



Arrangement of an Optical Spectrometer Setup at Petra III Beamline P23

Max Stöber, Technische Universität Dresden, Germany

September 6, 2016

Abstract

The Petra III is one of the worlds most brilliant sources for x-ray synchrotron radiation, located at the DESY (Deutsches Elektronen Synchrotron) national German research center in Hamburg.

Over the course of my work at *DESY Summer Student Programme 2016* I have set up, tested and calibrated an optical spectrometer for the soon opening beamline P23. Using the software development kit (SDK), delivered by the hardware manufacturer, a programming interface for python, named *andorpy* has been created. This will be used for the control of experiments on x-ray induced photoluminescence of nanostructures. Moreover, the software includes a command interface for the device and management of the data via the Sardana server system. Interactive macros have been prepared, suitable for several experimental setups.

As a result, the optical spectrometer can now be routinely used by scientists without any further programming necessary.

Contents

1	Motivation	3
2	Experimental Technique	3
2.1	The Spectrometer Setup	3
2.2	Calibration	3
2.3	Cosmic Particle Events	4
3	Enabling Device Control via Python	5
3.1	Access to C Libraries	6
3.2	Andorpy - a new Python Package	7
3.3	Help and Documentation	8
4	Enabling Remote Control via Sardana	8
4.1	The Sardana/Tango Infrastructure	8
4.2	Sardana Scripting	8
4.3	Sending and Receiving Data	10
5	Conclusion	11
	References	11
A	Alphabetical List of all Andorpy Functions	13
B	Code for the Sardana Interface	18

1. Motivation

Photo luminescence induced by X-ray synchrotron radiation offers interesting opportunities of characterizing electronic properties of new nano materials, and therefore revealing information relevant for their potential in future application. As a relatively young field, this effect is often referred to as XEOL (X-Ray Excited Optical Luminescence) [4]. The method uses a secondary radiation usually in a wavelength range 300 nm - 1200 nm, does not interfere with other registration channels as X-ray scattering, electrophysical, temperature, etc and is also very good suited for *in situ* experimental environments.

Due to complex emittance spectra, as well as the relatively low crosssection of X ray - to - luminescence excitation process, the devices for its detection have to be very sensitive and specially designed for this type of measurements.

The aim of this work is to incorporate the specialized equipment necessary for XEOL experiments into the general beamline control infrastructure of PETRA.

2. Experimental Technique

2.1. The Spectrometer Setup

At the P23 beamline at PETRA, the XEOL measurements will be made by a state of the art Shamrock 303i Spectrometer, manufactured by Andor. The device uses blazed gratings in a Littrow configuration - meaning the diffraction angle and the incident angle at the grating are identical. The layout is sketched in Figure 2.1. Three blazed gratings are exchangeable by rotating their holder ("turret") with a stepper motor, thus enabling different wavelength intervals.

An iVac 316 CCD from the same manufacturer is attached to the spectrometers body and both devices are connected to a PC for controlling and data reading.

2.2. Calibration

Transporting the spectrometer, exchanging parts or exposing it to major changes in temperature will lead to small mechanical displacements of the components, and therefore to incorrect data.

As a compensation, there are several degrees of freedom for calibrating the spectrometer, without opening its hood. An offset value can be set for the detector and every one of the three gratings, using the spectral lines of a Xe gas discharge

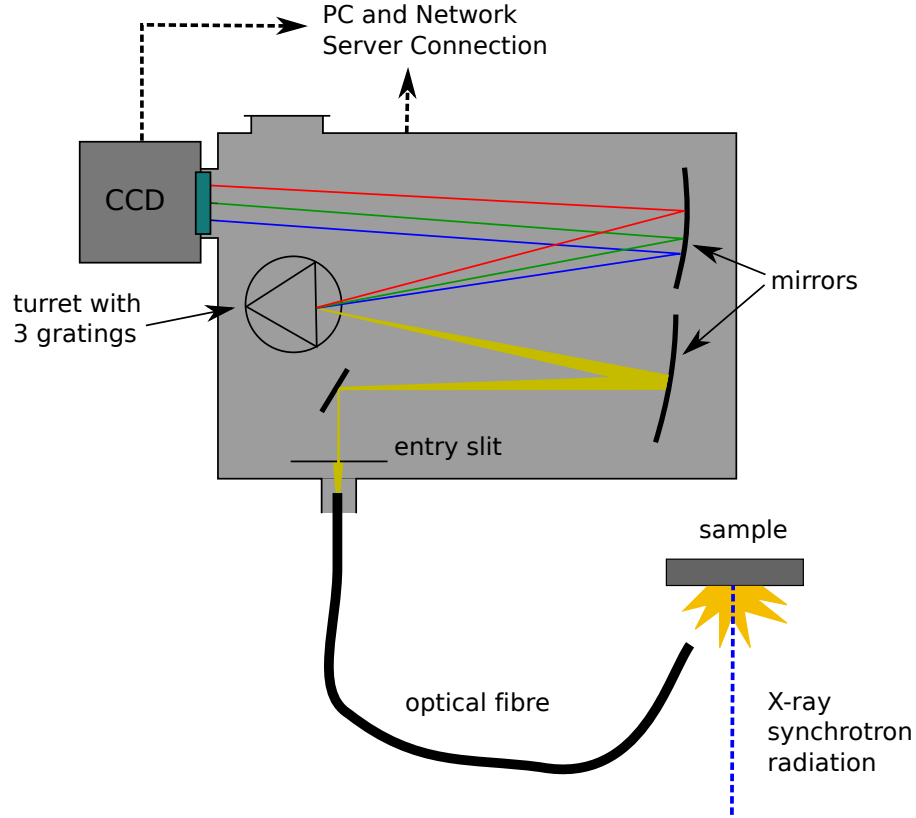


Figure 1: Sketch of the spectrometer setup working principle, with a possible application scenario demonstrated.

lamp as a reference. Fig. 2 demonstrates a small correction of that offset, as it was necessary after the delivery. The offset is saved in a nonvolatile memory integrated to the device, so that those offset values will be the same after power off or connecting to a different PC.

2.3. Cosmic Particle Events

High energy cosmic particles, like muons, can easily pass the building walls and the detectors case. When they approach the CCD, there is a chance for them to produce several thousand photoelectrons, and therefore counts which were not originated from the experiment. Such events can be observed approximately every 10 seconds, and lead to the risk of being mistaken for actual sharp spectral lines (see Fig. 3).

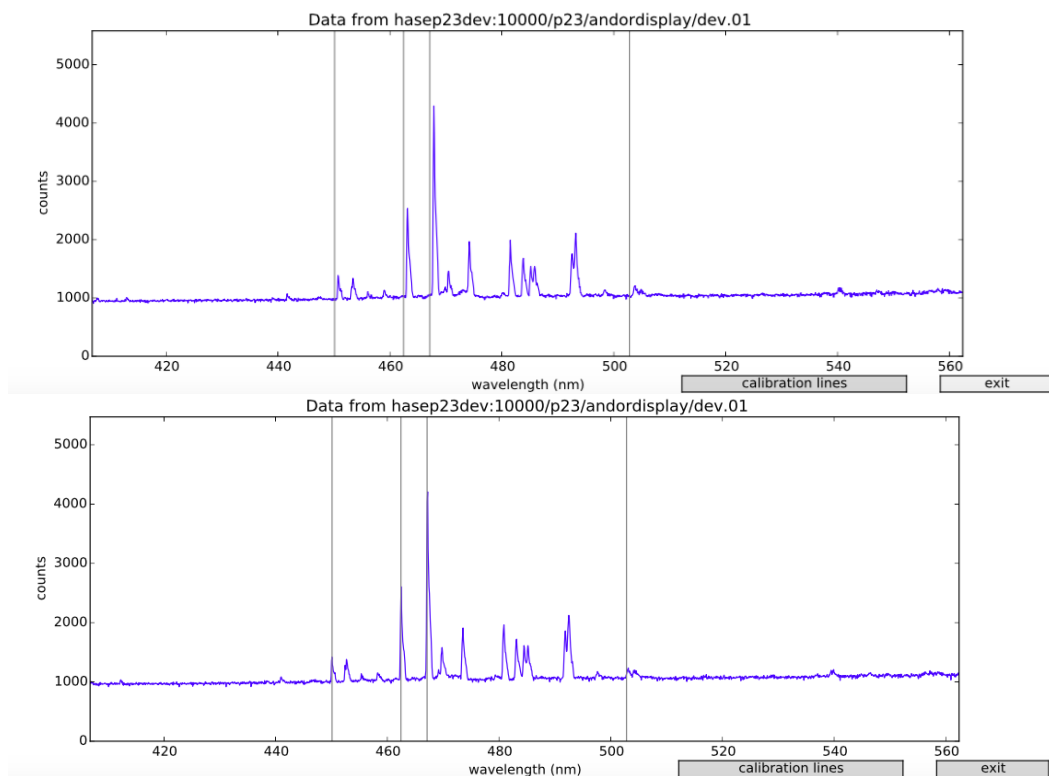


Figure 2: Spectrum of the test lamp with a slightly incorrect calibration and with a grating-offset correction (lower frame).

Those events are always limited to 1-4 pixels, and can be removed by comparing two scans and replacing the pixels with weighted data from the previous scan, if such an event is registered. The Spectrometer has a build in function for that, which was made accessible in Python via `andorpy.ccd.SetFilter()`.

3. Enabling Device Control via Python

Due to the expanding popularity of python for scientific purpose, one would like to have a unified software environment for laboratory devices. In this case the aim is to collect data and control settings of the spectrometer and its CCD in a simple and well documented way. The python package created to do so was named *andorpy*.

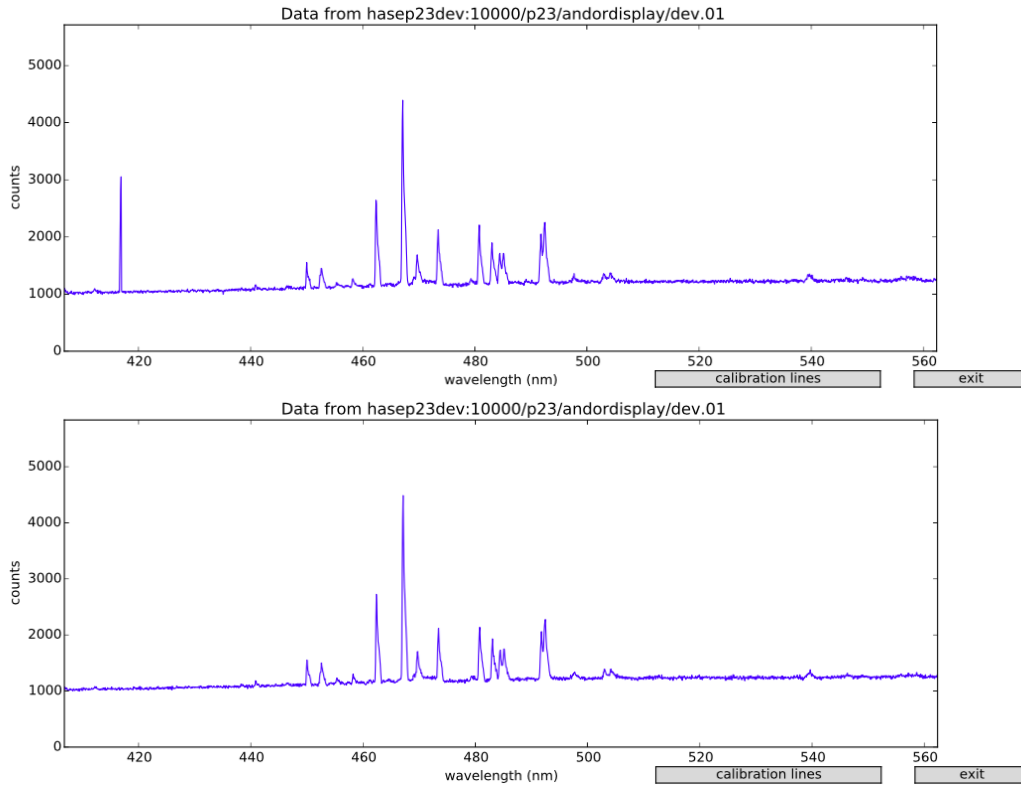


Figure 3: Spectrum featuring a cosmic particle event and without cosmic particle event (lower frame). Activating cosmic particle filtering does not discard whole datasets; the function only replaces a small number of pixels.

3.1. Access to C Libraries

Andor, the manufacturer of CCD cameras, Spectrometer, Microscopes and other devices offers a SDK package for all of their devices. This package consists of two parts, one for CCD, cameras and microscopes, the other for their spectrometers. All of the functions in those packages are written in C++.

The andorpy functions access those two libraries via ctypes. The following minimal example shows how to perform that in case of the CCD library:

```
# load ctypes functions:
import ctypes.util
import ctypes

# locate the library files:
```

```

ccd_lib = ctypes.CDLL(ctypes.util.find_library("andor"))

# Initialize the device:
error = ccd_lib.Initialize()

# Get Temperature of the ccd sensor:
T = ctypes.c_int(0)
error = ccd_lib.GetTemperature(ctypes.pointer(T))

print "Temperature: " + str(int(T.value))

```

Every variable passed to the Andor SDK library has to be defined as a ctype object of the appropriate type - in this case `ctypes.c_int` for an integer. It is important, not to forget the backwards conversion into normal python data types via `your_c_variable.value`, as it was done in the last line of the small example.

3.2. Andorpy - a new Python Package

Of course, repeating the steps described in the previous section every single time a function of the Andor SDK is needed would be not very practical. Using the `andorpy` package, which was created over the course of the *Desy Summer Student Programme*, the same thing can be done more easily:

```

import andorpy

T = andorpy.ccd.GetTemperature()

print "Temperature: " + str(T[1])

```

Now there is no need to take care of the conversion between ctypes objects and usual Python variables anymore. Return values of an `andorpy` function are always tuples, with an error or status message at the index `[0]` and data stored at the other indices.

The package can be obtained via download from the beamline group network space. In its main directory `andorpy_main` the installation routine can be started by entering

```
>>> python setup.py install --user
```

, where the `--user` flag can be left out if the package shall be available for all computer users (admin rights are required then).

3.3. Help and Documentation

A full sized manual for andorpy has been created over the course of the software development [3]. However, research has shown, that the probability of such a document to be read is close to zero [6]. If andorpy is already installed one can access help for each function via `andorpy.function?`. A quick way to do this is via `ipython`.

Example: Getting help on the `SetAcquisitionMode` function:

```
>>> import andorpy
>>> andorpy.ccd.SetAcquisitionMode?
```

This will print a text regarding the usage of the function and its purpose. Also all possible parameters and return values are described briefly as in the SDK users guide.

4. Enabling Remote Control via Sardana

While performing experiments with synchrotron radiation it is usually not allowed to stay in the same room as the measurement devices, for safety reasons. Controlling the spectrometer setup and receiving its data via a computer network is therefore required.

4.1. The Sardana/Tango Infrastructure

Sardana is an environment for control applications, especially in large installations, based on the Tango controls system. There where two main features used:

1. **Sardana Macros** for controlling device parameters, calibration and experimental settings.
2. **Tango Device Server** for sending, saving and receiving data produced by the devices

Figure 4 shows a basic outline of the whole system.

4.2. Sardana Scripting

Sardana Macros are written in plain python, with some restrictions that need to be considered. For instance there is no local reading or writing of files possible

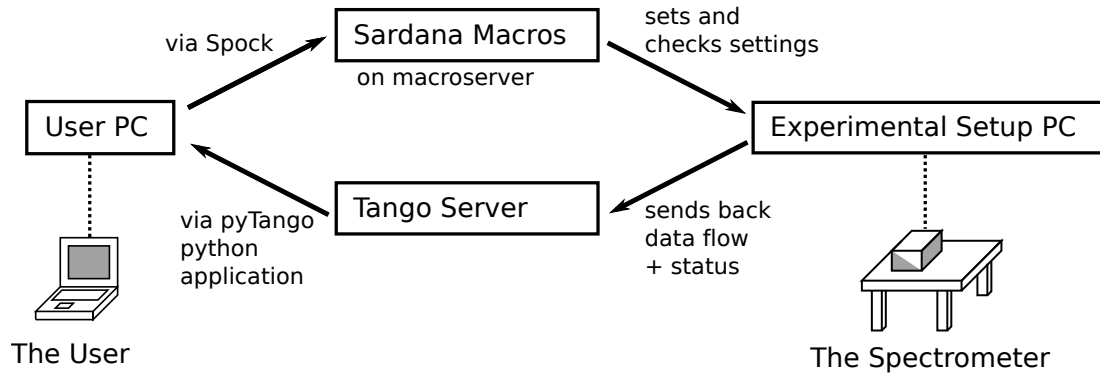


Figure 4: Overview of the infrastructure, as used for the remote control and data acquisition with the andor spectrometer setup.

(use `write` to Tango server instead) and the `print` command is disabled (use `self.output` instead). A collection of 5 Macros was prepared, to be able to handle most situations of data acquisition:

- `andor_init` initializes the spectrometer and detector, sets all parameters to the default value (except calibration parameters).
- `andor_settings` opens a menu that allows you to show and change the most common settings of the devices.
- `andor_measure` starts the measurement and saves all the data on a Sardana server. Those data then can be viewed from all beamline PCs
- `andor_shutdown` closes the system. If the applications are closed without calling that function a segmentation fault will be caused, that often leaves the devices not responding (power off reset may be necessary)
- `andor_status` prints status messages from CCD and spectrometer.

All of those macros are contained in the file `andor_macros.py`, which is listed in the appendix B. This, of course, can and should be used as a help to create macros for more specific functions in the future.

4.3. Sending and Receiving Data

A Tango device server has been setup specifically for the spectrometer. It offers 3 ways of datastream:

- **AndorData** sends arrays of integer values - these are used to store the counts for each pixel of the CCD
- **AndorDataCalibration** sends an array of float values - these are used to store the wavelength for each pixel (equal to the x-axis of the plots like in Figure 2).
- **InfoString** sends a string of textdata. It can in principle be used to store any kind of information, like settings or status. it is used this way in the example code of appendix B.

Writing data to the server is performed in the following way:

```
import PyTango

andor_display = PyTango.DeviceProxy("hasep23dev:10000/p23/
    andor_display/dev.01")
data = andorpy.ccd.GetAcquiredData()
andor_display.AndorData = data[1]
```

Those data now can be accessed by a python script running on any beamline PC, simply via:

```
andor_display = PyTango.DeviceProxy("hasep23dev:10000/p23/
    andor_display/dev.01")
data = andor_display.AndorData
```

Using these functions, a simple tool for viewing data and saving them was created: *Andor Display* . Figure 2 and 3 are actually screenshots of that tool. Moreover, the script saves all the data in textfiles, hdf5 files and also as a row of screenshots in a pdf file.

5. Conclusion

The startup and test procedures taken on the new optical spectrometer setup have been a success. Comparison of obtained data from a xenon gas discharge lamp with spectra from literature has proven that all parts work together well, even though some minor recalibration after transport has been necessary.

As the main part of my work within the *DESY Summer Student Program*, an easy to use interface of the devices functions for Python was created. In preparation of future experimental practice, this enables the development of automated measurement routines and data acquisition to be much faster than by using C and much more flexible than by using the manufacturers GUI software, which comes without any source files. Moreover, remote controlling the apparatus and gathering data via a server was enabled by using Sardana Macros and Tango Server. This enables the devices to be used without any further programming at all.

Therefore the collection of computer programs and their documentation should be seen as the overall outcome of that experimental work, more than the rather small amount of data created.

Acknowledgement

I would like to thank all my dear colleagues, who where never hesitating to answer questions and in an indispensable manner helped me finding errors and correcting mistakes. I have learned a lot in those weeks, and it is thanks to you. Moreover, the DESY Summer Student organizers shall not be forgotten to be mentioned, together with all the highly motivated lecturers - Enabling me to have a great and inspiring time here in Hamburg.

References

- [1] Andorpy - How it works and how it's integrated into Sardana Macros (software documentation) *Max Stöber*
- [2] Shamrock Spectrometer at Petra III Beamline 23 Quick Start Guide (short documentation) *Max Stöber*
- [3] andorpy (python software package) and utils *Max Stöber*
- [4] A time resolved microfocus facility at the Diamond Light Source *J F W Mosselmans et. al.*

- [5] Andor SDK User's Guide *andor.com*
- [6] Why Don't People Read the Manual? *David G. Novick, Karen Ward*
- *University of Texas at El Paso, Department of Computer Science*
http://digitalcommons.utep.edu/cgi/viewcontent.cgi?article=1010&context=cs_papers

A. Alphabetical List of all Andorpy Functions

The names of these functions have not been changed, even though some of them are quite long. The reason for this is to avoid confusion when reading the andor SDK users guide or official examples written in C.

Note:

1. One example of usage for each function is contained in the testsuite `test_all_functions.py`.
2. The function descriptions in this table are based on and in parts equal to those in the SDK manual [5]

Function	Description	Input	Output	Tested	Note
AbortAcquisition (ccd)	This function aborts the current acquisition if one is active.	none	none	OK	This is especially useful, if acquisition mode 5 (run until abort) is active. see also: <code>ccd.SetAcquisitionMode</code>
CoolerOn (ccd)	Switches on the cooling. On some systems the rate of temperature change is controlled until the temperature is within 3°C of the set value. Control is returned immediately to the calling application.	none	none	OK	see also: Initialize, SetTemperature, GetTemperature
CoolerOff (ccd)	Switches off the cooling. The rate of temperature change is controlled in some models until the temperature reaches 0°C. Control is returned immediately to the calling application.	none	none	OK	see also: Initialize, CoolerOn
FreeInternalMemory (ccd)	This function will deallocate any memory used internally to store the previously acquired data. Note that once this function has been called, data from last acquisition cannot be retrieved.	none	none	OK	—
GetAcquiredData (ccd)	This function will return the data from the last acquisition. The data are returned as long integers (32-bit signed integers).	Int: length	length*int array: counts	OK	—

GetAcquisitionTimings (ccd)	This function will return the current "valid" acquisition timing information. This function should be used after all the acquisitions settings have been set, e.g. SetExposureTime, SetKineticCycleTime and SetReadMode etc. The values returned are the actual times used in subsequent acquisitions. This function is required as it is possible to set the exposure time to 20ms, accumulate cycle time to 30ms and then set the readout mode to full image. As it can take 250ms to read out an image it is not possible to have a cycle time of 30ms.	none	Float: exposure Float: accumulate Float: kinetic	OK	exposure = valid exposure time in seconds; accumulate = valid accumulate cycle time in seconds; kinetic = valid kinetic cycle time in seconds
GetDetector (ccd)	This function returns the size of the detector in pixels. The horizontal axis is taken to be the axis parallel to the readout register.	none	Int: xpixels Int: ypixels	OK	The horizontal number of pixels is important for the Spectrometer calibration (see example <code>test_all_functions.py</code>)
GetFilterMode (ccd)	This function returns the current state of the cosmic ray filtering mode.	none	Int: status 0 = OFF 2 = ON	OK	—
GetPixelSize (ccd)	This function returns the dimension of the pixels in the detector in microns.	none	Float: xSize: width of pixel Float: ySize: height of pixel	OK	The return value xSize is important for the spectrometers calibration. See also: <code>spectrometer.ShamrockSetPixelWidth</code> , <code>ShamrockGetCalibration</code>
GetStatus (ccd)	This function will return the current status of the Andor SDK system. This function should be called before an acquisition is started to ensure that it is IDLE and during an acquisition to monitor the process. mode.	none	String: error	OK	Read the SDK user's guide or the use <code>help(ccd.GetStatus)</code> to view the list of possible errors and a short description.
GetStatus (ccd)	This function will return the current status of the Andor SDK system. This function should be called before an acquisition is started to ensure that it is IDLE and during an acquisition to monitor the process. mode.	none	String: error	OK	Read the SDK user's guide or the use <code>help(ccd.GetStatus)</code> to view the list of possible errors and a short description.
GetTemperature (ccd)	This function returns the temperature of the detector to the nearest degree. It also gives the status of cooling process.	none	Int: T	OK	<code>GetTemperatureF</code> may return a more accurate value, because it uses float instead of int

Initialize (ccd)	This function will initialize the Andor SDK System. As part of the initialization procedure on some cameras (i.e. Classic, iStar and earlier iXion) the DLL will need access to a DETECTOR.INI which contains information relating to the detector head, number pixels, read-out speeds etc. If your system has multiple cameras then see the section Controlling multiple cameras (Andor SDK users guide)	none	none	OK	No need to call this function, if ShamrockInitialize has already been called. Only use one to decrease startup time by several seconds!
SetAccumulationCycleTime (ccd)	This function will set the accumulation cycle time to the nearest valid value not less than the given value. The actual cycle time used is obtained by GetAcquisitionTimings.	Float: time (seconds)	none	OK	Please refer to andor SDK user's guide SECTION 5 - ACQUISITION MODES for further information.
SetAcquisitionMode (ccd)	This function will set the acquisition mode to be used on the next StartAcquisition. Acquisition modes to select: 1 = Single Scan, 2 = Accumulate, 3 = Kinetics, 4 = Fast Kinetics, 5 = Run till abort.	Int: mode	none	OK	In Mode 5 only, the camera continually acquires data until the AbortAcquisition function is called. By using the <code>ccd.SetDriverEvent</code> function you will be notified as each acquisition is completed.
SetExposureTime (ccd)	This function will set the exposure time to the nearest valid value not less than the given value. The actual exposure time used is obtained by GetAcquisitionTimings.	Float: time (seconds)	none	OK	Please refer to andor SDK user's guide SECTION 5 - ACQUISITION MODES for further information.
SetFanMode (ccd)	Allows the user to control the mode of the camera fan. If the system is cooled, the fan should only be turned off for short periods of time. During this time the body of the camera will warm up which could compromise cooling capabilities. If the camera body reaches too high a temperature, depends on camera, the buzzer will sound. If this happens, turn off the external power supply and allow the system to stabilize before continuing.	Int: mode 0 = fan full, 1 = fan low, 2 = fan off	none	OK	Possible reason to use this: avoiding vibration caused by fan. USE WITH CAUTION!

SetFilterMode (ccd)	This function will set the state of the cosmic ray filter mode for future acquisitions. If the filter mode is on, consecutive scans in an accumulation will be compared and any cosmic ray-like features that are only present in one scan will be replaced with a scaled version of the corresponding pixel value in the correct scan.	Int: mode 0 = off, 2 = on	none	OK	Cosmic particle events can lead to extremely high counts and therefore are not erased by choosing high exposure time. It is highly recommended using this function for every acquisition. CAUTION: does not work in single scan mode (see <code>ccd.SetAcquisitionMode</code>) and if the number of accumulations is set less than 2
SetNumberAccumulations (ccd)	This function will set the number of scans accumulated in memory. This will only take effect if the acquisition mode is either Accumulate or Kinetic Series. Parameters int number: number of scans to accumulate.	Int: number of scans	none	OK	It is recommended to set this at least to 2, in order to enable the cosmic particle filter to work. See also: <code>SetFilterMode</code>
SetReadMode(ccd)	This function will set the readout mode to be used on the subsequent acquisitions. Readout modes: 0 = Full Vertical Binning, 1 = Multi-Track, 2 = Random-Track, 3 = Single-Track, 4 = Image.	Int: mode	none	see footnote ¹	Read SDK user's guide for information on how those modes operate.
SetTemperature (ccd)	This function will set the desired temperature of the detector. To turn the cooling ON and OFF use the CoolerON and CoolerOFF function respectively.	Int: T	none	OK	Don't forget CoolerOn!
ShamrockAtZeroOrder (spectrometer)	Finds if wavelength is at zero order.	Int: device	Int: 0 = at zero order, 1 = not at zero order	OK	—
ShamrockClose (spectrometer)	Closes the Shamrock system down. That includes spectrometer and camera.	none	none	OK	IMPORTANT: If you do not call this function before your python code is terminated, it may cause a segmentation fault. After that in some cases the hardware has to be restarted by poweroff.
ShamrockEepromGetOpticalParams (spectrometer)	returns the Focal Length, Angular Deviation and Focal Tilt from the Shamrock device.	none	Float: fl Float: ad Float: ft	OK	—
ShamrockGetAutoSlitWidth (spectrometer)	Returns the specified slit width. Which slit is asked can be specified by slit_index: 1 = input slit side 2 = input slit direct 3 = output slit side 4 = output slit direct	Int: device Int: slit_index	Float: width	OK	—

¹Not all readout modes have been tested yet in respect to `andorpy.ccd.GetAcquiredData`, since Full Vertical Binning is usually the appropriate choice.

ShamrockGetCalibration (spectrometer)	Obtains the wavelength calibration of each pixel of attached sensor.	Int: num_px ² Int: device	Float Array: wavelength (nm)	OK	ShamrockSetNumber-Pixels and ShamrockSetPixelWidth must have been called with the correct parameters. Otherwise this function will return only zeros.
ShamrockGetCCDLimits (spectrometer)	Gets the upper and lower accessible wavelength through the port. That is not equal to the maximum and minimum wavelength of the spectrum! To get this use ShamrockGetCalibration.	Int: device Int: port	Float: low Float: high	OK	—
ShamrockGetGrating (spectrometer)	Returns the current grating.	Int: device	Int: grating	OK	—
ShamrockGetGratingInfo (spectrometer)	Returns the grating information.	Int: device int: grating	Float: lines String: Blaze Int: home Int: offset	incomplete ³	lines = grating lines per mm blaze = grating blaze wavelength (nm) home = grating home (steps) offset = grating offset (steps)
ShamrockGetPixelWidth (spectrometer)	Gets the current value of the pixel width in microns of the attached sensor.	Int: device	Float: width (micrometer)	soon!	—
ShamrockGetShutter (spectrometer)	Returns the current device shutter mode. Available modes: 1 = open, 0 = closed, -1 = shutter not set	Int: device	Int: mode	OK	—
ShamrockGetWavelength (spectrometer)	Returns the current wavelength.	Int: device	Float: wavelength	OK	—
ShamrockGetWavelengthLimits (spectrometer)	Returns the Grating wavelength limits.	Int: device Int: grating	Float: low Float: high	OK	Those are not the limits of the spectrum!
ShamrockGotoZeroOrder (spectrometer)	Sets wavelength to zero order.	Int: device	none	OK	—
ShamrockInitialize (spectrometer)	Initializes the Shamrock driver. Makes ccd.Initialize redundant.	none	none	OK	Can take up to about 20 seconds. Check ccd.GetStatus after that ⁴ , because in some cases the CCDs initialization fails even though SUCESS is returned.
ShamrockSetAutoSlitWidth (spectrometer)	Sets the width of the specified slit.	Int: device Int: slit Float: width (micrometer)	none	OK	Setting the slit width to 0.0 will not result in a perfectly dark sensor.
ShamrockSetGrating (spectrometer)	Sets the required grating.	Int: device Int: grating	none	OK	ShamrockGetCalibration has to be called again after changing the grating!

²Number of pixels of the attached sensor.

³The blaze returns only "SUCCESS" and no value.

⁴should return DRV_IDLE

ShamrockSetNumber-Pixels (spectrometer)	Sets the number of pixels of the attached sensor.	Int: de- vice Int: pixels	none	OK	ShamrockGetCalibration will return only zeros, if this functions call has been forgotten.
ShamrockSetPixel-Width (spectrometer)	Sets the pixel width in microns of the attached sensor. Important for calibration	Int: de- vice Float: width	none	OK	The correct value for that width can be obtained by calling andorpy.ccd.GetPixelSize
ShamrockSetShutter (spectrometer)	Sets the device shutter mode.	Int: de- vice Int: mode	none	OK	Only works with Shamrock 303 series.
ShamrockSetWave-length (spectrometer)	Set the required wavelength.	Int: de- vice Float: wl	none	OK	This sets the wavelength, that is then the centre of the spectrum. Minimum and maximum wavelength can be obtained by ShamrockGetCalibration
ShutDown (ccd)	This function will close the Andor MCD system down.	none	none	OK	This is not nessecarry, if ShamrockClose has already been called.

B. Code for the Sardana Interface

```

"""
Macros to set up and perform measurements using Andor CCD and
    Shamrock spectrometer.
IMPORTANT FOR TESTING/CHANGING MACROS:
to restart macroserver: [%SardanaRestartMacroserver.py -x]
to reload macros: [%relmac andor-init] for one macro or [%relmaclib
    andor-macros.py] for
a whole file
"""

from __future__ import print_function

__all__ = ["andor_init", "andor_shutdown", "andor_settings", "
    andor_measure", "andor_status"]
__docformat__ = 'restructuredtext'

from sardana.macroserver.macro import Type, Macro, macro, ParamRepeat
    , iMacro
import PyTango
import json
import andorpy
import time
import os

def send_info(out_server):
    T = andorpy.ccd.GetTemperature()

```

```

g = andorpy.spectrometer.ShamrockGetGrating(0)

settings = json.dumps({"ccdtemp":T[1],
                      "grating":g[1]})

out_server.InfoString = settings

def set_defaults(cl):
    """
        This function sets all the settings of spectrometer and ccd
        back to default.
    """
    andorpy.spectrometer.ShamrockSetAutoSlitWidth(0,1,10.0)
    andorpy.ccd.SetTemperature(-80)
    andorpy.ccd.CoolerOn()
    andorpy.ccd.SetFanMode(0)
    andorpy.ccd.SetReadMode(0)
    andorpy.ccd.SetAcquisitionMode(2)
    andorpy.ccd.SetNumberAccumulations(2)
    andorpy.ccd.SetAccumulationCycleTime(0.7)
    andorpy.spectrometer.ShamrockSetShutter(0,1)
    andorpy.spectrometer.ShamrockSetGrating(0,2)
    andorpy.spectrometer.ShamrockSetWavelength(0, 485.0)
    andorpy.ccd.SetExposureTime(1.)
    andorpy.ccd.SetFilterMode(2)

    global endless
    endless = 0

    global triggermode
    triggermode = 0

    andorpy.ccd.SetTriggerMode(triggermode)

    # gather information for spectrometer calibration:
    num_pixels = andorpy.ccd.GetDetector() # width of the detector (
        number of pixels)
    px_dim = andorpy.ccd.GetPixelSize() # width of a pixel (
        micrometers)

    # hand over those data for spectrometer calibration:
    tmp1 = andorpy.spectrometer.ShamrockSetPixelWidth(0,px_dim[1])
    tmp1 = andorpy.spectrometer.ShamrockSetNumberPixels(0,num_pixels
        [1])

    cl.output("all settings set to default")

class andor_init(Macro):
    """

```

```

andor_init Macro:
    - Starts CCD and spectrometer
    - checks if CCD and spectrometer are in idle
      —> pushes error (please power off – restart) if this is
          not the case
    - starts cooler, sets temperature to -80 and fan to maximum
"""
def run(self):

    self.output("-- starting ccd and spectrometer")
    spec_status = andorpy.spectrometer.ShamrockInitialize()
    ccd_status = andorpy.ccd.GetStatus()

    # check, if both devices are responding:

    if (spec_status[0] == "SUCCESS" and ccd_status[1] == "code
        20073 DRV_IDLE"):
        self.output("-- decices started. setting all parameters
            to default ")
        set_defaults(self)
        self.output("ready")
    else:
        time.sleep(3)          # wait, then check again if
                               # responding
        ccd_status = andorpy.ccd.GetStatus()
        spec_status = andorpy.spectrometer.ShamrockGetGrating()

        if (spec_status[0] == "SUCCESS" and ccd_status[1] == "
            code 20073 DRV_IDLE"):
            tmp1 = andorpy.ccd.SetTemperature(-80)
            tmp1 = andorpy.ccd.CoolerOn()
            tmp1 = andorpy.ccd.SetFanMode(0)      # ... maximum
                                                  # cooling
            self.output("ready")
        else:
            self.output("initialisation fault – check connections
                and restart devices")
            self.output("spectrometer responding: " + spec_status
                [0])
            self.output("ccd status: " + ccd_status[1])

class andor_shutdown(Macro):

    def run(self):
        self.output("-- shutting down the ccd and spectrometer")
        andorpy.ccd.AbortAcquisition()
        andorpy.spectrometer.ShamrockClose()
        self.output("sucess")

```

```

class andor_settings(iMacro):
    """
        andor_settings Macro:
        - prints out all status data from spectrometer and ccd
        - gives interactive menu to change settings (exposure
          time, grating, slit width ...)
    """

    interactive = True

    def run(self):

        # ask for current status:
        ccd_status = andorpy.ccd.GetStatus()
        grating = andorpy.spectrometer.ShamrockGetGrating()

        global endless
        global triggermode

        if not (grating[0] == "SUCCESS" and ccd_status[0] == "code
        20002 DRV_SUCCESS" ):
            self.output("error: Spectrometer and/or CCD not
            initialized or malfunctioning.")
            self.output("          —> Run andor_init ")
            self.output("          —> Poweroff+check+restart hardware,
            if andor_init also fails.")

        elif (ccd_status[1] == "code 20073 DRV_IDLE"):
            slit = andorpy.spectrometer.ShamrockGetAutoSlitWidth(0,1)
            shutter = andorpy.spectrometer.ShamrockGetShutter()
            wl = andorpy.spectrometer.ShamrockGetWavelength(0)
            T = andorpy.ccd.GetTemperature()
            timings = andorpy.ccd.GetAcquisitionTimings()
            fm = andorpy.ccd.GetFilterMode()
            os_d = andorpy.spectrometer.ShamrockGetDetectorOffset(0)
            os_g1 = andorpy.spectrometer.ShamrockGetGratingOffset
                (0,1)
            os_g2 = andorpy.spectrometer.ShamrockGetGratingOffset
                (0,2)
            os_g3 = andorpy.spectrometer.ShamrockGetGratingOffset
                (0,3)

            num_pixels = andorpy.ccd.GetDetector() # width of the
                detector (number of pixels)
            tmp1 = andorpy.spectrometer.ShamrockGetCalibration(0,
                num_pixels[1])
            calib = tmp1[1]

            self.output("")

```

```

self.output("##### CURRENT SETTINGS #####")
self.output("")
self.output("MEASUREMENT: ")
self.output("(a) Spectrometer grating selected: " + str(
    grating[1]) + " (present: 1,2,3)")
self.output("(b) Spectrometer slit width: " + str(slit
    [1]) + " micrometer")
self.output("(c) Spectrometer shutter status: " + str(
    shutter[1]) + " (0=closed, 1=open)")
self.output("(d) Spectrometer central wavelength: " + str
    (wl[1]) + "nm")
self.output("    —> min = " + str(calib[0])+"nm, max = "
    + str(calib[len(calib)-1])+" nm")
self.output("(e) CCD temperature: " + str(T[1]) + "C,
    status = " + str(T[0]))
self.output("(f) CCD exposure time: " + str(timings[1]) +
    "s")
self.output("(g) CCD cosmic particle filter: " + str(fm
    [1]) + " (0=off, 2=on)")
self.output("(h) Endless measurement enabled: " + str(
    endless)+ " (0=off, 1=on)")
self.output("(i) Trigger mode: " + str(endless)+ " (0=
    internal, 1=external)")
self.output("")
self.output("CALIBRATION:")
self.output("(j) Detector offset: " +str(os_d[1]))
self.output("(k) Offset grating 1: " + str(os_g1[1]))
self.output("(l) Offset grating 2: " + str(os_g2[1]))
self.output("(m) Offset grating 3: " + str(os_g3[1]))
self.output("")
self.output("(y) Set everything back to default")
self.output("(z) Quit")
self.output("")

choice = self.input("Enter letter of setting to change:")
if not(choice == "y" or choice == "z"):
    value = self.input("Enter new value:")

if choice == "a":
    tmp1 = andorpy.spectrometer.ShamrockSetGrating(0,int(
        value))
    self.output(tmp1[0])
elif choice == "b":
    tmp1 = andorpy.spectrometer.ShamrockSetAutoSlitWidth
        (0,1,float(value))
    self.output(tmp1[0])
elif choice == "c":
    tmp1 = andorpy.spectrometer.ShamrockSetShutter(0,int(
        value))

```

```

        self.output(tmp1[0])
    elif choice == "d":
        tmp1 = andorpy.spectrometer.ShamrockSetWavelength(0,
            float(value))
        self.output(tmp1[0])
    elif choice == "e":
        tmp1 = andorpy.ccd.SetTemperature(int(value))
        self.output(tmp1[0])
    elif choice == "f":
        tmp1 = andorpy.ccd.SetExposureTime(float(value))
        self.output(tmp1[0])
    elif choice == "g":
        tmp1 = andorpy.ccd.SetFilterMode(int(value))
        self.output(tmp1[0])
    elif choice == "h":
        val = int(value)
        if val == 0:
            endless = val
            self.output("endless measurement disabled")
        elif val == 1:
            endless = val
            self.output("endless measurement enabled")
        else:
            self.output("invalid choice")
    elif choice == "i":
        val = int(value)
        if val == 0:
            triggermode = val
            andorpy.ccd.SetTriggerMode(triggermode)
            self.output("internal trigger enabled")
        elif val == 1:
            triggermode = val
            self.output("external trigger enabled")
            andorpy.ccd.SetTriggerMode(triggermode)
        else:
            self.output("invalid choice")
    elif choice == "j":
        tmp1 = andorpy.spectrometer.ShamrockSetDetectorOffset
            (0,int(value))
        self.output(tmp1[0])
    elif choice == "k":
        tmp1 = andorpy.spectrometer.ShamrockSetGratingOffset
            (0,1,int(value))
        self.output(tmp1[0])
    elif choice == "l":
        tmp1 = andorpy.spectrometer.ShamrockSetGratingOffset
            (0,2,int(value))
        self.output(tmp1[0])
    elif choice == "m":

```

```

        tmp1 = andorpy.spectrometer.ShamrockSetGratingOffset
            (0,3,int(value))
        self.output(tmp1[0])
    elif choice == "y":
        set_defaults(self)
    elif choice == "z":
        self.output(" ")
    else:
        self.putput("invalid choice")

    else:
        self.output("error: CCD is busy. Probably some
            measurement is still running.")

class andor_measure(Macro):
    """
    andor_measure Macro:
        - starts measurement
        - loads data from device to PC
        - !!! stores data as htf5 format file
        - sends data to Tango server
        - !!! clears device memory after data are saved
        - prints out path of data files in console
    """
    def run(self):

        global endless
        global triggermode

        andor_display = PyTango.DeviceProxy("hasep23dev:10000/p23/
            andor_display/dev.01")

        # ask for status of devices:
        grating = andorpy.spectrometer.ShamrockGetGrating(0)
        ccd_status = andorpy.ccd.GetStatus()
        detector = andorpy.ccd.GetDetector()
        width = detector[1]          # this is the number of pixels

        if not (grating[0] == "SUCCESS" and ccd_status[0] == "code
            20002 DRV_SUCCESS" ):
            self.output("error: Spectrometer and/or CCD not
                initialized or malfunctioning.")
            self.output("          —> Run andor_init ")
            self.output("          —> Poweroff+check+restart hardware,
                if andor_init also fails.")

        elif (ccd_status[1] == "code 20073 DRV_IDLE"):

            # load the array for every wavelength axis tick (nm):

```



```

tmp1 = andorpy.spectrometer.ShamrockGetCalibration(0,
width)
calib = tmp1[1]
calib = calib[::-1] # reverse the axis calibration
# —> otherwise spectrum is shown mirrored to
middle wavelength

tmp1 = andorpy.ccd.GetAcquisitionTimings()
t_abs = tmp1[2]

if endless == 0:
    self.output("— starting measurement. This one will
take " + str(t_abs) + "s")
    if triggermode == 1: self.output("— waiting for
external trigger ")

    # now start measuring:
    andorpy.ccd.StartAcquisition()
    status = andorpy.ccd.GetStatus()

    while (status[1].startswith('code 20072')):
        status = andorpy.ccd.GetStatus() #this loops
until acquisition is over

    # get data from device and load them to sardana:
    tmp1 = andorpy.ccd.GetAcquiredData(width)
    data = tmp1[1]
    andor_display.AndorData = data
    andor_display.AndorDataCalib = calib
    send_info(andor_display)

    self.output("Success.")
    self.output("Data stored to Sardana server hasep23dev
:10000/p23/andordisplay/dev.01")
    andorpy.ccd.FreeInternalMemory()

else:
    self.output("— starting endless measurements. Each
one will take " + str(t_abs) + "s")
    self.output("NOTE: Ctrl + C to stop measurement!")

    if triggermode == 1: self.output("— waiting for
external trigger ")

    while True:
        # now start measuring:
        andorpy.ccd.StartAcquisition()
        status = andorpy.ccd.GetStatus()

```

```

        while (status[1].startswith('code 20072')):
            status = andorpy.ccd.GetStatus()          #this
                                                    loops until acquisition is over

            # get data from device and load them to sardana:
            data = andorpy.ccd.GetAcquiredData(width)
            andor_display.AndorData = data[1]
            andor_display.AndorDataCalib = calib
            send_info(andor_display)
            andorpy.ccd.FreeInternalMemory()
            self.checkPoint()    # for not confusing the
                                MacroServer by Ctrl+C
            self.output("Data stored to Sardana server
                        hasep23dev:10000/p23/andordisplay/dev.01")

    else:
        self.output("error: CCD is busy. Probably some other
                    measurement is still running.")

class andor_status(Macro):
    """
        This only communicates with devices and gives back the
        response.
        Useful for debugging.
    """
    def run(self):
        ccd_status = andorpy.ccd.GetStatus()
        grating = andorpy.spectrometer.ShamrockGetGrating()
        self.output("CCD responding:  " + ccd_status[0] + ccd_status
                    [1])
        self.output("Spectrometer responding:  " + grating[0])

```