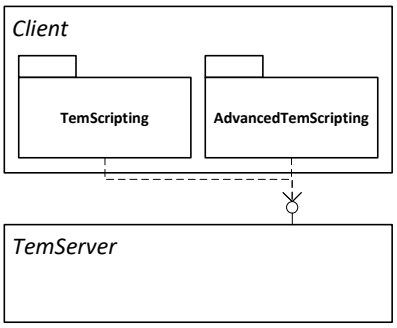
(Advanced) TEM Scripting User Guide

Document status: v2.0 approved

Introduction

TEM Scripting is a scripting adapter component that enables customers and 3rd party applications to automate the operation of their TEM microscope. It provides an interface through which scripts can control all TEM subsystems and perform acquisitions with the available detectors.

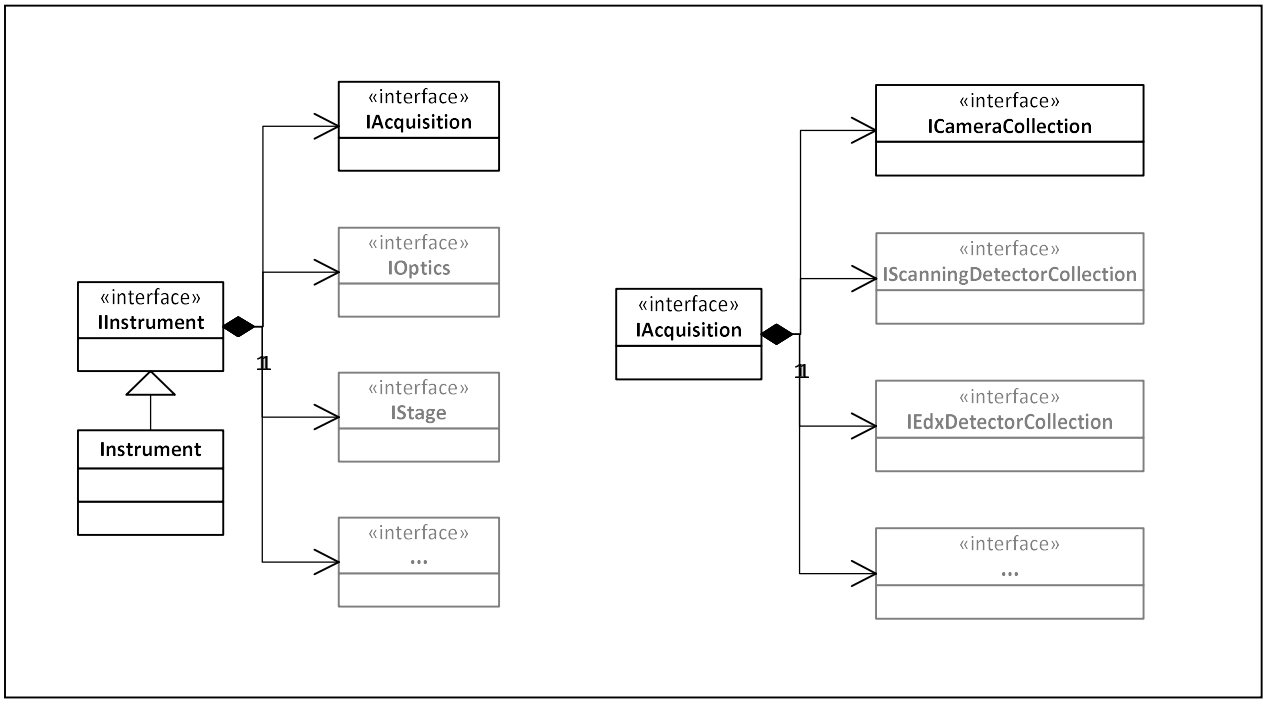


The interface is limited in scope such in order to make it stable across many TEM Server releases.

The existing TEM Scripting component does not properly support the Falcon cameras especially for handling the dose fractions that these cameras produce.

A new component is provided that directly interfaces with the TEM Server, providing access to single image acquisitions from the Falcon cameras. The component has been named AdvancedTemScripting.

Interface Structure and Concepts



The Advanced Scripting component provides a COM-based interface. COM is broadly supported in various scripting (and more general, application) environments. The set of interfaces is structured such

as to allow easy usage from single-threaded clients (such as scripts): only calls, no events or callbacks, synchronous calls, polling for state, no concurrency.

Navigation

The main interface is `IInstrument`. From this, all other interfaces are reached by navigating through a tree-like structure. The Instrument object that implements `IInstrument` is creatable and the created instance is owned by the client (script) that created it. As long as the client holds on to the `Instrument` object, it can use it to navigate the subsystem tree. Discarding the Instrument object breaks the connection of the client to the server and renders any references obtained during navigation invalid.

```
Camera FindCameraByName(string name, CameraList cameras)
{
  foreach(Camera c in cameras) if (c.Name == name) return c;
  return null;
}

var instrument = new TEMAdvancedScripting.Instrument();
var csa = instrument.Acquisitions.CameraSingleAcquisition;
var allCameras = instrument.Acquisitions.Cameras;
var singleAcquisitionCameras = csa.SupportedCameras;
var falconCamera = FindCameraByName("BM-Falcon", singleAcquisitionCameras);
csa.Camera = falconCamera;
var cs = csa.CameraSettings;
```

Copy vs Reference

Simple objects are returned as copies (aka "data" objects"). Any changes to such objects are not applied to the underlying system. This currently only applies to objects like the `FrameRange`. For example:

```
FrameRangeList dfd = cs.DoseFractionsDefinition;
dfd[0].Begin = 2; // This will not modify the range value
```

Objects that are parts of the navigation tree are returned by reference and any action or change performed on them affects the state of the scripting component. For example, changing values on the `CameraSettings` object retrieved through the `CameraSingleAcquisition` will apply the changed values:

```
cs.ExposureTime = 0.25; // This will be applied
```

Objects that originate in capabilities are read-only values that can be selected from a list and applied. This currently refers to `Binning`:

```
BinningList binnings = cc.SupportedBinnings;
cs.Binning = binnings[0];
```

Capability-driven

To avoid an extensive system of interface inheritance the interfaces are designed to be capability-driven. This means that starting with a basic set of operations and properties on the main interface of e.g. a detector other sets of operations and properties (or features) that more specialized versions of the detector provide are organized into navigable subsystems or subsets of operations and properties.

For such cases test functions are provided (Has__, Is__, Can__, Supports__) that can be used before accessing the respective setting or operation.

Threading

The interfaces are not meant to be accessed from multiple threads concurrently.

Supported Cameras

The `Acquisitions` object contains all the use-cases and detectors available. The list of cameras returned through the `Acquisitions` object contains all the cameras that `TEMAdvancedScripting` supports. Currently, these are: Ceta 1, Ceta 2, Falcon 3 and Falcon 4.

Only one use-case is supported at this moment: the `CameraSingleAcquisition`.

The list of cameras returned by `CameraSingleAcquisition` contains the subset of all the cameras that can participate in the use-case, which actually are all cameras.

```
var instrument = new TEMAdvancedScripting.Instrument();
var allCameras = instrument.Acquisitions.Cameras;
var singleAcquisitionCameras =
    instrument.Acquisitions.CameraSingleAcquisition.SupportedCameras;
```

Performing Camera Acquisitions

The example code for performing a simple single image acquisition looks like this:

```
Camera FindCameraByName(string name, CameraList cameras)
{
  foreach(Camera c in cameras) if (c.Name == name) return c;
  return null;
}

var instrument = new TEMAdvancedScripting.Instrument();
var csa = instrument.Acquisitions.CameraSingleAcquisition;
var allCameras = instrument.Acquisitions.Cameras;
var singleAcquisitionCameras = csa.SupportedCameras;
var falconCamera = FindCameraByName("BM-Falcon", singleAcquisitionCameras);
csa.Camera = falconCamera;
var cs = csa.CameraSettings;

// Perform any settings necessary; some examples:
cs.ExposureTime = cs.ExposureTimeRange.Max;
if (cs.SupportsDriftCorrection) cs.AlignImage = true;

// Here: Optionally, setup the dose fractions
dfd = cs.DoseFractionsDefinition;
dfd.Clear();
dfd.AddRange(fs1, fe1);
dfd.AddRange(fs2, fe2);
//...
dfd.AddRange(fsn, fen);
assert(fsx < fex);
assert(fen <= cs.CalculateNumberOfFrames());

// Perform the acquisition; the final image is received as a return value
var img = csa.Acquire();
var metadata = img.Metadata;

// Here: Save or otherwise use the image and metadata
// ...
```

```
// Wait for the camera to become available for another acquisition
csa.Wait();
// or
// while (csa.IsActive) sleep(some);

// Here: Access and process images available at cs.PathToImageStorage
```

Figure 1 Basic single-image camera acquisition

When the `instrument` object is released, all other objects become invalid references that can no longer be used.

To perform an acquisition with a camera, it has to be added to a use-case. In the example above, the use-case only accepts one camera. Then the acquisition is triggered by calling the `Acquire` method on the single acquisition use-case. Configuration of the camera takes place through the `CameraSettings` object.

Dose Fractions

A single acquisition returns one image called the "final image" which is the integrated image over the exposure time. The Falcon cameras can also provide intermediate images called "dose fractions". The dose fractions are temporarily stored on the camera device itself until they get downloaded.

When defining fractions the client needs to know the number of frames that the acquisition will produce. This depends on acquisition settings, especially on the exposure time. So it is wise to first configure all settings and leave dose fractions for last, when the number of frames has settled.

The TEM scripting component will download the dose fractions to either a local directory (hardcoded to: c:\OffloadData) or to a storage server (hardcoded to: \\192.168.10.2\OffloadData) before allowing the client to perform a new acquisition. The download itself takes place asynchronously so the script may perform other operations while the download is in progress. Before attempting to start a new acquisition the script should `Wait()` for the previous one to finish.

After the `Wait()` returns the images will be available at the location indicated by `PathToImageStorage` which can either be a local directory or a UNC path to a directory on the storage.

EER

The Falcon 4 camera supports storing every captured electron counted camera frame within the EER file format. EER is a "special" dose fraction scheme and downloading of the images (to the storage server) is handled the same as for Dose Fractions, only the file format and extension differs.

EER can only be enabled in case Electron Counting is enabled and can not be used in combination with dose fractions.

File naming

Please note that in the examples .mrc is shown as file extension, in case EER is enabled the extension will be .eer

When nothing else is specified, the intermediate images are saved in a file that is named with a timestamp composed of the current date and time: `YYYYMMDDhhmmss.`

The client can specify a pattern to be used for the file and subdirectory name:

```
cs.SubPathPattern = "dir\\file";
```

The pattern is given in the form of a relative path in which the last element is interpreted as the file name. Also, special tags can be used to be replaced with current values. The tags supported are:

- `{ymd}` – replaced with the current date in the format `YYYYMMDD`
- `{hms}` – replaced with the current time in the format `hhmmss`

Examples of valid formats:

- `"first_acquisition"` – a file will be created called `first_acquisition.mrc`
- `"my_study\\first_acquisition"` – a file will be created called `first_acquisition.mrc` inside the subdirectory called `my_study`
- `"my_study\\"` – file names will be formatted `YYYYMMDDhhmmss.mrc` and will be saved in subdirectory `"my_study"`
- `"day_{ymd}\\acquisition_{hms}"` – file names will be formatted as `acquisition_hhmmss.mrc` and placed in the subdirectory `day_YYYYMMDD`

Image and file formats

Dose Fractions

The Storage Server saves acquired images in the MRC file format with an associated XML file containing metadata information. It uses one MRC-XML pair for all the dose fractions and one for the final image.

The same strategy has been implemented in the Advanced Scripting too. Images saved locally also use the MRC-XML format.

The MRC format can only contain images of up to 16-bits per pixel, thus all images will internally be adapted to fit inside this format. The (down-)scaling factor used for the pixel values is multiplied into the `PixelValueToCameraCounts` metadata factor. For a client to get the actual pixel value as read by the camera (called `CameraCounts`) it must multiply each pixel value in the image by the `PixelValueToCameraCounts` factor in the metadata.

EER

The Storage Server saves acquired images in the EER file format, containing the metadata and every captured electron counted camera frame.

Final image

Final Images returned in memory are not converted, so the client must be able to accept 16-bit and 32-bit images. See the interface description for more details.

Metadata

Metadata information is returned together with the in-memory image. For the dose fractions, metadata is saved in an XML file with the same name as the MRC file containing the images.

Every metadata parameter can be read as a string or a COM variant.

The following metadata is expected to be present:

- `TimeStamp`
- `DetectorName`
- `AcquisitionUnit`
- `ImageSize.Width`
- `ImageSize.Height`
- `Encoding`
- `BitsPerPixel`
- `Binning.Width`
- `Binning.Height`
- `ReadoutArea.Left`
- `ReadoutArea.Top`
- `ReadoutArea.Right`
- `ReadoutArea.Bottom`
- `ExposureTime`
- `DarkGainCorrectionType`
- `Shutters[0].Type`
- `Shutters[0].Position`
- `PixelValueToCameraCounts`
- `AlignIntegratedImage`
- `DriftCorrected` (optional, only for drift corrected images)
- `DriftCorrectionConfidence` (optional, only for drift corrected images)
- `DriftCorrectionClipping` (optional, only for drift corrected images)
- `DriftCorrectionVectorX` (optional, only for drift corrected images)
- `DriftCorrectionVectorY` (optional, only for drift corrected images)
- `ElectronCounted`
- `CountsToElectrons` (optional, only for electron counted images)

Failures

No license

If the feature requested is not licenced, the command will be refused and an error will occur.

Inconsistency in parameters

In some cases it is possible that the set of parameters chosen by the client is inconsistent. In such a situation the acquisition will fail and the exception text will indicate which parameters have been adjusted to make the set consistent.

Camera in use

A camera can only be used in one acquisition at one time. If another client (script or other) is using the camera `Acquire()` will fail. `Acquire()` can also fail when the camera isn't used in another acquisition but cannot be inserted due to a physical conflict (i.e. another camera is inserted) or a defect.

Camera offline

A camera may be present in the system configuration but may be unavailable due to e.g. a network cable disconnect or hardware being powered off. In such cases `Acquire()` will fail and the text of the exception will indicate the most likely cause. Other operations on the camera may also fail in such a situation.

Storage server unavailable

The Storage Server may become (temporarily) unavailable. The acquisition will fail in case images should be downloaded to the storage server (e.g. Dose Fractions or EER being enabled).

Camera Support

The scripting adapter component mainly supports the Falcon cameras. It also supports the Ceta and Ceta2 cameras but only for very limited single acquisition scenarios.

Interface Description

`IInstrument` Interface

The entry into the AdvancedScripting interface.

Acquisitions

- Sort: property, read-only
- Type: Acquisitions
- Description: returns the list of all implemented use cases and detectors

Phaseplate

- Sort: property, read-only
- Type: Phaseplate
- Description: returns the phaseplate interface

`Acquisitions` interface

Represents the list of all implemented use cases and the list of all implemented detectors available in the system. Only camera detectors are implemented at the moment.

Cameras

- Sort: property, read-only
- Type: CameraList
- Description: returns a list of all cameras supported by the AdvancedScripting component

CameraSingleAcquisition

- Sort: property, read-only
- Type: CameraSingleAcquisition
- Description: returns a CameraSingleAcquisition object that will be used to perform single acquisitions using cameras

`CameraSingleAcquisition` interface

Represents the single acquisition (one image) use case performed with a camera. The Falcon camera's allow the acquisition of intermediate images.

SupportedCameras

- Sort: property, read-only
- Type: CameraList
- Description: returns a list of cameras that support the single acquisition use-case (which at this time are all cameras

Camera

- Sort: property, write-only
- Type: Camera
- Description: associates one camera with this use-case; this camera will perform the use-case; if no camera has been associated with the use-case prior to starting the acquisition, the acquisition will fail

CameraSettings

- Sort: property, read-only
- Type: CameraSettings
- Description: this is a view on the camera where settings can be applied; when no camera is set in the use-case – through the Camera property – attempting to read this property will fail; also, settings will be remembered for each given camera, even after it has been taken out of the use-case

Acquire

- Sort: method/function
- Return type: AcquiredImage
- Description: triggers the start of the acquisition; fails if settings not coherent; returns when the final (integrated) image has been acquired

IsActive

- Sort: property, read-only
- Type: boolean (VARIANT_BOOL)
- Description: used to poll for the end of the complete acquisition; used as a non-blocking wait for the downloading or offloading of intermediate images; no Acquire can be done while IsActive is true

Wait

- Sort: method/function
- Return type: none
- Description: blocking wait for the end of the complete acquisition

`AcquiredImage` interface

Represent the final image acquired by the CameraSingleAcquisition.

Width

- Sort: property, read-only
- Type: long
- Description: the width of the image

Height

- Sort: property, read-only
- Type: long
- Description: the height of the image

PixelType

- Sort: property, read-only
- Type: ImagePixelType
- Description: the PixelType of the image, can be ImagePixelType_UnsignedInt, ImagePixelType_SignedInt or ImagePixelType_Float

BitDepth

- Sort: property, read-only
- Type: long
- Description: the bit depth of the pixels in the image

Metadata

- Sort: property, read-only
- Type: KeyValuePairList
- Description: the metadata belonging to the image

AsSafeArray

- Sort: property, read-only
- Type: `SAFEARRAY(long)`
- Description: The image is returned as SAFEARRAY(long). The final image has been converted to long without any scaling. One can use the methods BitDepth and PixelType to retrieve the original format.

AsVariant

- Sort: property, read-only
- Type: `VARIANT`
- Description: The image is returned as VARIANT and is not converted, so the client must be able to accept 16-bit and 32-bit images. One can use the vt member of the VARIANT to get the datatype of the underlaying data.

SaveToFile

- Sort: method/function
- Input arguments: `BSTR filePath, VARIANT_BOOL bNormalize`
- Return type: none
- Description: save the image as raw image on disk. Filepath indicates the location where the image should be stored and bNormalize indicated if the image should be normalized. bNormalize has the default value VARIANT_FALSE

`Camera` interface

Represents a camera device and its use case independent properties.

Name

- Sort: property, read-only
- Type: string (BSTR)
- Description: the conventional name of the camera used to identify it throughout the TEM server

Width, Height

- Sort: property, read-only
- Type: long
- Description: the count of pixels along each axis

PixelSize

- Sort: property, read-only
- Type: PixelSize
- Description: the physical size of a pixel (in m) along each axis

Insert, Retract

- Sort: method/function
- Return type: none
- Description: inserts/retracts the camera; returns only when the camera is fully inserted of retracted; fails with an exception if problems encountered

IsInserted

- Sort: property, read-only
- Type: Boolean (VARIANT_BOOL)
- Description: checks the insertion status of the camera; may fail with exception in case of (communication) problems

`CameraSettings` interface

Represents the set of use case dependent settings that are to be used in the acquisition for the camera.

Capabilities

- Sort: property, read-only

- Type: Capabilities
- Description: returns the set of use case dependent capabilities for the camera

PathToImageStorage

- Sort: property, read-only
- Type: string (BSTR)
- Description: returns the base directory where the intermediate images will be saved;

SubPathPattern

- Sort: property, read-write
- Type: string
- Description: sets and retrieves the subpath pattern for the storage of intermediate images; this subpath is to be appended to PathToImageStorage to get the full path to the images; it can be set before each acquisition to place intermediate images grouped by acquisition

ExposureTime

- Sort: property, read-write
- Type: double
- Description: sets and retrieves the exposure time for the acquisition (in s); the actual exposure time can be slightly higher or slightly lower that the one set

ReadoutArea

- Sort: property, read-write
- Type: enumeration: Full, Half, Quarter
- Description: sets and retrieves the area to be read from the camera sensor; the area is defined around the center of the sensor, horizontally as well as vertically

Binning

- Sort: property, read-write
- Type: Binning
- Description: sets and retrieves the binning factor to be used with the acquisitions; it has a horizontal and a vertical component; it must be one of the values retrieved by the SupportedBinnings property in CameraCapabilities

DoseFractionsDefinition

- Sort: property, read-only
- Type: FrameRangeList; a list of pairs of long values (Begin, End) where Begin is the first frame, End is the last frame plus 1
- Description: retrieves the list of frame ranges that define the intermediate images; the list can be cleared and filled with the ranges; for Ceta 1 & 2 it throws and exception

AlignImage

- Sort: property, read-write

- Type: boolean (BSTR)
- Description: sets and retrieves whether frame alignment (i.e. drift correction) is to be applied to the final image as well as the intermediate images

ElectronCounting

- Sort: property, read-write
- Type: Boolean (BSTR)
- Description: sets and retrieves whether electron counting mode is to be used in the acquisition; In case electron counting is not supported it will throw.

EER

- Sort: property, read-write
- Type: Boolean (BSTR)
- Description: sets and retrieves whether EER mode is enabled; In case EER is not supported it will throw. ElectronCounting should always be enabled when selecting EER.

CalculateNumberOfFrames

- Sort: method/function
- Return type: long
- Description: retrieves the number of frames that the acquisition produces; it is dependent on other settings so it is better called last; it is used to determine the ranges for the DoseFractionsDefinitions

`CameraAcquisitionCapabilities` interface

Represents the use case dependent capabilities of the camera. Defines acceptable features and ranges of acceptable values for the settings.

SupportedBinnings

- Sort: property, read-only
- Type: BinningList
- Description: returns a list of binnings supported by the camera; one of the binnings in this list can be used to set the Binnning setting

ExposureTimeRange

- Sort: property, read-only
- Type: ITimeRange
- Description: returns the range of exposure times supported by the camera

SupportsDoseFractions

- Sort: property, read-only
- Type: Boolean (VARIANT_BOOL)

- Description: returns whether the camera supports the acquisition of dose fractions; for cameras that don't support it, attempting to specify dose fractions will fail with an exception

MaximumNumberOfDoseFractions

- Sort: property, read-only
- Type: long
- Description: returns the maximum number of dose fractions that the camera supports;

SupportsDriftCorrection

- Sort: property, read-only
- Type: Boolean (VARIANT_BOOL)
- Description: returns whether the camera supports drift correction; attempting to use drift correction (AlignImage) for cameras that don't support it will lead to an exception

SupportsElectronCounting

- Sort: property, read-only
- Type: Boolean (VARIANT_BOOL)
- Description: returns whether the camera supports the electron counting mode; attempting to use the electron counting mode for cameras that don't support it will lead to an exception

SupportsEER

- Sort: property, read-only
- Type: Boolean (VARIANT_BOOL)

Description: returns whether the camera supports the EER; attempting to use EER for cameras that don't support it will lead to an exception

`Phaseplate` interface

Represents the phaseplate aperture which resides on the Objective ApertureMechanism. The phaseplate is an aperture with a thin layer of material, which when hit with an electron beam only allows electrons with a high enough energy level pass. This material gets saturated at the point where the beam is located, and therefore it is required to periodically change over to a new location within the aperture. A select number of predefined locations is available, which can be looped over by calling a 'next' function.

Preconditions: In order to be able to successfully call these methods, the following must be true:

- The microscope contains a motorized objective aperturemechanism
- The server is running
- The objective aperturemechanism is enabled, and a phaseplate aperture is selected
- The appropriate license must be available

SelectNextPresetPosition

- Sort: method

- Return type: none
- Description: Goes to the next preset location on the aperture

GetCurrentPresetPosition

- Sort: property, read-only
- Type: long
- Description: returns the zero-based index of the current preset position.

Client script example

```
# Sample python script to show usage of Advanced Scripting
#
import comtypes
import comtypes.client as cc

instrument = cc.CreateObject("TEMAdvancedScripting.AdvancedInstrument")

def TakeSingleAcquisition():
      csa = instrument.Acquisitions.CameraSingleAcquisition
      cams = csa.SupportedCameras
      csa.Camera = cams[[c.name for c in cams].index("BM-Falcon")]
      cs = csa.CameraSettings
      cs.DoseFractionsDefinition.Clear
      cs.DoseFractionsDefinition.AddRange(0, 1)
      cs.DoseFractionsDefinition.AddRange(1, 2)
      cs.DoseFractionsDefinition.AddRange(2, 3)
      cs.SubPathPattern = "Jupyter_{ymd}\\{hms}"
      ai = csa.Acquire()
      ai.SaveToFile("c:\\users\\factory\\desktop\\myimage.raw")
      array = ai.AsSafeArray

def PhaseplateSelectNext():
      print("PhaseplateSelectNext")
      pp = instrument.Phaseplate
      pp.SelectNextPresetPosition()
      print("PhaseplateSelectNext done")

def PhaseplateGetCurrent():
      print("PhaseplateGetCurrent")
      pp = instrument.Phaseplate
      print(pp.GetCurrentPresetPosition)
      print("PhaseplateGetCurrent done")

def PhaseplateGetCurrentAndSelectNext():
      print("PhaseplateGetCurrentAndSelectNext")
      pp = instrument.Phaseplate
      pp.SelectNextPresetPosition()
      print pp.GetCurrentPresetPosition
      print("PhaseplateGetCurrentAndSelectNext done")

def PhaseplateGetCurrentAndSelectNextAndGetCurrent():
      print("PhaseplateGetCurrentAndSelectNextAndGetCurrent")
      pp = instrument.Phaseplate
```

```
      for x in range(1,100):
            pp.SelectNextPresetPosition()
            print(pp.GetCurrentPresetPosition)
      print("PhaseplateGetCurrentAndSelectNextAndGetCurrent done")

print("----")
PhaseplateSelectNext()
print("----")
PhaseplateGetCurrent()
print("----")
PhaseplateGetCurrentAndSelectNext()
print("----")
PhaseplateGetCurrentAndSelectNextAndGetCurrent()
print("----")
print("All Done")
```