TemPackage Documentation
With TEMScripting and AdvancedTEMScripting Guides

Created By Maegan Jennings
15 December 2021 · BASF Ludwigshafen Am Rhein, Germany

Preface:

This guide goes over the commands that can be used to interface with the Thermo Fischer Talos TEM via Python. The microscope is accessed via the *comtyes.client* package for both the TEMScripting and the Advanced-TEMScripting Instruments.
This guide is split into three parts, each containing two sections:

- **TemPackage Guide**: Sections 1 and 2
  - These sections go over how to use the TemPackage that I have written! These 'wrapper' functions use both the Advanced-TEMScripting and the base TEMScripting to operate the microscope. You can use these functions (and add onto them) for creating scripts for the TEM.
- **Advanced TEMScripting Guide**: Sections 3 and 4
  - These sections go over how to use the Advanced TEMScripting microscope interface to either build your own script or interact with the TEM via command prompt.
- **TEMScripting Guide**: Sections 5 and 6
  - These sections go over how to use the TEMScripting microscope interface to either build your own script or interact with the TEM via command prompt.

Contents

1 TemPackage Overview

My package has a singluar class, the TEM class. Inside this are all of the functions that I have written for the operation of the microscope, with no subfunctions or heirarchy.

class TEM():

Read-Only	Write-Only	Read-Write
get_stage	set_stage	blank_beam
get_beam_shift	set_beam_shift	column_valve
get_image_shift	set_image_shift	
get_magnification	set_magnification	
get_mode	set_mode	
get_projection_mode	set_projection_mode	
get_vacuum_safety		
get_vacuum		
acquisition		
metadata		

All of the functions listed above are accessible using `TEM().functionname( )`. This is the list of all of the functions that are implemented into the TemPackage – if the function you're looking for isn't here, then it isn't in the TemPackage.

2 TemPackage Documentation

2.1 Importing the TemPackage.py file

If you have the TemPackage.py file in the same directory as the file that you are running, the python file should be able to find the TemPackage with just the following line in the document preamble.

```
import TemPackage
```

If you have the TemPackage.py file somewhere else on the machine and you want your code to be able to reference it, add these lines of code to the top of your notebook.

```
import sys
sys.AddToPath( Full path to the TemPackage.py file )
```

This will add the TemPackage.py file to the path of your notebook, so that way when you call `import TemPackage`, python will be able to find the file.

2.2 Initializing

Before using any of the functions, I would recommend running the following code:

```
from TemPackage import TEM
tem = TEM()
```

My package has a singluar class, the TEM class. Inside this are all of the functions that I have written for the operation of the microscope. The rest of this manual is written assuming that you have set `tem = TEM()`.

2.3 Function Documentation

2.3.1 Vacuum

Getting Vacuum Status

Function: `tem.get_vacuum()`
Possible Arguments: None – read-only function
Output: printed text

Important Info:
The TEM will give pressure values in Pascals when this function is called, but the TEM pc displays the vacuum in "Log" units. The conversion between Pascals and Log units are shown below:

$$L = 4.0011 * \ln(100 * P) + 75.14$$

P = pressure in Pascals & L = measured pressure in 'Log' units

Examples:

```
>>> tem.get_vacuum()

 Vacuum                        Value (Pa)
 Accelerator                   1.4e-07
 Column                        1.4e-07
 PIRco (unknown)               1.4e-07
 PPm (unknown)                 1.4e-07
 IGPf (unknown)                1.4e-07
```

Check Vacuum Safety

Function: `tem.get_vacuum_safety()`
Possible Arguments: None – read-only function
Output: `'Safe'` or `'Unsafe'`

Examples:

```
>>> tem.get_vacuum_safety()
'Safe'
```

2.3.2 Column Valve

Getting Column Valve

Function: `tem.column_valve()`
Possible Arguments: None – acts as a read function if no arguments input
Output: Printed text of the column valve status

Examples:

```
>>> tem.column_valve()
'Column Valve is Closed'
```

Setting Column Valve

Function: `tem.column_valve()`
Possible Arguments:

Keyword	Description
set	Entering "Open" will open the column valve, and entering "Close" will close the column valve. A confirmation message will print out.

Examples:

```
>>> tem.column_valve(set = 'Close')
'Column Valve Closed'
```

2.3.3 Blank Beam

Getting Beam Blank Status

Function: `tem.blank_beam()`
Possible Arguments: None – acts as a read function if no arguments input
Output: Printed text of whether or not the beam is currently blanked.

Examples:

```
>>> tem.blank_beam()
'Beam is not blanked'
```

Setting Beam Blank Status

Function: `tem.blank_beam()`
Possible Arguments:

Keyword	Description
blank	Entering `True` will blank the beam, and entering `False` will unblank the beam. Do not use quotation marks.

Examples:

```
>>> tem.blank_beam(blank = True)
```

2.3.4 Stage

Getting Stage Position

Function: `tem.get_stage()`
Possible Keyword Arguments: None – read-only function
Output: Dictionary

Keyword	Description
x	Value of the x axis position in scientific notation with units of meters
y	Value of the y axis position in scientific notation with units of meters
z	Value of the z axis position in scientific notation with units of meters
a	Value of the alpha angle as a decimal with units of radians
b	Value of the beta angle as a decimal with units of radians

Examples:

```
>>> position = tem.get_stage()
>>> tem.get_stage()
{'x': 0.0, 'y': 0.0, 'z': 0.0, 'a': 0.0, 'b': 0.0}
```

Setting Stage Position

Function: `tem.set_stage( **kwargs )`
Possible Keyword Arguments:

Keyword	Default	Description
x	None	Input is the value of the x axis position in meters. Both decimals (ex, 0.0000002) and scientific notation (ex, 2.0e-07) are acceptable.
y	None	Input is the value of the y axis position in meters. Both decimals (ex, 0.0000002) and scientific notation (ex, 2.0e-07) are acceptable.
z	None	Input is the value of the z axis position in meters. Both decimals (ex, 0.0000002) and scientific notation (ex, 2.0e-07) are acceptable.
a	None	Input is a decimal value of the alpha angle in radians.
b	None	Input is a decimal value of the beta angle in radians.
type	"GO"	Input can be either "GO" or "MOVE." Using "GO" will make the instrument move from position 1 directly to position 2. Using "MOVE" will make the instrument go from position 1 to stage zero to position 2.
speed	1.0	Input can be any decimal below 1.0. Inputting a speed below 1.0 will make the instrument move stage positions at a slower speed.

Important Info:
The speed's default setting on the TEM is 1.0 – to convert to radians per second:

$$\text{TEM} = 1.4768 * v + 0.0001$$

$$\text{TEM} = \text{speed in 'TEM units'} \quad \& \quad v = \text{ speed in radians/second}$$

Examples:

```
>>> tem.set_stage(x = 0.000002 , y = 0.0000006)
>>> tem.set_stage(a = 0.024, y = 2.3e-07, x = 1.2e-06)
>>> tem.set_stage(a = 0.04, speed = 0.01)
```

2.3.5 Beam Shift

Getting Beam Shift

Function: `tem.get_beam_shift()`
Possible Keyword Arguments: None – read-only function
Output: [x , y]

Keyword	Description
x	Value of the x axis beam shift
y	Value of the y axis beam shift

Examples:

```
>>> position = tem.get_beam_shift()
>>> tem.get_beam_shift()
[ 0.0, 0.0 ]
```

Setting Beam Shift

Function: `tem.set_beam_shift( **kwargs )`
Possible Keyword Arguments:

Keyword	Default	Description
x	None	Input is the value of the x axis shift in meters. Both decimals (ex, 0.0000002) and scientific notation (ex, 2.0e-07) are acceptable.
y	None	Input is the value of the y axis shift in meters. Both decimals (ex, 0.0000002) and scientific notation (ex, 2.0e-07) are acceptable.

Important Info:

The TEM uses a different set of unit vectors for the image shift ($\hat{u}$ and $\hat{v}$) that are defined by a linear combination of $\hat{x}$ and $\hat{y}$ (shown below). The $\hat{v}$ unit vector for the beam shift is the $-\hat{v}$ unit vector of the image shift. The internal parts of this function will convert the x and y inputs to the microscope axis and execute the correct shift that the user indicates.

$$\hat{v}_{\text{beam}} = -\hat{v}_{\text{image}}$$

$$\begin{pmatrix} u \\ v \end{pmatrix} = \frac{1}{6}\begin{pmatrix} 5.34 & 2.78 \\ -2.78 & 5.34 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix}$$

Examples:

```
>>> tem.set_beam_shifte(x = 0.000002)
>>> tem.set_beam_shift(y = 2.3e-07)
```

2.3.6 Image Shift

Getting Image Shift

Function: `tem.get_image_shift()`
Possible Keyword Arguments: None – read-only function
Output: [x , y]

Keyword	Description
x	Value of the x axis image shift
y	Value of the y axis image shift

Examples:

```
>>> position = tem.get_image_shift()
>>> tem.get_image_shift()
[ 1.2e-07, 0.0 ]
```

....

Setting Image Shift

Function: `tem.set_image_shift( **kwargs )`
Possible Keyword Arguments:

Keyword	Default	Description
x	None	Input is the value of the x axis image shift in meters. Both decimals (ex, 0.0000002) and scientific notation (ex, 2.0e-07) are acceptable.
y	None	Input is the value of the y axis image shift in meters. Both decimals (ex, 0.0000002) and scientific notation (ex, 2.0e-07) are acceptable.

Important Info:

The TEM uses a different set of unit vectors for the image shift ($\hat{u}$ and $\hat{v}$) that are defined by a linear combination of $\hat{x}$ and $\hat{y}$ (shown below). The $\hat{v}$ unit vector for the beam shift is the $-\hat{v}$ unit vector of the image shift. The internal parts of this function will convert the x and y inputs to the microscope axis and execute the correct shift that the user indicates.

$$\hat{v}_{\text{beam}} = -\hat{v}_{\text{image}}$$
$$\begin{pmatrix} u \\ v \end{pmatrix} = \frac{1}{6}\begin{pmatrix} 5.34 & 2.78 \\ 2.78 & -5.34 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix}$$

Examples:

```
>>> tem.set_image_shift(x = 0.000002)
>>> tem.set_image_shift(y = 2.3e-07)
```

2.3.7 Magnification

Magnification Index

The magnification index is an integer value that corresponds to the microscope's magnification value. For example, the magnification of 22,500x corresponds to the magnifiction index of 27 (see table below for full list of magnification indicies). This table is **specifically** for the Talos TEM

Talos TEM - TEM Imaging Mode Magnification

Index	TEM Display	Magnification	Index	TEM Display	Magnification
1	LM 25 x	25.0	23	SA 8600 x	8600.0
2	LM 34 x	34.0	24	SA 11000 x	11000.0
3	LM 46 x	46.0	25	SA 14000 x	14000.0
4	LM 62 x	62.0	26	SA 17500 x	17500.0
5	LM 84 x	84.0	27	SA 22500 x	22500.0
6	LM 115 x	115.0	28	SA 28500 x	28500.0
7	LM 155 x	155.0	29	SA 36000 x	36000.0
8	LM 210 x	210.0	30	SA 46000 x	46000.0
9	LM 280 x	280.0	31	SA 58000 x	58000.0
10	LM 380 x	380.0	32	SA 74000 x	74000.0
11	LM 510 x	510.0	33	SA 94000 x	94000.0
12	LM 700 x	700.0	34	SA 120 kx	120000.0
13	LM 940 x	940.0	35	SA 150 kx	150000.0
14	LM 1300 x	1300.0	36	SA 190 kx	190000.0
15	LM 1700 x	1700.0	37	SA 245 kx	245000.0
16	LM 2300 x	2300.0	38	SA 310 kx	310000.0
17	M 2050 x	2050.0	39	SA 390 kx	390000.0
18	M 2600 x	2600.0	40	SA 500 kx	500000.0
19	M 3300 x	3300.0	41	SA 630 kx	630000.0
20	SA 4300 x	4300.0	42	Mh 650 kx	650000.0
21	SA 5500 x	5500.0	43	Mh 820 kx	820000.0
22	SA 7000 x	7000.0	44	Mh 1.05 Mx	1050000.0

Talos TEM - STEM Imaging Mode Magnification

Index	TEM Display	Magnification	Index	TEM Display	Magnification
1	??	3860	14	??	500000
2	??	5000	15	??	714000
3	??	7140	16	??	1000000
4	??	10000	17	??	1430000
5	??	14300	18	??	2220000
6	??	22200	19	??	3330000
7	??	33300	20	??	5000000
8	??	50000	21	??	7140000
9	??	71400	22	??	10000000
10	??	100000	23	??	14300000
11	??	143000	24	??	22200000
12	??	222000	25	??	25000000
13	??	333000			

Getting Magnification

Function: `tem.get_magnification()`
Possible Keyword Arguments: None – read-only function
Output: string

Examples:

```
>>> tem.get_magnification()
'1700.0x Magnification.  Magnification Index ID = 15'
>>> tem.get_magnification()
'74000.0x Magnification.  Magnification Index ID = 32'
```

....

Setting Magnification

Function: `tem.set_magnification( value )`
Possible Keyword Arguments:

Options	Example	Description
integer	22	Inputting a plain integer will set the microscope to that magnification index.
+ integer	+7	Inputting a integer with a plus sign in front will set the microscope to the current magnification index *plus* the input value. In this example if the magnification index was 10, it will be set to 17.
- integer	-7	Inputting a integer with a minus sign in front will set the microscope to the current magnification index *minus* the input value. In this example if the magnification index was 10, it will be set to 3.

Important Info:

If the ±integer value makes the magnification index exceed the bounds, then the magnification will be set to the bound that it crossed. Ex. calling +10 when the magnification index is 40 will set the magnification index to 44.

Examples:

```
>>> tem.set_magnification(32)
>>> tem.set_magnification(-07)
```

2.3.8 Instrument Mode (STEM or TEM)

Getting Mode

Function: `tem.get_mode()`
Possible Arguments: None – read-only function
Output: string; either “TEM” or “STEM”

Examples:

```
>>> tem.get_mode()
'STEM'
>>> tem.get_mode()
'TEM'
```

....

Setting Mode

Function: `tem.set_mode( value )`
Possible Arguments:

Options	Description
“STEM”, “Stem”, or “stem”	Set the instrument mode to STEM mode.
“TEM”, “Tem”, or “tem”	Set the instrument mode to TEM mode.

Examples:

```
>>> tem.set_mode(``Stem")
>>> tem.set_mode(``TEM")
```

2.3.9 Projection Mode (Diffraction or Imaging)

Getting Projection Mode

Function: `tem.get_projection_mode()`
Possible Arguments: None – read-only function
Output: string; either “Diffraction” or “Imaging”

Examples:

```
>>> tem.get_projection_mode()
Diffraction'
>>> tem.get_projection_mode()
'Imaging'
```

....

Setting Projection Mode

Function: `tem.set_projection_mode( value )`
Possible Arguments:

Options	Description
“Diffraction”, “diffraction”, or “d”	Set the instrument to diffraction mode.
“Imaging”, “imaging”, or “i”	Set the instrument to imaging mode.

Examples:

```
>>> tem.set_projection_mode(`d')
>>> tem.set_projection_mode(`Imaging')
```

2.3.10 Acquisition

Background Info

Due to the nature of the TEM Scripting interface, there is no way to alter the camera properties and then take an image. Each time we use the TEM Scripting interface to access the camera, the default camera properties are called, even if they were just changed. Therefore, the acquisition function takes arguments to change the camera properties before acquiring the image.

The output of this function isn't directly usable; the reasons for this is processing the data out of the pointer takes a small amount of time, and it is easier to leave the output as a pointer when taking lots of images. When taking multiple acquisitions in a row, the pointer output of this function can be written into a list and accessed with indexing.

The output can be placed into `tem.toarray()` to access the image as a numpy array or `tem.metadata()` to access the stored metadata.

. . . .

Acquiring

Function: `tem.acquisition(cameraname, **kwargs )`
Possible Arguments: (Table Below)
Output: Pointer containing metadata and the image array.

Argument	Example	Description
cameraname	‘BM-Ceta’	(Required) The name of the camera (as a string) that you’d like to use for the image acquisition
Keywords	Default	Description
Exposure Time	1.0	The desired exposure time for the image
Binning	‘4k’	The image size you’d like the acquisition to produce. Options are ‘4k’, ‘2k’, ‘1k’, or ‘0.5k’
ReadoutArea	‘Full’	Sort of like a secondary binning, reads images at either full, half or quarter of their binned size. ex. a ‘1k’ image at ’Quarter’ readout area will produce a 256x256 image. Options are ‘Full’, ‘Half’, and ‘Quarter’.

Examples:

```
>>> im = tem.acquisition('BM-Ceta', ExposureTime = 2.0, Binning = '1k')
>>> im = tem.acquisition('BM-Ceta', ReadoutArea = 'Quarter', ExposureTime = 1.0)
```

2.3.11 Accessing Metadata

Background Info

The metadata list is 34 items long. I figured that sometimes you don’t need all of the metadata, so I wrote the function to accept arguments in a few ways.

. . . .

Getting Metadata

Function: `tem.metadata(image pointer, request string list )`
Possible Arguments: (Table Below)
Output: Pointer containing metadata and the image array.

Argument	Example	Description
image pointer	ai, im, etc	(Required) The pointer of the acquired image whose metadata you’d like to access.
request string	‘PixelSize’	(Required) The string list of all the metadata parameters you’d like to get. If it is a single argument, just a string like ’Pixelsize’ is acceptable. If it is multiple arguments, place into a list, like [’PixelSize’, ’ExposureTime’]. If you’d like all of the items to be output as a dictionary, use ’dict’. If you’d like all of the items to be printed as text, use ’all’

All possible Request Strings:

‘DetectorName’	‘Binning’	‘ReadoutArea’
‘ExposureMode’	‘ExposureTime’	‘DarkGainCorrectionType’
‘ShuttersType’	‘ShuttersPosition’	‘AcquisitionUnit’
‘BitsPerPixel’	‘Encoding’	‘ImageSize’
‘Offset’	‘PixelSize’	‘PixelUnit’
‘TimeStamp’	‘MaxPossiblePixelValue’	‘SaturationPoint’
‘DigitalGain’	‘CombinedSubFrames’	‘CommercialName’
‘FrameID’	‘TransferOK’	‘PixelValueToCameraCounts’
‘ElectronCounted’	‘AlignIntegratedImage’	

Examples:

```
>>> im = tem.acquisition('BM-Ceta', ExposureTime = 2.0, Binning = '1k')
>>> im = tem.acquisition('BM-Ceta', ReadoutArea = 'Quarter', ExposureTime = 1.0)
```

2.3.12 Saving Image as Array

From Pointer to Array

Function: `tem.to_array( image pointer )`
Possible Arguments: The image pointer
Output: A numpy array of the image stored within the input pointer

Examples:

```
>>> array1 = tem.to_array(im1)
>>> array2 = tem.to_array(im2)
```

---

3 Advanced TEMScripting Package Overview

This seems to be the structure of the package, and the next sections will go over how to use the functions.

Instrument
Acquisitions

- Cameras
- CameraSingleAcquisition
  - SupportedCameras
  - Camera
    - Name, Width, Height, PixelSize, Insert, Retract, IsInserted
  - CameraSettings
    - Capabilities (CameraAcquisitionCapabilities)
      - SupportedBinnings, ExposureTimeRange, SupportsDoseFractions, MaximumNumberOfDoseFractions, SupportsDriftCorrection, SupportsElectronCounting
    - PathToImageStorage, SubPathPattern, ExposureTime, ReadoutArea, Binning, DoseFractionsDefinition, AlignImage, ElectronCounting, CalculateNumberOfFrames
  - Acquire
    - Width, Height, PixelType, MetaData, AsSafeArray, AsVariant, SaveToFile
  - IsActive
  - Wait
- CameraContinuousAcquisition

Phaseplate

- SelectNextPresetPosition, GetCurrentPresetPosition

4 Advanced TEMScripting Instrument

4.1 Acquisitions

First you have to download and install the package comtypes, then you can create the Instrument,

```
   >>> import comtypes
   >>> import comtypes.client as cc
   >>> PhD = cc.CreateObject("TEMAdvancedScripting.AdvancedIstrument")
```

Once you create the instrument, you can find the camera that you want to be working with, and then can change the settings for that acquisition.

4.1.1 Cameras

This is a read only function that should return a list of all of the available cameras that the TEMAdvancedScripting instrument can access. If you can access the information in the pointer, you have a list of all of the cameras.

```
   >>> PhD.Acquisitions.Cameras
<Pointer>
```

4.1.2 CameraSingleAcquisition

Because all of the commands that we are going to use come from this directory, we want to write a new variable for a "CameraSingleAcquisition." Using the syntax as given in the TEMAdvancedScripting Guide, we just use 'csa' as this variable.

```
   >>> csa = PhD.Acquisitions.CameraSingleAcquisition
```

We will use 'csa' in the next sections.

SupportedCameras
The next step is to define the camera that we are going to be using to take the Single Acquisition. Use the following code to to this.

```
>>> cams = csa.SupportedCameras
>>> csa.Camera = cams[[c.name for c in cams].index("BM-Ceta")]
```

`csa.SupportedCameras` should output a pointer, and the next line goes through that pointer to find the camera whose name is "BM-Ceta." This is syntax taken directly from the Thermo Fischer TEMAdvancedScripting Guide.

Camera Options under **Camera** should be: **Name**, **Width**, **Height**, **PixelSize**, **Insert**, **Retract**, and **IsInserted**.

PixelSize

Getting Pixel Size

This is where things get a bit fishy because `cams.Camera.PixelSize` threw up a error that `PixelSize` was not a known argument under `cams.Camera`.

CameraSettings

Overview
Changes to the Camera Settings should remain in effect until the microscope object is disconnected from the TEM (when the script shuts down).

Options under **CameraSettings** should be: **Capabilities**, **PathToImageStorage**, **SubPathPattern**, **ExposureTime**, **ReadoutArea**, **Binning**, **DoseFractionsDefinition**, **AlignImage**, **ElectronCounting**, **EER**, **CalculateNumberOfFrames**, and **RecordingDuration**.

There are two ways to call the options: either create a new variable defined as the camera settings, or just type out the full command,

```
>>> cs = csa.CameraSettings
>>> cs.OptionTitleHere
>>> csa.CameraSettings.OptionTitleHere
```

....

Capabilities

This one has options within! This is how we find out the **SupportedBinnings**, **ExposureTimeRange**, **SupportsDoseFractions**, **MaximumNumberOfDoseFractions**, **SupportsDriftCorrection**, and **SupportsElectronCounting** options.

Supported Binnings

```
>>> bin = csa.CameraSettings.Capabilities.SupportedBinnings
<POINTER(BinningList) ptr=0x1b7fc2355d8 at 1b7fbd7e3c8 >
```

The binning list is iterable, so using `len(bin)` returns 4, and we can even call the binnings individually with `bin[i]` where $0 \leq$ `i` ≤ 3. However, that is where this command runs out of usefullness because I don't know how to access what each of the binning values actually are ...

```
>>> bin[0]
<POINTER(Binning) ptr=0x1b7fc2355d8 at 1b7fbd7e3c8>
setbin = bin[0]
```

`setbin` as defined above would be the format of the input needed to change the binning with `csa.CameraSettings.Binning` (discussed below).

Exposure Time Range

```
>>> expt = csa.CameraSettings.Capabilities.ExposureTimeRange
<POINTER(ITimeRange) ptr=0x1b7fc235728 at 1b7fc9a04c8>
```

This pointer is not iterable, and we can't find the length of it.

....

Exposure Time

Getting Exposure Time

Can use either commands below; the output will be a float of the current exposure time in seconds (I think) in both cases.

```
>>> cs.ExposureTime
>>> csa.CameraSettings.ExposureTime
```

Setting Exposure Time

Use either commands; the input is a float of the wanted exposure time in seconds (I think).

```
>>> cs.ExposureTime = 0.5
>>> csa.CameraSettings.ExposureTime = 0.5
```

....

Binning

Getting Binning

Can use either commands below; the output will be a pointer of the current binning setting in both cases.

```
>>> cs.Binning
>>> csa.CameraSettings.Binning
```

How do we find out what the binning is?? That is a great question.

Setting Binning

Use either commands; the input is a float of the wanted exposure time in seconds (I think). This is UNTESTED as of yet because idk what I would be setting the binning to.

```
>>> bin = csa.CameraSettings.Capabilities.SupportedBinnings
>>> csa.CameraSettings.Binning = bin[0]
```

Acquire

Overview

Once all of the camera settings have been made, acquiring the image is just using the command acquire, but we have to write it to a variable to access the resulting image, or to use any of the subcommands listed below.

```
>>> ai = csa.Acquire
```

Options that should be available under `ai` are: **Width**, **Height**, **PixelType**, **BitDepth**, **Metadata**, **AsSafeArray**, **AsVariant**, **SaveToFile**, and **PixelSize**.

....

Metadata

There are a lot of options underneath Metadata. these options include:

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	**ElectronCounted**	

Drift Corrected images will have these extra metadata options:

DriftCorrected	**DriftCorrectionConfidence**	**DriftCorrectionClipping**
DriftCorrectionVectorX	**DriftCorrectionVectorY**	

Electron Counted images will have one extra metadata option:

CountsToElectrons

Accessing metadata goes as follows:

```
>>> ai.Metadata
< POINTER >
>>> meta = ai.Metadata
>>> meta.ImageSize.Width # Needs Testing
>>> ai.Metadata.ImageSize.Width # Needs Testing
>>> ai.Metadata.ImageSize.Height # Needs Testing
>>> ai.Metadata.ExposureTime # Needs Testing
```

....

AsSafeArray

`csa.Acquire` is one of the few commands that doesn't output as a pointer! We can access the image array with the following,

```
>>> raw_array = ai.AsSafeArray
>>> array = np.asarray(raw_array)
```

The numpy array can then be used in the script to detect the position of the particle, or it can be safed as a .tiff file.

IsActive

Wait

4.1.3 CameraContinuousAcquisition

There wasn't much information about things that are listed under CameraContinuousAcquisition. My assumption is that all of the functions and syntax that apply to CameraSingleAcquisition also apply to this. I haven't used this function yet, so please make notes if you chose to!

4.2 Phaseplate

5 TEMScripting Package Overview

This seems to be the structure of the package, and the next sections will go over how to use the functions.

Instrument

- Projection
  - ImageShift

        * X, Y
    - Mode (diffraction or imaging)
    - Magnification (read only)
    - MagnificationIndex (read/write)

- Illumination
    - Shift
        * X, Y
- Stage
    - GoTo
    - GoToWithSpeed
    - MoveTo
    - Position
- Vacuum

6 TEMScripting Instrument

6.1 Instrument

First you have to download and install the package comtypes, then you can create the Instrument,

```
import comtypes
import comtypes.client as cc
tem = cc.CreateObject("TEMScripting.Istrument")
```

6.2 Projection

6.2.1 ImageShift

Getting

Running the following command will output a Pointer object – I'm not really sure how to access the contents of these Pointer objects. They are `<class 'comtypes.POINTER(Vector)'>` but none of the ways that comtypes outlined accessing pointers worked.

```
>>> tem.Projection.ImageShift
<POINTER(Vector) ptr=0x123478678 at abcdefg >
```

However, we can simply specify to the TEM scripting to give us just the x component of the shift, or just the y component of the shift. This works and prints out a float.

```
>>> tem.Projection.ImageShift.X
0.0
>>> tem.Projection.ImageShift.Y
0.0
```

These calls aren't case sensitive, so `tem.Projection.ImageShift.X` and `tem.projection.imageshift.x` will have the same output.

....

Setting

The inputs need to be in the same format as the output. Start by defining the current Image Shift to a variable, changing the internal values, and then setting the current image shift to the modified variable.

```
>>> imshift = tem.Projection.ImageShift
>>> imshift.X = 0.0001
>>> imshift.Y = 0.0001
>>> tem.Projection.ImageShift = imshift
```

If you want to set the new shift to a value relative to the current shift, use:

```
>>> imshift = tem.Projection.ImageShift
>>> imshift.X = imshift.X ± 0.00001
>>> imshift.Y = imshift.Y ± 0.00001
>>> tem.Projection.ImageShift = imshift
```

The input value for the image shiftcannot be larger than 1×10^{-4}. If the value is larger, there won't be an error generated, the command will just not have any effect on the instrument's image shift.

6.2.2 Projection Mode

Getting
Getting the current projection mode is very easy. You can either use the function to directly output the mode, or you can write it to a variable.

```
>>> tem.Projection.Mode
1
>>> mode = tem.Projection.Mode
```

Setting
The options for projection mode are either 1 or 2. Mode 1 corresponds to imaging and mode 2 is diffraction.

```
>>> tem.Projection.Mode = 1 # sets mode to Imaging
>>> tem.Projection.Mode = 2 # sets mode to Diffraction
```

Attempting to set the mode to a number other than 1 or 2 will result in an error.

6.2.3 Magnification

Getting
Getting the current magnification returns a float. You can either use the function to directly output the mode, or you can write it to a variable.

```
>>> tem.Projection.Magnification
0.0
>>> mag = tem.Projection.Magnification
```

6.2.4 MagnificationIndex

Indicies Table
Note: This is specifically for the Talos. Other microscopes might have different magnification index to magnification relationships.
Getting and setting the magnification of the microscope requires using a "Magnification Index" which is just an integer between 1 and 44. This integer corresponds to the magnification of the microscope. For the Talos, there are 44 different magnification indicies. The chart below is useful to see which magnification index corresponds to which magnification.

Index	TEM Display	Mag.
1	LM 25 x	25.0
2	LM 34 x	34.0
3	LM 46 x	46.0
4	LM 62 x	62.0
5	LM 84 x	84.0
6	LM 115 x	115.0
7	LM 155 x	155.0
8	LM 210 x	210.0
9	LM 280 x	280.0
10	LM 380 x	380.0
11	LM 510 x	510.0
12	LM 700 x	700.0
13	LM 940 x	940.0
14	LM 1300 x	1300.0
15	LM 1700 x	1700.0
16	LM 2300 x	2300.0
17	M 2050 x	2050.0
18	M 2600 x	2600.0
19	M 3300 x	3300.0
20	SA 4300 x	4300.0
21	SA 5500 x	5500.0
22	SA 7000 x	7000.0
23	SA 8600 x	8600.0
24	SA 11000 x	11000.0
25	SA 14000 x	14000.0
26	SA 17500 x	17500.0
27	SA 22500 x	22500.0
28	SA 28500 x	28500.0
29	SA 36000 x	36000.0
30	SA 46000 x	46000.0
31	SA 58000 x	58000.0
32	SA 74000 x	74000.0
33	SA 94000 x	94000.0
34	SA 120 kx	120000.0
35	SA 150 kx	150000.0
36	SA 190 kx	190000.0
37	SA 245 kx	245000.0
38	SA 310 kx	310000.0
39	SA 390 kx	390000.0
40	SA 500 kx	500000.0
41	SA 630 kx	630000.0
42	Mh 650 kx	650000.0
43	Mh 820 kx	820000.0
44	Mh 1.05 Mx	1050000.0

Getting
Getting the current magnification returns an integer. You can either use the function to directly output the mode, or you can write it to a variable.

```
>>> tem.Projection.MagnificationIndex
1
>>> mag = tem.Projection.MagnificationIndex
```

Setting
Setting the magnification to a new value requires the magnification index as given in the table above. Simply set the magnification index to the desired number and the microscope will switch magnifications.

```
>>> tem.Projection.MagnificationIndex = 1
```

6.3 Illumination

6.3.1 Shift

Getting
The 'Illumination Shift' is the beam shift. Much like the previous, accessing just `tem.Illumination.Shift` will give us a pointer. However, we can specify the x or y component of the shift and we will return a float.

```
>>> tem.Illumination.Shift.X
0.0
>>> tem.Illumination.Shift.Y
0.0
```

....

Setting

Function: `tem.Illumination.Shift`
Possible (useful) Attributes:

Attribute	Type	Description
.X	read/ write	Value of the **<u>u axis</u>** beam shift in meters. Accepts both decimals and scientific notation values lower than 5×10^{-4}.
.Y	read/ write	Value of the **<u>v axis</u>** beam shift in meters. Accepts both decimals and scientific notation values lower than 5×10^{-4}.

Important Info:
If the input value is larger than stated limit (above), the beam shift will not be excecuted and no errors will be raised.
The TEM uses a different set of unit vectors for the image shift ($\hat{u}$ and $\hat{v}$) that are defined by a linear combination of $\hat{x}$ and $\hat{y}$ (shown below). The $\hat{v}$ unit vector for the beam shift is the $-\hat{v}$ unit vector of the image shift.

$$\hat{v}_{\text{beam}} = -\hat{v}_{\text{image}}$$

$$\begin{pmatrix} u \\ v \end{pmatrix} = \frac{1}{6}\begin{pmatrix} 5.34 & 2.78 \\ -2.78 & 5.34 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix}$$

Example:
```
>>> bshift = tem.Illumination.Shift # bshift is the beam shift pointer
>>> bshift.X # will output the current u-axis beam shift
0.0
>>> bshift.X = 0.00001 (± bshift.X) # modifying the .X attribute
>>> tem.Illumination.Shift = bshift # setting the beam shift to new value.
```

6.4 Stage

6.4.1 Position

Getting

`tem.Stage.Position` is a read-only function. For setting the wanted stage position, we have to use the `GoTo/GoToWithSpeed/MoveTo` commands. Accessing just `tem.Stage.Position` will give us a pointer.

Accessing the individual compnents gives us a float of the current position. Options are **X**, **Y**, **Z**, **A**, and **B**.

```
>>> tem.Stage.Position.A
0.0
```

Alternatively, we could define the position pointer to a variable, `pos = tem.Stage.Position` and then access the individual components with `pos.(X,Y,Z,A,B)`.

Setting Position There are three different options for setting the position of the stage. In our script, for changing the alpha position from α_i to α_f we want it to move at a speed slower than default so we will use the **GoToWithSpeed** option.

The difference between the two **GoTo** options and the **MoveTo** option: The **GoTo** functions will Go To the new position directly from the old position, whereas the **MoveTo** option will go from the old position, zero out the stage, and then move to the new position.

Axis of Movement

For all of the movement methods, the second argument will be the axis of movement. You must pass through the integer value that defines the axis on which you wish the stage to move.

axis type	x	y	x-y	x-y-z	z	a	x-y-z-a	b	ax-y-z-a-b
integer value	1	2	3	7	4	8	15	16	31

Changing multiple variables within the stage position is possible as long as there is an axis defined for it. For example you can change the x and y axis position with the same command, but to change the x and z axis positions you would need two commands.
Update: There seems to be axis options from 1 to 31, and each "axis" interger value is x, y, z, a, b, or some combination of them. Further experimentation could be used to define all of the axes.

....

GoTo

If you want the stage to move but do not care about the movement speed, then this is the function to use. The positional argument must be in the same format as the positional output of `tem.Stage.Position`. If you wanted to change just the alpha tilt, you could use the following:

```
>>> pos = tem.Stage.Position
>>> pos.A = pos.A ± 0.1
>>> tem.Stage.GoTo(pos, 8)
```

Changing both the x and y position simutaneously would look like:

```
>>> pos = tem.Stage.Position
>>> pos.X = pos.X ± 0.1
>>> pos.Y = 0.1
>>> tem.Stage.GoTo(pos, 3)
```

The `tem.Stage.GoTo` function works for only the following axis and axis integer values:

axis type	x	y	x-y	z	a
integer value	1	2	3	4	8

....

MoveTo

The `tem.Stage.MoveTo` command works the same way as the `tem.Stage.GoTo` command. The only difference is how the stage is moving from point A to point B. Follow the instructions from the `tem.Stage.GoTo` section, and simply replace "`GoTo`" with "`MoveTo`"

....

GoToWithSpeed

If you want the stage to move at a certain speed, then this is the function to use. The positional argument must be in the same format as the positional output of `tem.Stage.Position`. The speed of 1.0 is the default speed; I haven't tried moving the stage *faster* than default, only slower. There seems to be no lower limit for how slow you'd like the stage to move.

```
>>> pos = tem.Stage.Position
>>> pos.A = pos.A ± 0.1
>>> tem.Stage.GoToWithSpeed(pos, 8, 0.1)
```

The `tem.Stage.GoToWithSpeed` function works for only the following axis and axis integer values:

axis type	x	y	z	a
integer value	1	2	4	8

Therefore to change both the x and y axis position at a certain speed you'd have to use two seperate `tem.Stage.GoToWithSpeed` commands.

```
>>> pos = tem.Stage.Position
>>> pos.X = pos.X ± 0.1
>>> tem.Stage.GoToWithSpeed(pos, 1, 0.1)
>>> pos.Y = 0.1
>>> tem.Stage.GoToWithSpeed(pos, 2, 0.1)
```

7 References

To construct this guide, I used the TEMScripting User Guide and the Advanced TEMScripting User Guide; both were provided by ThermoFisher. To create my TemPackage, I used Niermann's Temscript (github) as a reference for which functions were needed!