget, TabWidget and GroupBox objects

Widget and Groupbox are an objects for graphical arrangement. GroupBox is visible and can have label, Widget is invisible. The TabWidget can be used to switch Widgets by a tabs.

You can use hierarchical arrangement:

1. Create a GridLayout and a Widget/Groupbox.
2. Insert some objects to the GridLayout using the **addWidget** method.
3. Set the GridLayout to your Widget/Groupbox using the **setLayout** method.
4. Insert your Widget/Groupbox to the parent layout using the **addWidget** method.

20.2.1. The Widget object and its methods

Create: **window.Widget()**

setPosition(x, y)	Sets position of the widget in the parent area.
setSize(width, height)	Sets size of the widget.
setLayout(layout)	Sets layout of the widget, using a GridLayout object.
Show(visible)	Shows/hides the widget.

20.2.2. The GroupBox object and its methods

Create: **window.GroupBox()**

setPosition(x, y)	Sets position of the widget in the parent area.
setSize(width, height)	Sets size of the widget.
setLayout(layout)	Sets layout of the widget, using a GridLayout object.
Show(visible)	Shows/hides the widget.
setTitle(text)	Sets the label text.
setStyleSheet(styleSheetText)	Sets style of the label text. Example: box.setStyleSheet("color: red")

20.2.3. The TabWidget object and its methods

Create: (only in UI file or string)

addTab(widget, name)	Adds a new tab with name and associated with defined widget.
removeTab(index)	Removes the indexed tab.
setCurrentIndex(index)	Changes active tab to index.
currentIndex()	Returns index of the active tab.
setTabText(index, text)	Sets text of the indexed tab.
setEnabled(en)/isEnabled()	Enables/disables and returns enable state.
setMinimumSize/setMaximumSize	Sets size limits (width, height) for window arrangement and resize.

It has the “**changed**” event. Occurs on change of the active tab index.

20.3. The PropertyTreeView object and its methods

PropertyTreeView from the QT. Tree contains string-named groups. Groups contains string-named properties. Properties can be strings, ints, doubles, bools and lists. Properties can be editable or view only. Numeric values have an allowed range min-max, doubles also have precision. Values can be added using **add...** methods, changed using **set...** methods, reads using the **getProperty** or you can clear all using the **clear** method.

Create: **window.PropertyTreeView()**

addInt(group, name, value, min, max, isEditable) Adds int-type property with allow min/max range.

addDouble(group, name, value, min, max, prec, isEditable) Adds double-type property with min/max, precision.

addBool(group, name, value, isEditable) Adds bool-type property.

addString(group, name, value, isEditable) Adds a string-type property.

addList(group, name, [values], isEditable) Adds a list-type property.

getProperty(group, name) Returns value of the property.

setInt(group, name, value) Changes the int-type property value.

setDouble(group, name, value) Changes the double-type property value.

setBool(group, name, value) Changes the bool-type property value.

setString(group, name, value) Changes the string-type property value.

setList(group, name, [values]) Changes list in the list-type property.

setListSelectedText(group, name, selValue) Changes the list-type property selected value.

expandAll(isExpanded) Expands/collapses all groups.

clear() Clears whole the PropertyTreeView.

setColumnWidth(col, width) Sets width of the column.

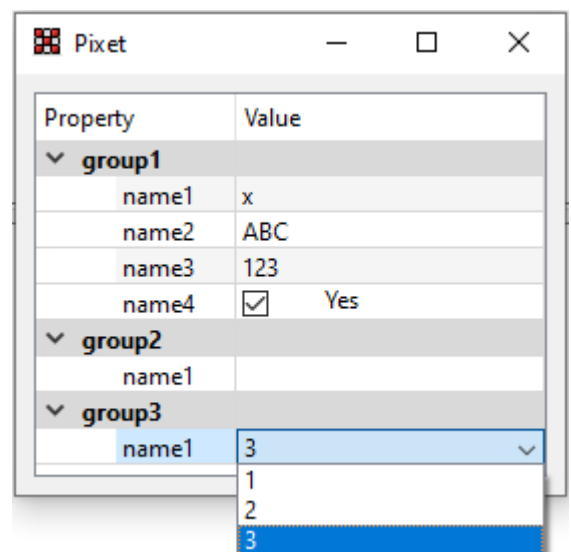
setEnabled(en)/isEnabled() Enables/disables and returns enable state.

setMinimumSize/setMaximumSize Sets size limits (width, height) for window arrangement and resize.

List-type properties notes:

- The essence of the list-type value is which list element is selected.
- New list-type property has no selected item and the **getProperty** returns nothing.
- The **setListSelectedText** can set string that is not in the list and this disappears if editable value is clicked.

```
window = pygui.MainWindow()
ptv = window.PropertyTreeView()
ptv.addList("group1", "name1", ["a", "b", "c"], 1)
ptv.addList("group2", "name1", ["A", "B", "C"], 1)
ptv.addList("group3", "name1", ["1", "2", "3"], 1)
ptv.addString("group1", "name2", "ABC", 1)
ptv.addInt("group1", "name3", 123, 0, 200, 1)
ptv.addBool("group1", "name4", 1, 1)
ptv.setListSelectedText("group1", "name1", "x")
ptv.setListSelectedText("group3", "name1", "3")
ptv.expandAll(1)
gl = window.newGridLayout()
gl.addWidget(tst, 0, 0, 1, 1)
window.setLayout(gl)
window.show()
```



20.4. The MpxFrame object and its methods

This is the sizable viewer of detector frame data with color range description bar and position/value tooltip.

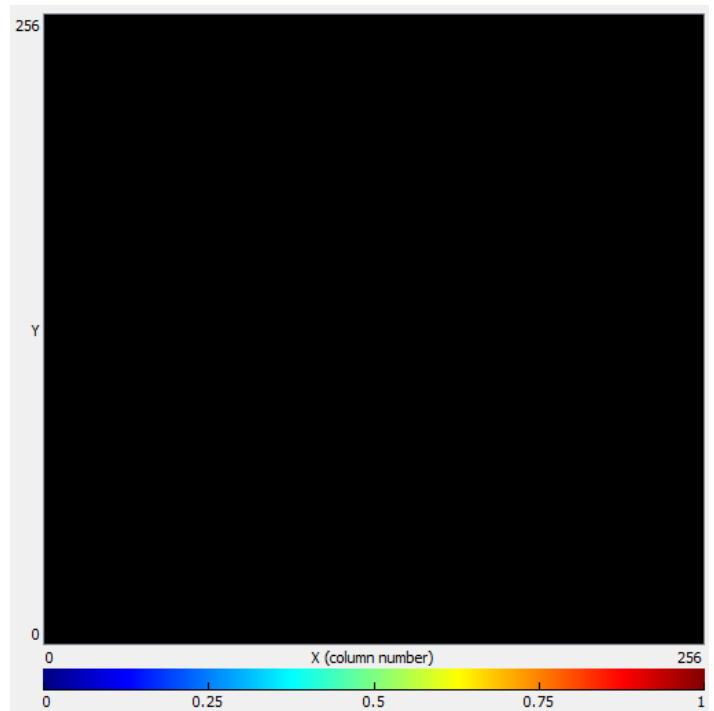
Create: **window.MpxFrame()**

showColorBar (show)	Show/hide color range description bar.
showAxis (show)	Show/hide axis.
showGrid (show)	Show/hide pixel grid (usesfull with high zoom).
keepAspectRatio (keep)	Enable/disable keep aspect ratio while resizing window.
setUnderWarning (show)	Enable/disable
setOverWarning (show)	Enable/disable
setMirrored (isMirrored)	Mirror/don't mirror the image.
setRotation (rotation)	Rotate the image.
setColorMap (colorMapIndex)	Switch color map. Range 0 (grayscale), 1 to 8 (colorized).
setRange (min, max)	Sets range for colorizing.
range ()	Returns list with min and max from colorizing range.
setData ([data], width, height)	Sets the image data.
setEnabled (en)/ isEnabled ()	Enable/disable and returns enable state.
setMinimumSize / setMaximumSize	Sets size limits (width, height) for window arrangement and resize.

Events:

clicked	Occurs at mouse button press. Callback parameters: (x, y, button)
selRectsChanged	Occurs after moving mouse while Ctrl hold. Parameters: (list of [x, y, width, height])
zoomRectChanged	Occurs after moving mouse while button hold. Parameters: (single [x, y, width, height])

MpxFrame with ColorMap 1



20.4.1. Timepix3 measuring with MpxFrame widget example

```

import pygui
devs = pixet.devicesByType(pixet.PX_DEVTYPE_TPX3)

def btnStart_Clicked(): # =====
    if len(devs)>0:
        btnStart.setEnabled(0)
        sbAcqTime.setEnabled(0)
        lab0.setText("Working...")
        lab1.setText("")
        window.processEvents()
        acqSimple(sbAcqTime.value(), cmbOPM.currentIndex())
        btnStart.setEnabled(1)
        sbAcqTime.setEnabled(1)
    else:
        pygui.showError("Error", "No devices found")

def frameInfo(fr): # =====
    return "Subframe {}: minNZ {}, max {}, hits {}".format(fr.frameName(),
fr.minNonZero(), fr.max(), fr.nonZeroCount())

def acqSimple(time, opm): # =====
    print("acqSimple %i sec, opm %i -----" % (time, opm))
    dev.setOperationMode(opm)
    print("dev.doSimpleAcquisition - start")
    rc = dev.doSimpleAcquisition(1, time, pixet.PX_FTYPE_AUTODETECT, "")
    print("dev.doSimpleAcquisition - end %i" % rc)
    if rc!=0:
        lab0.setText("Error in acq: %i" % rc)
        return

    print("acqFrameCount %i" % dev.acqFrameCount())
    frame = dev.acqFrameRefInc(0)
    print("frame.frameCount: %i subFrameCount %i" % (frame.frameCount(),
frame.subFrameCount()))
    data = frame.data()
    if frame.subFrameCount()>0:
        f0=frame.subFrames()[0]
        lab0.setText(frameInfo(f0))
        print(lab0.text())
        mpfr0.setRange(f0.min(), f0.max())
        mpfr0.setData(f0.data(), 256, 256)
    if frame.subFrameCount()>1:
        f1=frame.subFrames()[1]
        lab1.setText(frameInfo(f1))
        print(lab1.text())
        mpfr1.setRange(f1.min(), f1.max())

```

```

        mpfr1.setData(f1.data(), 256, 256)
    frame.destroy()
    window.processEvents()
# def acqSimple

# =====
if len(devs)>0:
    dev = devs[0]
    print(dev)
    pars = dev.parameters()
    tcpu = pars.get("TemperatureCpu").getDouble()
    tchip = pars.get("TemperatureChip").getDouble()
    fw = pars.get("FirmwareCpu").getString()
    print("CPU Temp: %f, Chip Temp: %f, FirmwareCpu: %s" % (tcpu, tchip, fw))
    print("=====")

    print("dev.loadConfigFromDevice %i" % dev.loadConfigFromDevice())
else:
    print("No devices found")

window = pygui.MainWindow()
window.setTitle("Timepix3 test")
mpfr0 = window.MpxFrame()
mpfr1 = window.MpxFrame()
mpfr0.setMinimumSize(512, 512)
mpfr1.setMinimumSize(512, 512)
mpfr0.setColorMap(1)
mpfr1.setColorMap(1)
lab0 = window.Label("Left frame: No data")
lab1 = window.Label("Right frame: No data")
btnStart = window.Button("Start acq")
btnStart.clicked = btnStart_Clicked
labNone1 = window.Label("")
labAcqTime = window.Label("Acq. time [sec]:")
sbAcqTime = window.DoubleSpinBox()
sbAcqTime.setMaximum(1000000)
sbAcqTime.setValue(5)
labNone2 = window.Label("")
labOPM = window.Label("Operation mode:")
cmbOPM = window.ComboBox()
cmbOPM.setItems(["PX_TPX3_OPM_TOATOT", "PX_TPX3_OPM_TOA", "PX_TPX3_OPM_EVENT_ITOT"])

glwin = window.newGridLayout() # layout of the window
glgr = window.newGridLayout() # layout of groupbox
gr = window.GroupBox()
gr.setLayout(glgr)

glwin.addWidget(mpfr0, 0, 0, 1, 1)
glwin.addWidget(mpfr1, 0, 1, 1, 1)

```

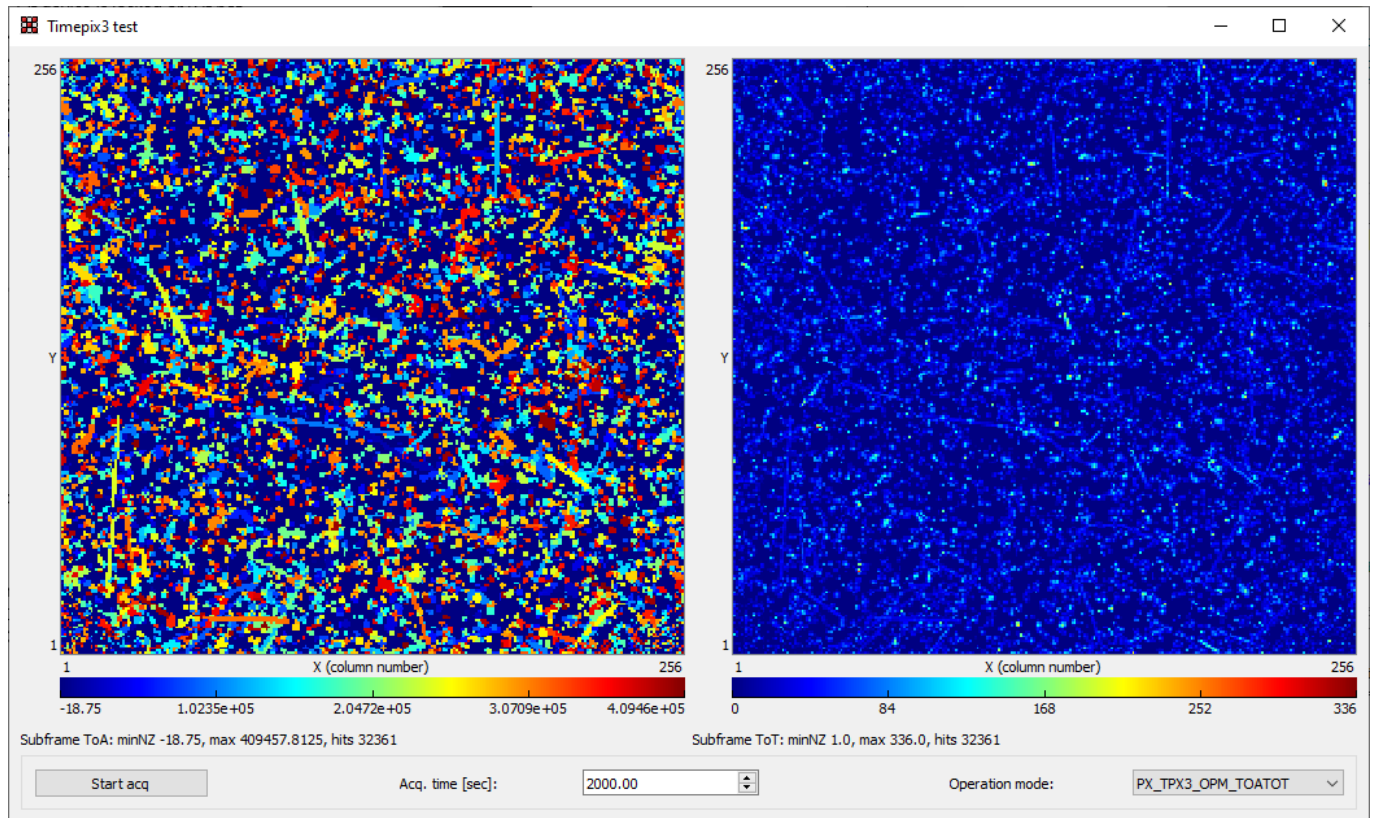
```

glwin.addWidget(lab0, 1, 0, 1, 1)
glwin.addWidget(lab1, 1, 1, 1, 1)
glwin.addWidget(gr, 2, 0, 1, 2)

glgr.addWidget(btnStart, 0, 0, 1, 1)
glgr.addWidget(labNone1, 0, 1, 1, 1)
glgr.addWidget(labAcqTime, 0, 2, 1, 1)
glgr.addWidget(sbAcqTime, 0, 3, 1, 1)
glgr.addWidget(labNone2, 0, 4, 1, 1)
glgr.addWidget(labOPM, 0, 5, 1, 1)
glgr.addWidget(cmbOPM, 0, 6, 1, 1)

window.setLayout(glwin)
window.show()

```



20.5. The MpxFramePanel object and its methods

Graphical object like as MpxFrame, but with panel containing usesfull controls.

Create: **window.MpxFramePanel()**

Methods different from MpxFrame:

showSaveButton(show)	Show/hide the "Save image" button.
showGridButton(show)	Show/hide the "Show grid" button.
showRotateButton(show)	Show/hide the "Rotate clockwise" button.
showAutoRangeButton(show)	Show/hide the "Auto range" button.
showColorMapButton(show)	Show/hide the "Color map" button.
showUnderButton(show)	Show/hide the "Under warming" button.
showOverButton(show)	Show/hide the "Over warming" button.
setAutoRange(isAutoRange)	Enable/disable auto range for image colorizing.

Methods identical with MpxFrame:

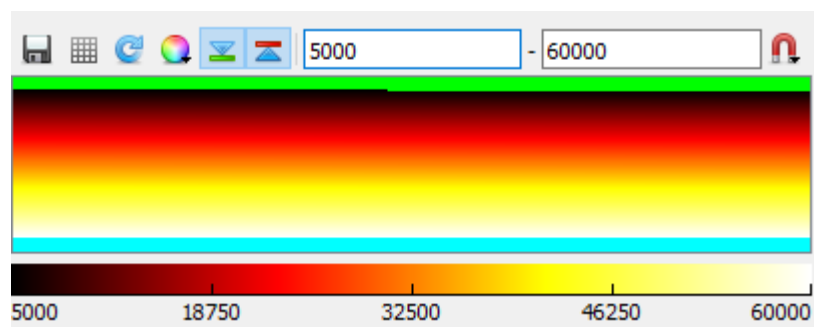
showColorBar(show)	Show/hide color range description bar.
showAxis(show)	Show/hide axis.
showGrid(show)	Show/hide pixel grid (usesfull with high zoom).
keepAspectRatio(keep)	Enable/disable keep aspect ratio while resizing window.
setUnderWarning(show)	Enable/disable
setOverWarning(show)	Enable/disable
setMirrored(isMirrored)	Mirror/don't mirror the image.
setRotation(rotation)	Rotate the image.
setColorMap(colorMapIndex)	Switch color map. Range 0 (grayscale), 1 to 8 (colorized).
setRange(min, max)	Sets range for colorizing.
range()	Returns list with min and max from colorizing range.
setData([data], width, height)	Sets the image data.
setEnabled(en)/isEnabled()	Enable/disable and returns enable state.
setMinimumSize/setMaximumSize	Sets size limits (width, height) for window arrangement and resize.

Events (identical with MpxFrame):

clicked	Occurs at mouse button press. Callback parameters: (x, y, button)
selRectsChanged	Occurs after moving mouse while Ctrl hold. Parameters: (list of [x, y, width, height])
zoomRectChanged	Occurs after moving mouse while button hold. Parameters: (single [x, y, width, height])

Range and under/over warming example:

The MpxFramePanel is filled with 0-65535 values



20.6. The Plot object and its methods

Graphical object designed to showing a graphs. Supports layers, zooming, autosize, highlights.

Plot have a layers.

Use **addLayer()** for add next layer and **clear()** to delete all except 1 layers.

Layer have a Lines, BarLines, Scatters and Highlights.

Use **addLine(layerIdx)** for add line to the indexed layer.

Line have a pixels coupled to continuous line. Whole line have one color.

Use **addPoint(layerIdx, lineIdx, x, y)** for add point or **replacePoint** to change Y value.

Or use **setLineData(layerIdx, lineIdx, [xs], [ys])** to set all line points.

BarLine is some as Line, indexed as Line, but area under whole graph is filled.

Maintaining of a BarLine is some as for Line.

Scatter is some as Line, but points are separated and can have various colors using colorized "Z axis".

Use **addScatterPoint(layerIdx, scatterIdx, x, y, z, repaint)** for add point to the scatter.

Highlight is highlighted area on the layer. Have a HighlightRect.

Use **setHighlightRect(layerIdx, highlightIdx, x1, x2, y1, y2)** for define area.

Example:

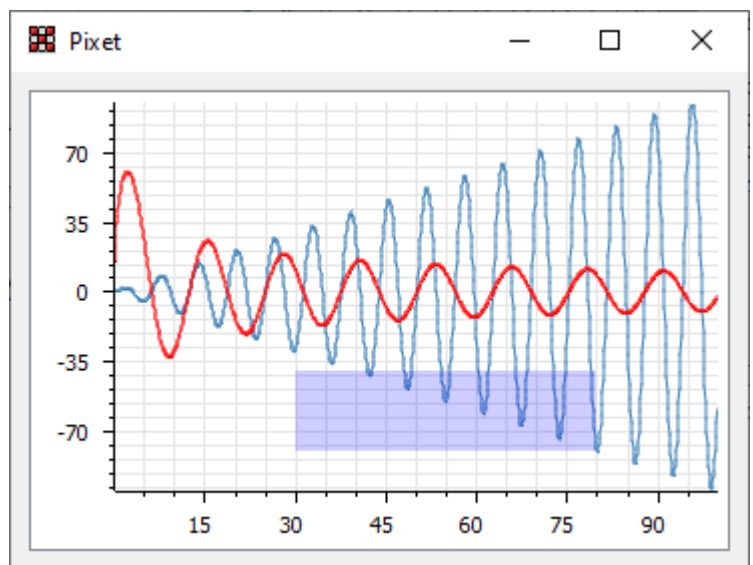
```
import pygui, math

window = pygui.MainWindow()
plot = window.Plot()
plot.setMinimumSize(350, 230)
gl = window.newGridLayout()
gl.addWidget(plot, 0, 0, 1, 1)
window.setLayout(gl)
window.show()

datax = []
datay0 = []
datay1 = []
for n in range(1, 1000):
    x = n/10
    datax.append(x)
    datay0.append(x * math.sin(x))
    datay1.append(100/math.sqrt(x) * math.sin(x/2))

plot.addLine(0)
plot.addLine(0)
plot.setLineData(0, 0, datax, datay0)
plot.setLineData(0, 1, datax, datay1)
plot.setLineColor(0, 1, 255, 0, 0)

plot.addHighlight(0)
plot.setHighlightRect(0, 0, 30, 80, -80, -40)
plot.setHighlightColor(0, 0, 0, 0, 255, 50)
plot.update()
```



Create: **window.Plot()**

addLayer()/layerCount()	Adds new layer / Returns count of layers. Default is 1.
clear()	Clears whole plot to layerCount 1.
showLayer(layerIdx, show)	Shows/hides the indexed layer.
clearLayer(layerIdx)	Clears all Lines, BarLines, Scatters and Highlights in the layer to count 0.
update()	Updates all. Lines, colors, Scatter points with False repaint.
addLine(layerIdx)/addBarLine(layerIdx)	Adds new line/barLine to the indexed layer.
lineCount(layerIdx)	Returns count of (bar)lines on the indexed layer. Default is 0.
clearLine(layerIdx, lineIdx)	Clears the indexed (bar)line to pointCount 0.
setLineData(layerIdx, lineIdx, [xs], [ys])	Sets data of whole the indexed (bar)line.
showLine(layerIdx, lineIdx, show)	Shows/hides the indexed (bar)line.
setLineWidth(layerIdx, lineIdx, width)	Sets width of the indexed (bar)line.
setLineColor(layerIdx, lineIdx, r, g, b)	Sets color of the indexed (bar)line.
addPoint(layerIdx, lineIdx, x, y)	Adds a single point to the indexed (bar)line.
replacePoint(layerIdx, lineIdx, x, y)	Replaces a single point in the indexed (bar)line.
pointCount(layerIdx, lineIdx)	Returns counts of points in the indexed (bar)line.
addScatter(layerIdx)	Adds new scatter to the indexed layer.
scatterCount(layerIdx)	Returns count of scatters on the indexed layer. Default is 0.
addScatterPoint(layerIdx, scatterIdx, x, y, z, repaint)	Adds a single scatter point. Z is color index to the ColorMap.
clearScatter(layerIdx, scatterIdx)	Clears all points in the indexed scatter on the indexed layer.
setScatterColorMap(layerIdx, scatterIdx, colorMapIdx)	Sets ColorMap applied to Z parameter of scatter points.
setScatterPointSize(layerIdx, scatterIdx, width, height)	Sets size of the scatter points.
showScatter(layerIdx, scatterIdx, show)	Shows/hides the indexed scatter.
addHighlight(layerIdx)	Adds new highlight to the indexed layer.
highlightCount(layerIdx)	Returns count of highlights in the indexed layer.
setHighlightRect(layerIdx, highlightIdx, x1, x2, y1, y2)	Sets highlighted zone rectangle coordinates.
highlightRect(layerIdx, highlightIndex)	Returns list of coordinates of the indexed highlight.
setHighlightMode(layerIdx, highlightIdx, enabled)	Enables/disables the indexed highlight.
setHighlightColor(layerIdx, highlightIdx, r, g, b, alpha)	Sets color and transparency % of the indexed highlight.
setAutoRangeX/setAutoRangeY	Enables/disables lower and upper autoranges on X/Y. (lowerOn, upperOn)
setRangeX/setRangeY/setRangeZ	Sets lower and upper ranges of line axis and of Z in scatter. (lower, upper)
setTitle(title)	Sets the plot tittle text.
showTitle(show)	Shows/hides the plot tittle text.
setLabelX(text)/setLabelY(text)	Sets the axis labels texts.
showLabelX(show)/showLabelY(show)	Shows/hides the axis labels texts.
showAxisX(show)/showAxisY(show)	Shows/hides the axes.
showColorBar(show)	Shows/hides color-mapping bar for scatters.
setRightLegend/setBottomLegend	Sets the lines legends on the indexed layer (layerIdx, [items]).
showRightLegend(show)/showBottomLegend(show)	Shows/hides axis legends.
setEnabled(en)/isEnabled()	Enable/disable and returns enable state.
setMinimumSize/setMaximumSize	Sets size limits (width, height) for window arrangement and resize.

No events.

20.7. Other graphical objects and its methods

Label Simple system label, create: `window.Label("text")`

`setText(text)/text()` Sets and returns label text.
`setEnabled(en)/isEnabled()` Enable/disable and returns enable state (disabled = gray).
`setMinimumSize/setMaximumSize` Sets size limits (width, height) for window arrangement and resize.
`setStyleSheet(styleSheetText)` Sets style of the label text. Example: `label.setStyleSheet("color: red")`

CheckBox Simple system checkbox, create: `window.CheckBox("text")`

`setText/text, setEnabled/isEnabled, setMinimumSize/setMaximumSize` – Same as Label.
`setChecked(check)/isChecked()` Sets and returns check status.

It has the “**clicked**” event. Occurs on button clicked.

RadioButton Simple system radiobutton, create: `window.RadioButton()`

(All methods and events are same as in the CheckBox.)

PushButton Simple system button, create: `window.Button("text")`

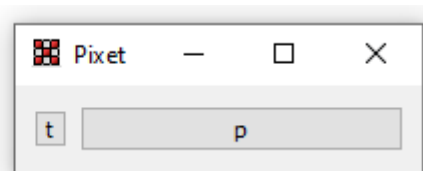
`setText/text, setEnabled/isEnabled, setMinimumSize/setMaximumSize` – Same as Label.

It has the “**clicked**” event. Occurs on button clicked.

ToolButton Smaller button than PushButton, without default text, create: `window.ToolButton()`

PushButton
vs
ToolButton

```
tb = window.ToolButton()
tb.setText("t")
pb = window.Button("p")
gl.addWidget(tb, 0, 0, 1, 1)
gl.addWidget(pb, 0, 1, 1, 1)
```



SpinBox Spinbox for int-type numbers (line edit with +/- buttons), create: `window.SpinBox()`

`setValue(val)/value()` Sets and returns the value.
`setMaximum(val)/maximum()` Sets and returns the maximal allowed value.
`setMinimum(val)/minimum()` Sets and returns the minimal allowed value.
`setStep(val)/step()` Sets and returns the step of +/- buttons.
`setEnabled(en)/isEnabled()` Enable/disable and returns enable state.
`setMinimumSize/setMaximumSize` Sets size limits (width, height) for window arrangement and resize.

DoubleSpinBox Spinbox for double-type numbers (line edit with +/-), create: `window.DoubleSpinBox()`

(All methods are same as in the SpinBox.)

ComboBox Simple system combobox, create: **window.ComboBox()**

setCurrentIndex(val)/currentIndex() Sets and returns the list index.
currentText() Returns current text.
setItems([items]) Sets items list.
setEnabled(en)/isEnabled() Enable/disable and returns enable state.
setMinimumSize/setMaximumSize Sets size limits (width, height) for window arrangement and resize.

It has the “**changed**” event. Occurs on value changed.

LineEdit Single line text edit frame, create: **window.LineEdit()**

setText/text, setEnabled/isEnabled, setMinimumSize/setMaximumSize – Same as Label.

TextEdit Multiline text edit frame, create: **window.TextEdit()**

setText/text, setEnabled/isEnabled, setMinimumSize/setMaximumSize – Same as Label.
append(text) Appends a text to the edit.
clear() Clear text in the edit.

Slider Slider with “regulator” style, create: **window.Slider()**

setValue(val)/value() Sets and returns the slider position value.
setSliderPosition(val)/sliderPosition() Same as setValue/value.
setMaximum(val)/maximum() Sets and returns the highest slider value.
setMinimum(val)/minimum() Sets and returns the lowest slider value.
setSingleStep(val)/singleStep() Sets and returns the step of up/down arrow keys.
setPageStep(val)/pageStep() Sets and returns the step of pageUp/pageDown keys.
setEnabled(en)/isEnabled() Enable/disable and returns enable state.
setMinimumSize/setMaximumSize Sets size limits (width, height) for window arrangement and resize.
setStyleSheet(styleSheetText) Sets style of the text. Example: **slider.setStyleSheet("color: red")**

It has the “**changed**” event. Occurs on value changed.

ProgressBar Progressbar with percents, create: **window.ProgressBar()**

setValue(val)/value() Sets and returns the bar position value.
setMaximum(val)/maximum() Sets and returns the highest bar value.
setMinimum(val)/minimum() Sets and returns the lowest bar value.
setEnabled(en)/isEnabled() Enable/disable and returns enable state.
setMinimumSize/setMaximumSize Sets size limits (width, height) for window arrangement and resize.
setStyleSheet(styleSheetText) Sets style of the text. Example: **bar.setStyleSheet("color: red")**

ListWidgetSimple system listview, create: **window.ListWidget()**

setItems([items])	Sets items list.
setItemChecked(index, check)	Sets checked state of the indexed item.
setItemSelected(index, sel)	Sets selected state of the indexed item.
itemText(index)	Returns text from the indexed item.
selectedIndexes()	Returns list of selected items indexes.
checkedIndexes()	Returns list of checked items indexes.
setEnabled(en)/isEnabled()	Enable/disable and returns enable state.
setMinimumSize/setMaximumSize	Sets size limits (width, height) for window arrangement and resize.

It has the “**changed**” event. Occurs on value changed.