Introduction to Spinnaker SDK (Python version) with FLIR cooled cameras

This document is a short introduction to the use of Spinnaker SDK - Python - with a FLIR ATS cooled camera. It is not a training course!

Prerequisites:

- Minimum knowledge of the Python programming language, and of the GenICam protocol.
- Good knowledge of cooled cameras and how to operate them, for instance with RIR4Max/ResearchStudio.
- Of course, knowledge of infrared imaging, radiometry and thermography.

If you have questions and need to contact the Support Team, submit a ticket at https://support.flir.com

We are committed to answer, but we cannot develop the solution for you.

Date	Version	Author
03/2020	1 (En, Fr, It)	Raphaël Danjoux

Contents

Introduction

Cooled/photonic cameras from FLIR ATS

FLIR ATS cooled cameras are manufactured in Niceville, Florida. Current series are the A, the X and the RS.

All A and X models are natively GenICam compliant through GigE and CoaXPress (model dependent). Among the RS models, only the recent RS-85xx ones are GenICam compliant.

Spinnaker SDK

The Spinnaker SDK is FLIR's next generation GenICam3 API library. It is built around the GenICam standard, and it is compatible with USB3, 10GigE, and most GigE area scan cameras.

The Spinnaker SDK does not support Camlink nor CoaXPress.

Platforms

Supported platforms:

- Windows 7 (32- and 64-bit)
- Windows 10 (32- and 64-bit)
- Desktop Ubuntu 18.04 (64-bit)
- Desktop Ubuntu 16.04 (32-bit) / Ubuntu 18.04 (ARM64)
- Ubuntu (16.04 ARMHF & ARM64)
- MacOS (Mojave & High Sierra).

The current Spinnaker SDK version is 1.29 for Windows and MacOS. And it is 1.27 for Linux.

Supported languages

Spinnaker supports several programming languages:

- C
- C++
- C#
- Python

Python

PySpin is a wrapper for the Spinnaker library to allow you to code your application in Python.

GenICam Processes

Generalities

Every GenICam compliant camera has an XML description file. It details camera registers, their interdependencies, and all other information needed to access high-level features by means of low-level register read and write operations. The elements of a camera description file are represented as software objects called nodes. A nodes map is a list of nodes created dynamically at run time.

Regardless of the programming language used, the same node setup applies. Most nodes fall within seven types. They are:

Type	Description
Enumeration	Any feature that has a selection of text entries
EnumEntry	The individual entry within an enumeration feature
Command	Any feature that requires a command to execute
Float	Any feature that has a number entry that may include a decimal point
Boolean	Any feature that acts as an on or off switch
Integer	Any feature that has a number entry without a decimal point
String	Any feature that has a user-defined or static text entry

FLIR cameras GenICam processes

FLIR cooled cameras that are GenICam compliant share the same list of registers. Details can be found in the camera's GenICam ICD which is available for download on htpps://support.flir.com. Note that this document is generic, meaning that not all registers apply to all models. Trying to activate a mode or set a parameter that is not implemented won't work. You won't necessarily encounter any runtime error, but you won't get any result. A few examples:

- A670x models cannot be set to Superframing.
- A67xx models do not feature Preset Sequencing.
- Windowing is limited on A670x models.
- The TempLinear mode is not implemented on the X-series cameras.
- Only the X-series models have an SSD.
- Maximum framerate, window size and position are model dependent.
- Inputs/Outputs are model dependent.

  ……

Spinnaker BHP vs. BHP SDK. Differences, limitations.

The BHP SDK is another SDK for ATS cooled cameras. It has been available for years. It is therefore natural to wonder if Spinnaker SDK replaces BHP SDK. Well, the answer is no. The two products are totally different, do not offer the same capabilities and are not designed to respond to the same development needs. In short, Spinnaker SDK is not the product you need if you wish to develop an application that works whatever the cooled model and the streaming protocol. Also, Spinnaker SDK does not allow for image and movie acquisition in a FLIR radiometric format. For advanced applications, BHP SDK is needed.

	Spinnaker SDK	BHP SDK
Supported cameras	A62xx, A67xx, A85xx, X68xx, X69xx, X85xx, RS85xx	SC6100, SC6200, SC6700, SC6800, SC8200, SC8300 RS67xx, RS83xx A62xx, A67xx, A85xx, X68xx, X69xx, X85xx, RS85xx
Supported output	GigE	GigE, CoaXPress, CamLink
Supported grabbers	None	Any CoaXPress compliant grabber.

	Spinnaker SDK	BHP SDK
		Any CamLink grabber that maps the CamLink RS232 port to a virtual COM port.
FLIR File Support	None	SAF structure (Read and Write)
Supported development environments	C, C++, C#, Python	C/C++ Partial support for C# and .NET.
Supported OS	Windows Linux MacOS	Windows. Can be compiled for Linux. For embedded environments that cannot support high level compiled languages, additional documentation is available that provides details of how to manually construct the binary protocol messages.
Supported Camera Control	GenICam through GigE	GigE, GenICam through GigE, CamLink, RS232, GenICam through CoaXPress
Thermographic calibration	Enumerate, load and apply	Enumerate, load and apply
Factory correction	Enumerate, load and apply	Enumerate, load and apply
Non Uniformity Correction process	Yes	Yes
Save a state	Yes	Yes
Available units	Counts, temperature when TempLinear applies	Counts, radiance, temperature (°C, °F, K)
Advanced Graphic User Interface	No, to develop	Yes
Active Support by FLIR Staff through Custhelp	Yes	Yes

Miscellaneous

As written in the Generalities, the system of GenICam nodes is identical regardless of the programming language used. Therefore, any FLIR cameras that is natively GenICam compliant can be controlled by any programming language and any operating system: LabView, MATLAB, Halcon, Cognex, Matrox, etc. Note that performances may vary a lot from one to the other.

FLIR ATS Camera header

Definition

FLIR ATS cooled cameras stream out a header before the image itself. It contains important information that updated each frame.

The description of the header line can be given upon request. Contact the Support Team through your support account to get it.

Masking the header line

Once the full image array is captured by PySpin, the first line can easily be masked out in order to process exclusively radiometric data values.

Example assuming full format:

```
#PySpin array transferred as a numpy array
 image_data = image_result.GetNDArray()
#Masking the first line of the full array to consider only counts.
image_Counts= image_data[1:,0:]
```

Example of generic code. *GigE_Example_Cooled_Visu_Only.py*

This code is too long to be inserted here. It is available for free download on Custhelp.

When executed, you get a live video in counts, automatically scaled. A colorbar is placed on the right-hand side.

Enumerating factory calibrations

Factory calibration files are loaded and indexed in the camera memory. The index varies between zero and CalibrationQueryIndexMax. The procedure to get details on each of them is:

- Set CalibrationQueryIndex to the current index
- Read CalibrationQueryTag.

Example of code.

```
#Querying a calibration
CalibrationQueryIndexMax_node =
PySpin.CIntegerPtr(nodemap.GetNode('CalibrationQueryIndexMax'))
Index_Max = CalibrationQueryIndexMax_node.GetValue()
print('Index Max = ', Index_Max)

i=0
while i < Index_Max+1:
CalibIndex_node = PySpin.CIntegerPtr(nodemap.GetNode('CalibrationQueryIndex'))
CalibIndex_node.SetValue(i)
node_CalibrationTag = PySpin.CStringPtr(nodemap.GetNode('CalibrationQueryTag'))
CalibrationTag = node_CalibrationTag.GetValue()
print('Calib Index = ', i)
print('Calibration = ', CalibrationTag)
print('-----')
i = i + 1
```

Executing this code gives the following result:

```
Index Max = 17
Calib Index = 0
Calibration = 100mm, Empty, -20C - 55C
-----
Calib Index = 1
Calibration = 100mm, Empty, 10C - 90C
-----
Calib Index = 2
Calibration = 100mm, Empty, 150C - 350C
-----
Calib Index = 3
Calibration = 100mm, Empty, 35C - 150C
-----
Calib Index = 4
Calibration = 100mm, Empty, 80C - 200C
-----
Calib Index = 5
Calibration = 25mm, Empty, -20C - 55C
-----
Calib Index = 6
Calibration = 25mm, Empty, 10C - 90C
-----
Calib Index = 7
Calibration = 25mm, Empty, 150C - 350C
-----
Calib Index = 8
Calibration = 25mm, Empty, 35C - 150C
-----
Calib Index = 9
Calibration = 25mm, Empty, 80C - 200C
-----
Calib Index = 10
Calibration = 50mm, Empty, -20C - 55C
-----
Calib Index = 11
Calibration = 50mm, Empty, 10C - 90C
```

```
-----
Calib Index = 12
Calibration = 50mm, Empty, 150C - 350C
-----
Calib Index = 13
Calibration = 50mm, Empty, 35C - 150C
-----
Calib Index = 14
Calibration = 50mm, Empty, 80C - 200C
-----
Calib Index = 15
Calibration = 50mm, ND2, 250C - 600C
-----
Calib Index = 16
Calibration = 50mm, ND2, 500C - 1200C
-----
Calib Index = 17
Calibration = 50mm, ND2, 850C - 2000C
-----
```

Loading a factory calibration in a preset

Once you have obtained details on the calibration files and know which one you want to load in which preset, follow these two steps:

- Set PSxCalibrationLoadTag to the tag of the desired calibration
- Execute PSxCalibrationLoad

The code below loads the calibration file "`50mm, Empty, 35C - 150C`", having index 13 in the list of the previous section, into Preset0.

```
#Loading a calibration
Calibration_to_load ='50mm, Empty, 35C - 150C'
CalibrationTag_node = PySpin.CStringPtr(nodemap.GetNode('PS0CalibrationLoadTag'))
CalibrationTag_node.SetValue(Calibration_to_load)
CalibrationActive_node = PySpin.CCommandPtr(nodemap.GetNode('PS0CalibrationLoad'))
CalibrationActive_node.Execute()
```

Note that loading a factory calibration also loads the corresponding factory correction. The camera may also undergo a shutter maneuver.

Extracting the maximum and minimum count values for the current factory calibration

We assume here that a factory calibration is loaded.

```
#Retrieving maximumm and minimum counts values for current calibration
maximum_counts_node =
PySpin.CIntegerPtr(nodemap.GetNode('CalibrationQueryMaxCounts'))
maximum = maximum_counts_node.GetValue();
minimum_counts_node =
PySpin.CIntegerPtr(nodemap.GetNode('CalibrationQueryMinCounts'))
minimum = minimum_counts_node.GetValue();
```

Extracting the Clock Value

The clock value is the base time step considered for the camera. Most time-based parameters like integration time, framerate and superframe rates, etc. can be expressed as a multiple of clock values.

```
#Retrieving clock value
#Preset0
PS0FrameWidthActual_node =
PySpin.CIntegerPtr(nodemap.GetNode('PS0FrameWidthActual'))
PS0FrameWidthActual = PS0FrameWidthActual_node.GetValue()
PS0IntegrationTime_node = PySpin.CFloatPtr(nodemap.GetNode('PS0IntegrationTime'))
PS0IntegrationTime = PS0IntegrationTime_node.GetValue()
#Clock Value calculation in ms
Clock_Value = PS0IntegrationTime / PS0FrameWidthActual
```

Activating Single preset

There are several solutions for Single preset.

Either there is a factory calibration that is loaded in the desired preset. Or there is no factory calibration, and the integration time is manually set. In the first case, TempLinear mode can be activated on top. In the second case, streaming is only in counts.

```
#Activating SinglePreset mode, then activating a given preset.
#The example below is given for PS0. Change to PS1 or PS2 or PS3 if needed
node_Preset_Sequencing_Mode =
PySpin.CEnumerationPtr(nodemap.GetNode('PresetSequencingMode'))
node_SinglePreset =
PySpin.CEnumEntryPtr(node_Preset_Sequencing_Mode.GetEntryByName('SinglePreset'))
node_SinglePreset_on = node_SinglePreset.GetValue()
node_Preset_Sequencing_Mode.SetIntValue(node_SinglePreset_on)#Should be 0

#Activating a particular preset.
node_GigEPreset = PySpin.CEnumerationPtr(nodemap.GetNode('GigEPreset'))
node_GigEPreset_PS0 = PySpin.CEnumEntryPtr(node_GigEPreset.GetEntryByName('PS0'))
GigE_PS0 = node_GigEPreset_PS0.GetValue()
node_GigEPreset.SetIntValue(GigE_PS0)
```

Getting/Setting the frame rate of a preset

Getting the frame rate

The following example is given for Preset 0. Change the value to use the other presets.

```
#Getting the actual framerate for Preset 0
PS0FrameRateActual_node = PySpin.CFloatPtr(nodemap.GetNode('PS0FrameRateActual'))
PS0FrameRateActual = PS0FrameRateActual_node.GetValue()
```

Alternately, when in Single preset mode, one can also use the node PS0FrameRate instead of PS0FrameRateActual.

```
#Getting the actual framerate for Preset 0
PS0FrameRate_node = PySpin.CFloatPtr(nodemap.GetNode('PS0FrameRate'))
PS0FrameRate = PS0FrameRate_node.GetValue()
```

Setting the framerate

Framerate is set to 30 fps on Preset 0.

```
#Setting the framerate in Preset 0
PS0FrameRate_node = PySpin.CFloatPtr(nodemap.GetNode('PS0FrameRate'))
PS0FrameRate_node.SetValue(30.0)
```

Activating Superframing mode

The examples below show some solutions, but not all of them. Possible values of registers are found in the GenICamICD document.

We assume that the proper factory calibrations are loaded in the desired presets.

```
#Activating PS0 and PS1 for Superframing, PS2 and PS3 are not activated
node_PS0_For_Superframing =
PySpin.CBooleanPtr(nodemap.GetNode('PS0SubframeEnabled'))
node_PS1_For_Superframing =
PySpin.CBooleanPtr(nodemap.GetNode('PS1SubframeEnabled'))
node_PS2_For_Superframing =
PySpin.CBooleanPtr(nodemap.GetNode('PS2SubframeEnabled'))
node_PS3_For_Superframing =
PySpin.CBooleanPtr(nodemap.GetNode('PS3SubframeEnabled'))
node_PS0_For_Superframing.SetValue(True)
node_PS1_For_Superframing.SetValue(True)
node_PS2_For_Superframing.SetValue(False)
node_PS3_For_Superframing.SetValue(False)

#Defining the SuperFrame period (in clocks), Example for 40 ms (or 25 Hz)
#For X-series Base Clock = 1.01759 E-6 ms
Superframe_Width = 444000
SuperframeWidth_node = PySpin.CIntegerPtr(nodemap.GetNode('SuperframeWidth'))
SuperframeWidth_node.SetValue(Superframe_Width)

#Activating Superframing mode, and selecting which preset(s) to output
#The example below is given for All. Change to PS0, PS1, PS2 or PS3 and adapt
naming if needed.
node_Preset_Sequencing_Mode =
PySpin.CEnumerationPtr(nodemap.GetNode('PresetSequencingMode'))
node_Superframing_Mode =
PySpin.CEnumEntryPtr(node_Preset_Sequencing_Mode.GetEntryByName('Superframing'))
node_Superframing_on = node_Superframing_Mode.GetValue()
node_Preset_Sequencing_Mode.SetIntValue(node_Superframing_on)#Should be 2

#Outputting all active presets
node_GigEPreset = PySpin.CEnumerationPtr(nodemap.GetNode('GigEPreset'))
node_GigEPreset_All = PySpin.CEnumEntryPtr(node_GigEPreset.GetEntryByName('All'))
GigE_All = node_GigEPreset_All.GetValue()
node_GigEPreset.SetIntValue(GigE_All)
```

Preparing Preset Sequencing

Preparing Preset Sequencing means for each preset:

- Loading a calibration file (or the integration time in case there is no need for a calibration)
- Defining the framerate
- Defining the dwell count

The maximum framerate depends on the integration time. Therefore, the operator must verify first what its value is using the node PSxFrameRateMax.

Methods for loading a calibration file and defining the framerate are described in previous sections. Here are the details for the dwell count. Example is given for Preset 0; change numbering for the other presets. Note that inactivating a preset is obtained by setting a dwell count to zero.

```
#Getting the dwell count in Preset 0
PS0DwellCount_node = PySpin.CIntegerPtr(nodemap.GetNode('PS0DwellCount'))
PS0DwellCount = PS0DwellCount_node.GetValue()

#Setting the dwell count in Preset 0
PS0DwellCount_node = PySpin.CIntegerPtr(nodemap.GetNode('PS0DwellCount'))
PS0DwellCount_node.SetValue(30)#Value is 30
```

Activating Preset Sequencing mode

We assume here that the proper factory calibrations are loaded in the desired presets, and the sequence is already prepared.

```
#Activating PresetSequencing mode
node_Preset_Sequencing_Mode =
PySpin.CEnumerationPtr(nodemap.GetNode('PresetSequencingMode'))
node_PresetSequencing =
PySpin.CEnumEntryPtr(node_Preset_Sequencing_Mode.GetEntryByName('PresetSequencing')
)
node_PresetSequencing_on = node_PresetSequencing.GetValue()
node_Preset_Sequencing_Mode.SetIntValue(node_PresetSequencing_on)#Should be 1
```

Getting/Setting the object parameters

Setting the object parameters is important when the temperature linear mode is implemented. The conversion from counts to temperature is then performed in the camera head.

There are 9 object parameters:

Node	Description
ObjectEmissivity	Emissivity of the target (varies between 0 and 1)
ObjectDistance	Distance to target (in meters)
ReflectedTemperature	Reflected apparent temperature (in kelvins)
AtmosphericTemperature	Average atmospheric temperature along the optical path (in kelvins)
ExtOpticsTemperature	External optics temperature (in kelvins)
ExtOpticsTransmission	External optics transmission coefficient (varies between 0 and 1)
EstimatedTransmission	Estimated atmospheric transmission coefficient (varies between 0 and 1). If not possible to set, then value 1.0 is used.
RelativeHumidity	Average relative humidity along the optical path (varies between 0 and 1)

When the estimated transmission is set to 0, the value that is effectively considered is calculated from the object parameters with a mathematic model. It is replaced by a fixed value if one wants to override the estimation.

The example is given for the emissivity.

```
#Getting and setting the current emissivity
ObjectEmissivity_node = PySpin.CFloatPtr(nodemap.GetNode('ObjectEmissivity'))
ObjectEmissivity = ObjectEmissivity_node.GetValue()
ObjectEmissivity_node.SetValue(0.75)
```

Setting the Temperature Linear mode

When the temperature linear mode is activated, the digital output is proportional to the temperature in kelvins.

*Temperature in kelvins = count_value * res*

Res (resolution) is 0.1 for TempLinear100mK and 0.01 for TempLinear10mK. Object parameters are those contained in the camera memory, and they are applied at once to all pixels.

The output is set to 16 bits and the obtained ranges are therefore:

- 0 to 655.36 K, or 0 to 382.21 °C, with TempLinear10mK.
- 0 to 6553.6 K, or 0 to 6280.45 °C, with TempLinear100mK.

```
#For Niceville cooled cameras in Temp Linear Mode (only A-series as not implemented
on X-series)
node_temp_linear = PySpin.CEnumerationPtr(nodemap.GetNode('IRFormat'))
node_temp_linear_low =
PySpin.CEnumEntryPtr(node_temp_linear.GetEntryByName('TemperatureLinear100mK'))
linear_low = node_temp_linear_low.GetValue()
node_temp_linear.SetIntValue(linear_low)
```

Retrieving the calibration coefficients, and obtaining radiance and temperature images for a blackbody

We assume that a given factory calibration has been selected before and is active. Also, the temperature linear mode is either not implemented or the operator does not want to use it.

```
# Retrieving Calibration constants

        CalibrationQueryCoeff0_node =
PySpin.CFloatPtr(nodemap.GetNode('CalibrationQueryCoeff0'))
        CalibrationQueryCoeff0 = CalibrationQueryCoeff0_node.GetValue()

        CalibrationQueryCoeff1_node =
PySpin.CFloatPtr(nodemap.GetNode('CalibrationQueryCoeff1'))
        CalibrationQueryCoeff1 = CalibrationQueryCoeff1_node.GetValue()

        CalibrationQueryCoeff2_node =
PySpin.CFloatPtr(nodemap.GetNode('CalibrationQueryCoeff2'))
        CalibrationQueryCoeff2 = CalibrationQueryCoeff2_node.GetValue()

        CalibrationQueryR_node =
PySpin.CFloatPtr(nodemap.GetNode('CalibrationQueryR'))
        R = CalibrationQueryR_node.GetValue()

        CalibrationQueryB_node =
PySpin.CFloatPtr(nodemap.GetNode('CalibrationQueryB'))
        B = CalibrationQueryB_node.GetValue()

        CalibrationQueryF_node =
PySpin.CFloatPtr(nodemap.GetNode('CalibrationQueryF'))
        F = CalibrationQueryF_node.GetValue()
```

For a blackbody

CalibrationQueryCoeff0, CalibrationQueryCoeff1 and CalibrationQueryCoeff2 are used to calculate radiance (in W/cm²/sr) from counts (14-bit values). The interval of calculation is defined by CalibrationQueryMinCounts and CalibrationQueryMaxCounts. Final values can also be retrieved using CalibrationQueryMinRadiance and CalibrationQueryMaxRadiance.

$$Radiance = Coeff0 + Coeff1 \times counts\_value + Coeff2 \times (counts\_value)^2$$

If we get an array of count values called *image_Counts*, then the corresponding array in radiance is obtained with the formula:

```
image_Radiance = (CalibrationQueryCoeff0 + CalibrationQueryCoeff1 *
image_Counts + CalibrationQueryCoeff2 * image_Counts *image_Counts)
```

When a dual factory calibration (radiance and temperature) is achieved, the assumption is that the system responds linearly in radiance. Therefore, Coeff2 is very small.

CalibrationQueryR, CalibrationQueryB and CalibrationQueryF are used to calculate temperature in kelvins from radiance. The interval of calculation follows the same rule as for radiance. However, final values (in kelvins) can also be retrieved using CalibrationQueryMinTemp and CalibrationQueryMaxTemp.

The calibration in temperature obeys datapoints fit with a modified monochromatic Planck's formula, elaborated in 1982 by a Japanese team lead by Fumihiro Sakuma. It is now referred to as the RBF equation.

Temperature = B / Ln [(R/Radiance) +F]

From the radiance array previously defined, the temperature array in degrees Celsius is obtained with the formula:

```
image_Temp = (B / (numpy.log((R/image_Radiance) + F))) - 273.15
```