Python Scripting for Pixelman

Daniel Turecek

IEAP CTU in Prague

Contents

1 Introduction

The Python Scripting plugin (JScripting.jar) is a plugin that embeds a python interpreter (Jython http://www.jython.org/) into Pixelman. It allows to control Pixelman functionality from within python script and realize complex automatic measurements, etc. The Python scripting has access to all API of Pixelman that is available for plugins as well as functions exported by other plugins.

2 Usage of the plugin

The Scripting plugin can be started from Pixelman menu: **Script -> Python Scripting**, or from Device Control Panel: **Utilities -> Python Scripting**.

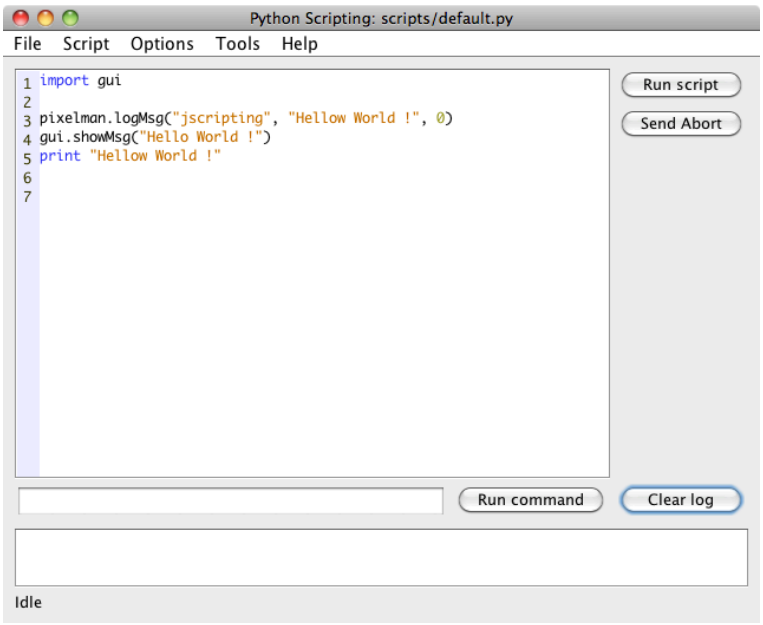

Figure 1: Main window of the Scripting plugin

The main window contains an editor pane into which a script can be written. The pane automatically highlights syntax. The text field under the editor is field for quick python commands that are executed by **Run command** button. In the bottom of the window a log pane shows output of python scripts (output of commands such as print), syntax error and exception messages and also messages logged in Pixelman log.

2.1 Executing a script

A script can be executed by pressing **Run script** button. The script is then executed line by line.

2.2 Aborting a script

The abortion of the execution of the script is performed by clicking on **Abort** button (Run script button changes name when script is running). It is possible to detect the event of clicking on Abort button in the script. The script can react on this event and exit gracefully (e.g. close files and other resources) instead of forced interruption. There are two possible ways to detect aborting:

Aborting variable

When the script defines global variable named Aborting and the Abort button is pressed, the value of this variable is set to True. Second click on the Abort button then forces script abortion.

```
import time
Aborting=False

for i in range(100):
    print i
    time.sleep(0.5)
    if Aborting:
        break
```

Code Listing 1: Script that outputs numbers from 0 to 100 and stops when Aborting is True

OnAbort function

When the script defines global function OnAbort then this function is called when Abort button is pressed. Second click on the Abort button then forces script abortion.

```
import time
abort=False

def OnAbort():
    global abort
    abort=True
    print "Script aborted."

for i in range(100):
    print i
    time.sleep(0.5)
    if abort:
        break
```

Code Listing 2: Script that outputs numbers from 0 to 100 and stops when OnAbort function is called.

In the case that the script is running in a separate thread, the user can send the abort event by clicking on **Send Abort** button. This will change Aborting variable to True and calling OnAbort function.

2.3 Startup scripts

The Startup scripts dialog can be opened from the menu **Options->Startup scripts**. The user can specify in this window scripts that should be loaded on the Pixelman (scripting plugin) startup. New scripts is added by **Add** button, removed by **Remove** button. Selected script can be moved up and down in the execution order by **Move Up** and **Move Down**. To enable execution of scripts on startup the checkbox **Execute scripts on program startup** must be checked.

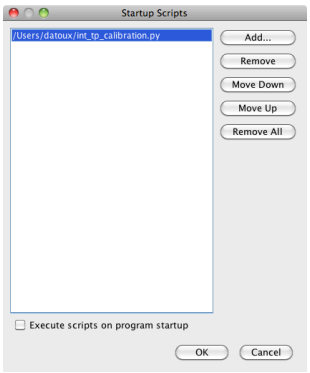

Figure 2: Startup scripts dialog

It is also possible to add a script that will be executed on startup via **Add python plugin** menu item in Pixelman menu. The script will be added to the list of startup scritps and if startup execution is disabled it is re-enabled.

2.3.1 Script packages

Scripting plugin also supports scripts packed in zip archive (but with extension renamed to **.pyz**). Such a script package has to have a **main.py** script that will be executed by Pixelman. Package contents is unziped into a temporary directory and this directory is automatically added to python sys.path. That way main.py script can simply import other scripts in that directory.

2.4 Functions list dialog

The help dialog that contains list of all available pixelman functions and functions of exported by plugins is accessible from menu **Help->List Pixelman functions**.

This dialog contain list of these items:

- **Pixelman Constants** - start with*final static*
- **Pixelman API fuctions** - functions starting with pixelman. All the API functions that are available for plugins.
- **Exported functions** - functions exported by plugins, start with name of the plugin
- **Scripts functions** - functions of scripts in *lib* directory (this directory is automatically scanned), start with script name

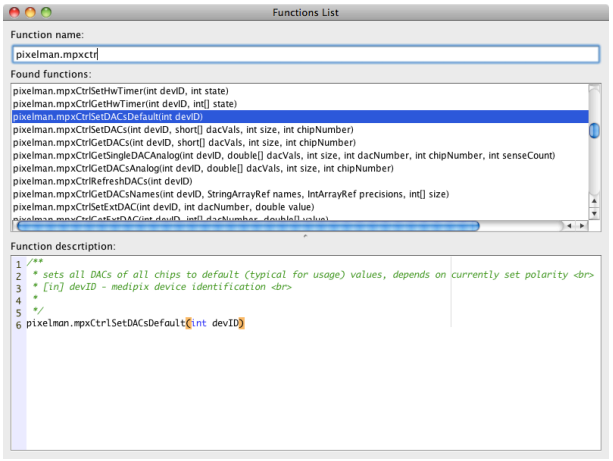

Figure 3: Functions list dialog

- **Scripts constants** - constants in the scripts in *lib* directory, start with script name

The function can be searched by typing in the *Function name* field. The list is automatically filtered as user types. When only small case letters are inputed the search is non-case sensitive. When both cases are used, it is case sensitive search.

The documentation for Pixelman functions was taken from java source codes, therefore it contais java types and declaration, therefore it does not corresponed to python types. The simple types such as int, double, etc. can be used directly from python. For the other, java object has to be passed. How to use java classes and objects from scripting plugin will be described later.

2.5 Python Console

Python console, accessed from menu **Tools->Python Console**, is a console window that allows to enter python commands, execute pixelman functions interactively. The commands is executed by pressing **Enter** key. The console supports partial function completion. When a module name such as *pixelman* or object is entered and dot is written a small window appeards that lists object functions.

The console also supports several shortcuts:

- **Arrow Up** - going back in the of entered commands
- **Arrow Down** - going forward in the history of entered commands
- **CTRL+E** - go with cursor to the end of line
- **CTRL+A** - go with cursor to the beginnig of line
- **CTRL+U** - delete text from cursor position to the beginning of the line
- **CTRL+K** - delete text from cursor position to the end of line
- **CTRL+L** - clears content of console
- **F5** - run last command

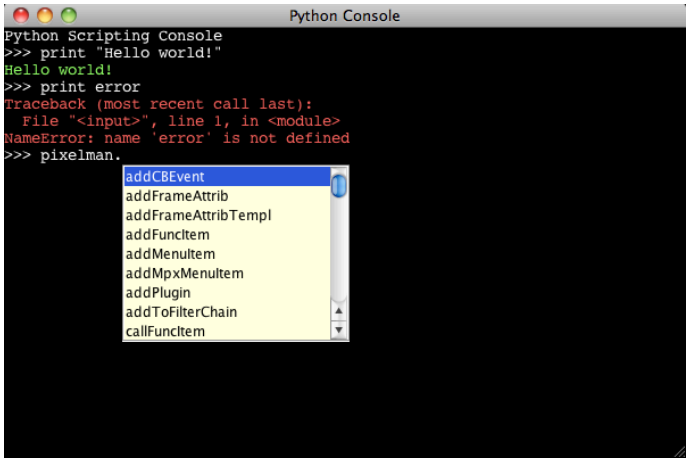

Figure 4: Python console with functions completion

3 Python language overview

3.1 Introduction

Python is strongly typed (i.e. types are enforced), dynamically, implicitly typed (i.e. you don't have to declare variables), case sensitive (i.e. var and VAR are two different variables) and object-oriented (i.e. everything is an object) language.

Syntax

- No mandatory statement termination (e.g. **;** not necessary)
- Blocks are specified by **identation**
- Statements expecting identation level ends with **:**
- Comments starts with # sign and are single-line
- Multi-line comments starts with """
- Variables are assigned with =
- Equality testing is done using ==

Example:

```
myvar = 3  # myvar is 3
myvar += 2 # myvar is 5
myvar -= 1 # my var is 4
"""This is a multiline comment.
The following lines concatenate the two strings."""
mystring = "Hello"
mystring += " world." # mystring is Hellow world.
myvar, mystring = mystring, myvar # This swaps the variables in one line
```

Operators and Maths

- Arithmatic: +, -, *, /, % **(modulus)**
- Comparison: ==, !=, <, >, <=, >=
- Logical: and, or, not
- Exponentiation: **

3.2 Variables and Datatypes

Variables in Python follow the standard nomenclature of an alphanumeric name beginning in a letter or underscore. Variable names are case sensitive. Variables do not need to be declared and their datatypes are inferred from the assignment statement. Python supports following data types:

boolean, integer, long, float, string, list, dictionary, tuple, Object, None

```
a = True # boolean variable. Can be True or False
b = 1 # integer variable
c = 10.5 # float variable
d = "text" # string variable
e = [1, 2] # list variable containing numbers 1 and 2
f = {a=1, b=3} # dictionary variable containing pair key-value: a=1, b=3
```

Variable scope

Most variables in Python are local in scope to their own function or class. For instance if you define *a = 1* within a function, then a will be available within that entire function but will be undefined in the main program that calls the function. Variables defined within the main program are accessible to the main program but not within functions called by the main program.

Global Variables: Global variables are declared outside of functions and can be read without any special declarations, but if you want to write to them you must declare them at the beginning of the function with the "global" keyword, otherwise Python will bind that object to a new local variable (be careful of that, it's a small catch that can get you if you don't know it). For example:

```
number = 5

def myfunc():
    # This will print 5.
    print number

def anotherfunc():
    # This raises an exception because the variable has not
    # been bound before printing. Python knows that it an
    # object will be bound to it later and creates a new, local
    # object instead of accessing the global one.
    print number
```

```
    number = 3

def yetanotherfunc():
    global number
    # This will correctly change the global.
    number = 3
```

Lists, Dictionaries, Tuples

Lists are like **one-dimensional arrays** (but you can also have lists of other lists). **Dictionaries** are **associative arrays** (a.k.a. hash tables) and tuples are immutable one-dimensional arrays (Python "arrays" can be of any type, so you can mix e.g. integers, strings, etc in lists/-dictionaries/tuples). The **index** of **the first item** in all array types is **0**. Negative numbers count from the end towards the beginning, -1 is the last item. Variables can point to functions.

```
a = [1, 2, 3] # simple list containig just numbers
# list with different types and sub list:
b = [1, ["another", "list"], "text"]
mylist = ["List item 1", 2, 3.14]
mylist[0] = "List item 1 again"
mylist[-1] = 3.14 # negative index - indexing goes from end
mydict = {"Key 1": "Value 1", 2: 3, "pi": 3.14}
mydict["pi"] = 3.15
mytuple = (1, 2, 3)
myfunction = len
```

You can access array ranges using a colon (:). Leaving the start index empty assumes the first item, leaving the end index assumes the last item. Negative indexes count from the last item backwards (thus -1 is the last item) like so:

```
mylist = ["List item 1", 2, 3.14]
print mylist[:]     #  ['List item 1', 2, 3.1400000000000001]
print mylist[0:2]   # ['List item 1', 2]
print mylist[-3:-1] # ['List item 1', 2]
print mylist[1:]    # [2, 3.14]
```

Strings

Strings can use either **single or double quotation marks**, and you can have quotation marks of one kind inside a string that uses the other kind (i.e. "He said 'hello'." is valid). **Multiline** strings are enclosed in **triple double** (or single) **quotes** (""). Python supports **Unicode** out of the box, using the syntax **u"This is a unicode string"**.

The **raw strings** are prefixed with r and they do not expand escape sequences. To fill a string with values, you use the % (modulo) operator and a tuple. Each %s gets replaced with an item from the tuple, left to right, and you can also use dictionary substitutions, like so:

```
print "Name: %s Number: %s String: %s" % (myclass.name, 3, 3 * "-")
#Result: Name: Poromenos Number: 3 String: ---
strString = """This is
```

```
a multiline
string."""

# WARNING: Watch out for the trailing s in "%(key)s".
print "This %(verb)s a %(noun)s." % {"noun": "test", "verb": "is"}
#Result: This is a test.

r"This is a raw string, the newline \n will not expand"
```

3.3 Flow control statements (if, for, while)

Flow control statements are if, for, and while. There is no select/switch; instead, use if. Use for to enumerate through members of a list. To obtain a list of numbers, use range(<number>). These statements' syntax is thus:

```
rangelist = range(10)
print rangelist # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for number in rangelist:
    # Check if number is one of
    # the numbers in the tuple.
    if number in (3, 4, 7, 9):
        # "Break" terminates a for without
        # executing the "else" clause.
        break
    else:
        # "Continue" starts the next iteration
        # of the loop. It's rather useless here,
        # as it's the last statement of the loop.
        continue
else:
    # The "else" clause is optional and is
    # executed only if the loop didn't "break".
    pass # Do nothing

if rangelist[1] == 2:
    print "The second item (lists are 0-based) is 2"
elif rangelist[1] == 3:
    print "The second item (lists are 0-based) is 3"
else:
    print "Dunno"

while rangelist[1] == 1:
    pass
```

3.4 Functions

Functions are **declared with the "def"** keyword. **Optional arguments** are set in the function declaration after the mandatory arguments by being assigned a default value. For

named arguments, the name of the argument is assigned a value. Functions can return a tuple (and using tuple unpacking you can effectively return **multiple values**). **Lambda functions** are ad hoc functions that are comprised of a single statement. Parameters are **passed by reference**, but immutable types (tuples, ints, strings, etc) *cannot be changed*. This is because only the memory location of the item is passed, and binding another object to a variable discards the old one, so immutable types are replaced. For example:

```
# Same as def f(x): return x + 1
functionvar = lambda x: x + 1
print functionvar(1) # 2

# an_int and a_string are optional, they have default values
# if one is not passed (2 and "A default string", respectively).
def passing_example(a_list, an_int=2, a_string="A default string"):
    a_list.append("A new item")
    an_int = 4
    return a_list, an_int, a_string

my_list = [1, 2, 3]
my_int = 10
print passing_example(my_list, my_int)
# ([1, 2, 3, 'A new item'], 4, "A default string")
print my_list # [1, 2, 3, 'A new item']
print my_int # 10
```

3.5 Exceptions

Exceptions in Python are handled with try-except [exceptionname] blocks:

```
def some_function():
    try:
        # Division by zero raises an exception
        10 / 0
    except ZeroDivisionError:
        print "Oops, invalid."
    else:
        # Exception didn't occur, we're good.
        pass
    finally:
        # This is executed after the code block is run
        # and all exceptions have been handled, even
        # if a new exception is raised while handling.
        print "We're done with that."

print some_function()
# Oops, invalid.
#We're done with that.
```

3.6 Importing

External libraries are used with the import [libname] keyword. You can also use from [libname] import [funcname] for individual functions. Here is an example:

```
import random
from time import clock

randomint = random.randint(1, 100)
print randomint # 64
```

3.7 File I/O

Python has a wide array of libraries built in. As an example, here is how serializing (converting data structures to strings using the pickle library) with file I/O is used:

```
import pickle
mylist = ["This", "is", 4, 13327]
# Open the file C:\binary.dat for writing. The letter r before the
# filename string is used to prevent backslash escaping.
myfile = open(r"C:\binary.dat", "w")
pickle.dump(mylist, myfile)
myfile.close()

myfile = open(r"C:\text.txt", "w")
myfile.write("This is a sample string")
myfile.close()

myfile = open(r"C:\text.txt")
print myfile.read() # 'This is a sample string'
myfile.close()

# Open the file for reading.
myfile = open(r"C:\binary.dat")
loadedlist = pickle.load(myfile)
myfile.close()
print loadedlist # ['This', 'is', 4, 13327]
```

4 Pixelman specific functions

4.1 Pixelman API

Every python script can access all function of Pixelman API via **pixelman** module. This module does not have to be explicitly imported. It is imported automatically. The module contains functions as well as important constants that functions use. The Pixelman API consists of two sets of functions. One set starts with prefix **mpxCtrl**. These functions are functions of Medipix Control library and mostly represent function that access devices. The functions without any prefix are functions of Medipix Manager that can be used for working with frames, other plugins, menu items, etc. Below is an example:

```
# file save flags constants
flags=pixelman.FSAVE_ASCII|pixelman.FSAVE_SPARSEX

# Control Library function that acquire frames from device
pixelman.mpxCtrlPerformFrameAcq(devID, acqCount, acqTime, flags, fileName)

# logs a message to Pixelman log
pixelman.logMsg(pluginName, "Sample message", 0)
```

Code Listing 3: Example of Pixelman API functions and constants.

4.2 Functions registered by Plugins

Every plugin has possibility to register some functions that will be available to other plugins (e.g. usb stepper motor plugin registers functions to control motors). These functions can be called from scripting plugin. In order to use registered function a module **pixelman_regfuncs** has to be imported first, then a module with a name corresponding to the plugin name whose functions needs to be used. For example to use functions of *jmpxctrlui* plugin, **pixelman_regfuncs** and **jmpxctrlui** modules has to be imported. Example:

```
# importing module that creates access to registered functions
import pixelman_regfuncs
# importing registered functions of jmpxctrlui plugin (Device Control)
import jmpxctrlui
# starts acquisition (simulates clicking on Start button)
# in Device Control panel
jmpxctrlui.StartAcq(devID)
```

Code Listing 4: Starting acquisition in Device Control plugin from script

4.3 Data Types of API functions parameters

The Pixelman API function available in python are called through Java. This means that, these functions take java types as parameters. Some basic java types can be directly mapped to python types such as (int, short, double or String). Such parameters can be filled with python variable of same type or just by value (e.g. "string", 2, 1.5). Example:

```
frameID=10
fileName="c:\\test.txt"
pixelman.saveFrame(frameID, fileName, 0)
```

Code Listing 5: Example of data types

However, some java types requires a bit more work. These are desribed further.

4.3.1 Arrays

The parameters that contain array (e.g. int[] parameter) need java array as input. These can be created in python with functions **zeros** or **array**.

zeros - This function creates a java array. The first argument is the length of the array, and the second is either a character representing a basic type, or a Java class name. The character mapping to java type is as follows:

char	java type
z	boolean
c	char
b	byte
h	short
i	int
l	long
f	float
d	double

array - will create a new array of the same length as the input sequence and will populate it with the values in sequence.

Example:

```
from java.lang import String
array([1,2,3], 'd') # array of double
zeros(5, String) # array of String
frameID=zeros(1, 'i') # array of int
#pixelman.loadFrame(int[] frameID, String fileName, int idx)
pixelman.loadFrame(frameID, "c:\\frame.txt", 0)
```

Code Listing 6: Example of usage of java arrays

5 Samples

5.1 Working with frames

Every frame in pixelman is referenced by **frame ID**. All the functions that manage frames expect as parameter this frame ID. To facilitate working with frames, a python module **mpxframe**

was created. To use this module it has to be first imported by: **import mpxframe**. This module contains a class **MpxFrame**, which contains frame data and parameters (frameID, width, height, type, ...), when frame is loaded/created.

5.1.1 Creating a new frame

A new frame can be created by function **mpxframe.createFrame(frameName, type, width, height, creator)**. Example:

```
import mpxframe
#creates empty frame with name "test frame"
#other parameters are default (type=TYPE_DOUBLE, width=height=256)
frame = mpxframe.createFrame("test frame")
```

Code Listing 7: Example of creating a new frame

5.1.2 Loading frames

A frame can be loaded into MpxFrame object either from file, by frame ID or from last acquisition.

```
import mpxframe
# loads frame by frame ID
frame = mpxframe.loadFromID(frameID)

#get number of frames in the file (multiframe file)
count = mpxframe.getFrameCount(fileName)

# loads frame with index 0 from (multiframe) file c:\testframe.txt
frame2 = mpxframe.loadFromFile("c:\\testframe.txt", 0)
# .... some work with frame ....
frame2.close() # close frame loaded from file, delete from memory

# loads frame from last acquisition with frame index
frame3 = mpxframe.getAcqFrame(devID, frameIndex)

# loads last frame from last acquisition
frame4 = mpxframe.getLastAcqFrame(devID)
```

Code Listing 8: Example of loading a frame

5.1.3 Access to frame data

There are two possiblities to access the frame data. Readonly one, and direct access, when frame data can be changed. To get copy of frame data, the function **getDataCopy()** of frame object can be called. This function returns a **DoubleBuffer** object. The single pixels can be retrieved by calling method **get()** or **get(index)** of this object.

To get direct access to frame data, frame has to be locked, in order to avoid concurent changes of frame by other plugins or Pixelman core. To lock frame and get data, function

lock is called. This function returns either DoubleBuffer, ShortBuffer or IntBuffer according to frame type. Pixels can be accesses by directly working on this buffer or by calling functions of MpxFrame object **get(index)** and **put(index)**. When the operation with frame is finished, it is crucial to call **unlock()** so that other plugins can work with the frame.

Examples:

```
import mpxframe
# loads frame with index 0 from (multiframe) file c:\testframe.txt
frame = mpxframe.loadFromFile("c:\\testframe.txt", 0)
buffer = frame.getDataCopy() # get readonly buffer
pixel = buffer.get(10) # get pixel with index 10

frame.lock() # locks frame and get buffer
pixel = frame.get(10) # get pixel with index 10

# change all pixels of frame to value 5
for i in range(frame.getPixelCount()):
        frame.put(i, 5)

frame.unlock() # unlock frame and release it to other plugins
```

Code Listing 9: Example of accessing frame data

5.1.4 Frame parameters

There are several convenient functions to get basic frame information and parameters:

```
import mpxframe
frame = mpxframe.loadFromFile("c:\\testframe.txt", 0)
width = frame.getWidth()
height = frame.getHeight()
type = frame.getType()
pixelCount = frame.getPixelCount()
frameName = frame.getName()
```

Code Listing 10: Example of frame parameters

5.1.5 Frame attributes

Every frame may contain several attributes such as acquisition time, start time, used detector chip id, etc. This attributes can be read, changed and new attributes can be appended.

addAttrib(name, description, type, data) - adds new attribute to the frame. Attribute has name *name* and information text *description*. Type is type of frame such as (mpxutils.TYPE_DOUBLE, mpxutils.TYPE_I16, ...). Data is value of the attribute (can be python variable or directly variable such as 5, 2.2, "string").

getAttrib(name) - gets value of frame attribute specified by name.

setAttrib(name, type, data) - sets value of frame attribute specified by name. Type is attribute type (TYPE_DOUBLE, ...) and data value of the attribute.

```
import mpxframe
import mpxutils
frame = mpxframe.loadFromFile("c:\\testframe.txt", 0)
frame.addAttrib("parameter","info",  mpxutils.TYPE_U32, 5)
frame.addAttrib("parameter2","info",  mpxutils.TYPE_CHAR, "Ahoj")
frame.setAttrib("parameter", mpxutils.TYPE_U32, 10)
print frame.getAttrib("parameter")
print frame.getAttrib("parameter2")
```

Code Listing 11: Example of frame attributes

5.2 Controlling detector

To change detector parameters such as threshold, bias and others, a conveniente module **mpxutils** was created. This module contains constants (DAC names, types constants, ...) and also function to change detector parameters.

```
import mpxutils
devID = 0 # first connected detector
bias = mpxutils.getBias(devID) # get current bias value
mpxutils.setBias(devID, 100) # sets bias to 100 V

tpxclock = mpxutils.getTpxClock(devID) # gets Timepix clock
mpxutis.setTpxClock(devID, 10) # sets TpxClock to 10 Mhz

# chip id e.g. K08-W0082
chipID = mpxutils.getChipID(devID)
# chip type mpxutils.MPX_TPX, mpxutils.MPX_3, mpxutils.MPX_MXR
chipType = mpxutils.getChipType(devID)

#sets threshold to value 400 for timepix detector
mpxutils.setThreshold(devID, mpxutils.MPX_TPX, 400)

# set ikrum dac of timepix detector to value 1
mpxutils.setDAC(devID, mpxutils.MPX_TPX, mpxutils.TPX_IKRUM, 1)

# sets input file/directory of file device c:\data\testframes
mpxutils.setFileDevFile(devID, "c:\\data\\testframes")

# get input file/directory of file device
directory = mpxutils.getFileDevFile(devID)
```

Code Listing 12: Detector parameters

5.3 Acquisition of frames

The acquisition of frames can be performed either by direcly calling functions of Pixelman API such as **pixelman.performFrameAcq()** or by controlling Device Control Panel plugin (jmpxctrlui).

5.3.1 Controlling Device Control Panel

```
import pixelman_regfuncs
import jmpxctrlui
import mpxutils
devID = 0 # first device

# starts acquisition
jmpxctrlui.StartAcq(devID)

# wait for acq finish
mpxutils.waitForAcqFinish(devID)

# abort acquisition
jmpxctrlui.AbortAcq(devID)

# sets acq parameters (acq type frames, 10 frames, time 0.1s)
jmpxctrlui.SetAcqPars(devID, 0, 10, 0.1)
```

Code Listing 13: Controlling Device Control Panel

5.3.2 Calling Pixelman API directly

There are two function to control acquisition: **pixelman.mpxCtrlPerformFrameAcq()** acquires specified number of frames and **pixelman.mpxCtrlPerformIntegralAcq()** performs integral acquisition.

```
pixelman.mpxCtrlPerformFrameAcq(devID, numberOfFrames, timeOfEachAcq,
                                fileFlags, fileName)
pixelman.mpxCtrlPerformIntegralAcq(devID, numberOfFrames, timeOfEachAcq,
                                fileFlags, fileName)
```

where **fileFlags** is bitwise combination ("|") of these flags:

parameter	description
pixelman.FSAVE_BINARY	save frame in binnary format
pixelman.FSAVE_ASCII	save frame in ASCII format
pixelman.FSAVE_I16	save frame as integer 16bit
pixelman.FSAVE_U32	save frame as unsigned integer 32bit
pixelman.FSAVE_DOUBLE	save frame as double
pixelman.FSAVE_SPARSEX	save frame in sparse format [X, COUNT]
pixelman.FSAVE_SPARSEXY	save frame in sparse format [X, Y, COUNT]
pixelman.FSAVE_NODESCFILE	do not save description file
pixelman.FSAVE_APPEND	save into multi-frame file

flags FSAVE_BINARY/FSAVE_ASCII and flags FSAVE_I16/FSAVE_U32/FSAVE_DOUBLE are mutually exclusive.

Example that performs acquisition of 10 frames with acq. time 1s and repeats it 10 times, by changing name of the file accordingly:

```
devID=0 # first device
acqTime=1
acqCount=10
for i in range(10):
    fileName1 = "c:\\folder\\rep%02d_.txt" % i
    pixelman.mpxCtrlPerformFrameAcq(devID, acqCount, acqTime,
          pixelman.FSAVE_ASCII | pixelman.FSAVE_SPARSEX, fileName)
```

Code Listing 14: Example of acquisition thtough Pixelman API