CXP-12 Interface Card 1C

AcquisitionApplets User Documentation for

Acq_SingleCXP12Line

Functional Description
For Framegrabber SDK Usage

Document Number: AW001810
Part Number: 000 (English)
Document Version: 02
Release Date: 22 December 2023
Applet Version 1.1.1.0

Contacting Basler Support Worldwide

Europe, Middle East, Africa
Tel. +49 4102 463 515
support.europe@baslerweb.com

The Americas
Tel. +1 610 280 0171
support.usa@baslerweb.com

Asia-Pacific
Tel. +65 6367 1355
support.asia@baslerweb.com

Singapore
Tel. +65 6367 1355
support.asia@baslerweb.com

Taiwan
Tel. +886 3 558 3955
support.asia@baslerweb.com

China
Tel. +86 10 6295 2828
support.asia@baslerweb.com

Korea
Tel. +82 31 714 3114
support.asia@baslerweb.com

Japan
Tel. +81 3 6672 2333
support.asia@baslerweb.com

https://www.baslerweb.com/en/sales-support/support-contact

Supplemental Information

Acquisition Card Documentation:
https://docs.baslerweb.com/acquisition-cards
Frame Grabber Documentation:
https://docs.baslerweb.com/frame-grabbers
Framegrabber SDK Documentation:
https://docs.baslerweb.com/frame-grabbers/framegrabber-sdk-overview.html

Table of Contents

Chapter 1. Introduction

This document provides you with detailed information on applet "Acq_SingleCXP12Line" for CXP-12 Interface Card 1C .

This document will outline the features and benefits of this applet. Furthermore, the output formats and the software interface is explained. The main part of this document includes the detailed description of all applet modules and their parameters which make the applet adaptable for numerous applications.

1.1. Features of Applet Acq_SingleCXP12Line

"Acq_SingleCXP12Line" is an applet for one camera (single-camera applet). You can configure the CoaXPress camera interface for CoaXPress cameras version 1.1.1 and 2.0, transferring grayscale (monochrome) or color pixels. Allowed pixel formats are Gray (Mono8, Mono10, Mono12, Mono14, Mono16), Color (RGB8, RGB10, RGB12, RGB14, RGB16), and YCbCr422_8. You can use a camera with a single CoaXPress link with this applet. The maximum link speed is CXP-12. A multi-functional line trigger is included in the applet. This allows you to control the camera or external devices using interface card generated, external or software generated trigger pulses. Line scan cameras up to a width of 32768 pixels can be processed. The trigger system will generate images of a maximum height of 8388607 pixels. The applet is processing data at a bit depth of 16 bits. An image selector at the camera port facilitates the selection of one image out of a parameterizable sequence of images. This enables the distribution of the images to multiple interface card and PCs. For reverse operation, you can mirror the image in x-direction and y-direction before cutting the ROI. Acquired images are buffered in interface card memory. You can select a region of interest (ROI) for further processing. The stepsize of the ROI width is 4 pixel. The ROI stepsize for the image height is 1 line. A color converter automatically converts the input pixel formats to the output formats. In this applet conversions from monochrome, RGB to monochrome and RGB can be performed.

Processed image data are output by the applet via a high speed DMA channel. You can select the pixel format of the output. The pixel format can either be 8 bit, 10 bit packed, 12 bit packed, 14 bit packed, or 16 bits per pixel (or per pixel component if you work with a color format).

You can easily include the applet into your own applications using the Basler Framegrabber SDK.

Table 1.1. Feature Summary of Acq_SingleCXP12Line

Feature	Applet Property
Applet Name	 Acq_SingleCXP12Line
Type of Applet	AcquisitionApplets
Board	CXP-12 Interface Card 1C
No. of Cameras	1
Camera Type	CoaXPress, link aggregation max. 1, maximum speed CXP-12, Version 1.1.1 and 2.0
Sensor Type	Line Scan
Camera Format	Monochrome or RGB
Pixel Format	Gray (Mono8, Mono10, Mono12, Mono14, Mono16), Color (RGB8, RGB10, RGB12, RGB14, RGB16), and YCbCr422_8.
Processing Bit Depth	16 Bit per color component
Sensor Correction / Tap Sorting	no
Maximum Images Dimensions	32768 * 8388607
ROI Stepsize	x: 4, y: 1
Tap Geometry Sorting	1X-1Y only
Mirroring	Yes, horizontal and vertical (set the parameter *FG_VANTAGEPOINT*)
Image Selector	Yes
Noise Filter	No
Shading Correction	No
Dead Pixel Interpolation	No
Bayer Filter	No
Color White Balancing	Yes
Color Converter	yes, Mono, RGB to Mono or RGB
Lookup Table	No
DMA	Full Speed
DMA Image Output Format	All grayscale and color formats. See description above.
Event Generation	yes
Overflow Control	yes

1.1.1. Parameterization Order

We recommend to configure the functional blocks which are responsible for sensor setup/correction first. This will be the camera settings, shading correction, and dead pixel interpolation (if available). Afterwards, you can configure other image enhancement functional blocks such as white balancing, noise filter, and lookup table. By default, all presets are configured for receiving images directly.

1.2. Bandwidth

The maximum bandwidths of applet Acq_SingleCXP12Line are listed in the following table.

Table 1.2. Bandwidth of Acq_SingleCXP12Line

Description	Bandwidth
Max. CXP Speed	CXP-12
Peak Bandwidth per Camera	1200 MPixel/s
Mean Bandwidth per Camera	1200 MPixel/s
DMA Bandwidth	3260 MByte/s (depends on PC mainboard)

The peak bandwidth defines the maximum allowed bandwidth for each camera at the camera interface. If the camera's peak bandwidth is higher than the mean bandwidth, the interface card on-board buffer will fill up as the data can be buffered, but not be processed at that speed.

The mean bandwidth per camera describes the maximally allowed mean bandwidth for each camera at the camera interface. It is the product of the framerate and the image pixels. For example, with 1-megapixel images at a framerate of 100 frames per second, the mean bandwidth will be 100 MPixel/s. In case of 8bit per pixel as output format, this would be equal to 100 MB per second.

The required output bandwidth of an applet can differ from the input bandwidth. A region of interest (ROI) and the output format can change the required output bandwidth and the maximum mean bandwidth.

Regard the relation between MPixel/s and MByte/s: The MByte/s depend on the applet and its parameterization concerning the pixel format. It is possible to acquire more than 8 bit per pixel or to convert from one bit depth to another. 1 MByte is 1,000,000 Byte.

Bandwidth Varies

The exact maximum DMA bandwidth depends on the used PC system and its chipset. The camera bandwidth depends on the image size and the selected frame rate. The given values of 3260 MByte/s for the possible DMA bandwidth might be lower due to the chipset and its configuration. Additionally, some PCIe slots do not support the required number of lanes to transfer the requested or expected bandwidth. In these cases, have a look at the mainboard specification. A behaviour like multiplexing between several PCIe slots can be seen in rare cases. Some mainboard manufacturers provide a BIOS feature where you can select the PCIe payload size: Always try to set this to its maximum value or simply to automatic. This can help in specific cases.

1.3. Requirements

In the following, the requirements on software, hardware and interface card license are listed.

1.3.1. Software Requirements

To run this applet, a Basler Framegrabber SDK installation is required. Ensure you use the applet with compatible versions only. You should also take care to use the board firmware and drivers included in the Basler Framegrabber SDK.

For integration in 3rd party applications, check Chapter 2, '*Software Interface*'.

1.3.2. Hardware Requirements

To run applet "Acq_SingleCXP12Line", a Basler CXP-12 Interface Card 1C is required.

For PC system requirements, check the interface card hardware documentation. The applet itself does not require any additional PC system requirements.

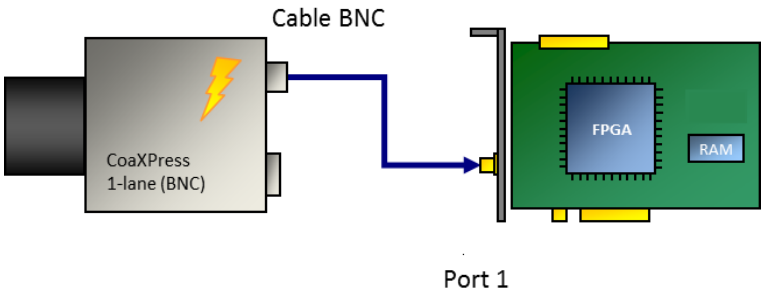
1.3.3. License

This applet is of type AcquisitionApplets. For applets of this type, no license is required. All compatible interface cards can run the applet using the Basler Framegrabber SDK.

1.4. Camera Interface

Applet "Acq_SingleCXP12Line" supports 1 CXP camera. The interface card has 1 connector. Use a single CoaXPress cable to connect the camera with the interface card. The maximum link aggregation of this applet is one.

Figure 1.1. Camera Interface and Camera Cable Setup



1.5. Frame ID

For CoaXPress linescan cameras the CXP Source Tag is not used as it is constand throughout the acquisition. Instead an internal counter is used to represent frame IDs. This applet will output each frame to the host PC attached with this frame ID. Moreover, overflow events will also include this frame ID. By this, the exact mapping of a given frame in the host PC to the frame the frame grabber's image trigger is possible.

Check chapter Chapter 11, '*Overflow*' for more information about overflow conditions and the overflow event data structure including the frame ID.

Check chapter Section 1.7, 'DMA Image Tag' to get information on how to optain the frame ID along with a given image in the host PC application.

1.6. Image Transfer to PC Memory

The image transfer between interface card and PC is performed via DMA transfers. In this applet, only one DMA channel exists for transferring image data. The DMA channel has index 0. The applet output format can be set via the parameters of the output format module. See Chapter 15, '*Output Format*'. All outputs are little-endian coded.

1.7. DMA Image Tag

The applet generates a DMA image tag (**FG_IMAGE_TAG**) for every correct transmitted frame. The **FG_IMAGE_TAG** has the folowing structure:

Table 1.3. Structure of FG_IMAGE_TAG

Bits	Description
0..15	frameID transmitted by the Camera
16..29	reserved = 0
30	invalid image flag (the image was cut of due to overflow in the framegrabber)
31	Last Image of Multi Buffer Sequence (always 1 in case of an area applet)

You may check for lost or corrupted frames using the overflow module described in Chapter 11, '*Overflow*'.

Chapter 2. Software Interface

The software interface of this applet is fully compatible to the Basler Framegrabber SDK. Please read the Basler Framegrabber API manual of the Basler Framegrabber SDK to understand how to include the frame grabbers and their applets into own applications. https://docs.baslerweb.com/frame-grabbers/framegrabber-sdk-overview.html

The Basler Framegrabber SDK includes functional SDK examples which use the features of the Framegrabber SDK. Most of these examples can be used with this AcquisitionApplets. These examples are very useful to learn on how to acquire images, set parameters and use events.

This document is focused on the explanation of the functionality and parameterization of the applet. The next chapters will list all parameters of this applet. Keep in mind that for multi-camera applets, parameters can be set for all cameras individually. The sample source codes parameterize the processing components of the first camera. The index in the source code examples has to be changed for the other cameras.

Amongst others, parameters of the applet are set and read using functions

- int Fg_setParameter(Fg_Struct *Fg, const int Parameter, const void *Value, const unsigned int index)
- int Fg_setParameterWithType(Fg_Struct *Fg, const int Parameter, const void *Value, const unsigned int index, const enum FgParamTypes type)
- int Fg_getParameter(Fg_Struct *Fg, int Parameter, void *Value, const unsigned int index)
- int Fg_getParameterWithType(Fg_Struct *Fg, const int Parameter, void *Value, const unsigned int index, const enum FgParamTypes type)

The index is used to address a DMA channel, a camera index or a processing logic index. It is important to understand the relations between cameras, processes, parameters and DMA channels. This AcquisitionApplets is a single camera applet and is using only one DMA channel. All parameterizations are made using index 0 only.

For applets having multiple DMA channels for each camera, the relation between the indices is more complex. Please check the respective documentation of these applets for more details.

Chapter 3. CoaXPress

This applet can be used with one line scan camera. To receive correct image data from your camera, it is crucial that the camera output format matches the selected interface card input format. The following parameters configure the interface card's camera interface to match with the individual camera pixel format. Most cameras support different operation modes. Consult the manual of your camera to obtain the necessary information how to configure the camera to the desired pixel format.

Ensure that the lines transferred by the camera do not exceed the maximum allowed line length for this applet (32768).

With the following parameters you can define the way trigger packets are sent from the interface card to the camera on the CXP link.

3.1. FG_PIXELFORMAT

This parameter specifies the data format of the connected camera.

The formats defined in the following list can be selected. Choose the pixel format which best matches with your camera.

In this applet, the processing data bit depth is 16 bit. The camera interface automatically performs a conversion to the 16 bit format using bit shifting independently from the selected camera format. If the camera bit depth is greater than the processing bit depth, bits will be right shifted to meet the internal bit depth. If the camera bit depth is less than the processing bit depth, bits will be left shifted to meet the internal bit depth. In this case, the lower bits are fixed to zero.

Table 3.1. Parameter properties of FG_PIXELFORMAT

Property	Value
Name	**FG_PIXELFORMAT**
Display Name	**Pixel Format**
Type	**Enumeration**
Access policy	**Read/Write**
Storage policy	**Persistent**
Allowed values	**Mono8** Mono 8 **Mono10** Mono 10p **Mono12** Mono 12p **Mono14** Mono 14p **Mono16** Mono 16p **RGB8** RGB 8 **RGB10** RGB 10p **RGB12** RGB 12p **RGB14** RGB 14p **RGB16** RGB 16 **YUV422_8** YCbCr422_8
Default value	**Mono8**

Example 3.1. Usage of FG_PIXELFORMAT

```
int result = 0;
int value = Mono8;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_PIXELFORMAT, &value, 0, type)) < 0) {
```

```
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_PIXELFORMAT, &value, 0, type)) < 0) {
    /* error handling */
}
```

3.2. FG_SYSTEMMONITOR_USED_CXP_CONNECTIONS

The currently used number of CXP ports used in this process.

Table 3.2. Parameter properties of FG_SYSTEMMONITOR_USED_CXP_CONNECTIONS

Property	Value
Name	**FG_SYSTEMMONITOR_USED_CXP_CONNECTIONS**
Display Name	**System Monitor Used Cxp Connections**
Type	**Unsigned Integer**
Access policy	**Read-Only**
Storage policy	**Persistent**
Allowed values	**Minimum 1** **Maximum 4** **Stepsize 1**

Example 3.2. Usage of FG_SYSTEMMONITOR_USED_CXP_CONNECTIONS

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_USED_CXP_CONNECTIONS, &value, 0, type)) < 0) {
    /* error handling */
}
```

3.3. FG_PACKET_TAG_ERROR_COUNT

The parameter reflects the current status of the camera operator. The parameter signalizes CXP stream packet loss detection.

Table 3.3. Parameter properties of FG_PACKET_TAG_ERROR_COUNT

Property	Value
Name	**FG_PACKET_TAG_ERROR_COUNT**
Display Name	**Packet Tag Error Count**
Type	**Unsigned Integer**
Access policy	**Read-Only**
Storage policy	**Persistent**
Allowed values	**Minimum 0** **Maximum 4095** **Stepsize 1**

Example 3.3. Usage of FG_PACKET_TAG_ERROR_COUNT

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_getParameterWithType(fg, FG_PACKET_TAG_ERROR_COUNT, &value, 0, type)) < 0) {
```

```
    /* error handling */
}
```

3.4. FG_CORRECTED_ERROR_COUNT

The parameter reflects the current status of the camera operator. The parameter signalizes single byte error correction in CXP stream packets.

Table 3.4. Parameter properties of FG_CORRECTED_ERROR_COUNT

Property	Value
Name	**FG_CORRECTED_ERROR_COUNT**
Display Name	**Corrected Error Count**
Type	**Unsigned Integer**
Access policy	**Read-Only**
Storage policy	**Persistent**
Allowed values	**Minimum 0** **Maximum 4095** **Stepsize 1**

Example 3.4. Usage of FG_CORRECTED_ERROR_COUNT

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_getParameterWithType(fg, FG_CORRECTED_ERROR_COUNT, &value, 0, type)) < 0) {
    /* error handling */
}
```

3.5. FG_UNCORRECTED_ERROR_COUNT

The parameter reflects the current status of the camera operator. The parameter signalizes multiple byte error detection in CXP stream packets.

Table 3.5. Parameter properties of FG_UNCORRECTED_ERROR_COUNT

Property	Value
Name	**FG_UNCORRECTED_ERROR_COUNT**
Display Name	**Uncorrected Error Count**
Type	**Unsigned Integer**
Access policy	**Read-Only**
Storage policy	**Persistent**
Allowed values	**Minimum 0** **Maximum 4095** **Stepsize 1**

Example 3.5. Usage of FG_UNCORRECTED_ERROR_COUNT

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_getParameterWithType(fg, FG_UNCORRECTED_ERROR_COUNT, &value, 0, type)) < 0) {
    /* error handling */
}
```

3.6. FG_SYSTEMMONITOR_PACKETBUFFER_OVERFLOW_COUNT

Represents the number of overflows, if an overflow occurred due to not correctly aligned package order.

Table 3.6. Parameter properties of FG_SYSTEMMONITOR_PACKETBUFFER_OVERFLOW_COUNT

Property	Value
Name	**FG_SYSTEMMONITOR_PACKETBUFFER_OVERFLOW_COUNT**
Display Name	**System Monitor Packet Buffer Overflow Count**
Type	**Unsigned Integer**
Access policy	**Read-Only**
Storage policy	**Persistent**
Allowed values	**Minimum 0** **Maximum 4095** **Stepsize 1**

Example 3.6. Usage of FG_SYSTEMMONITOR_PACKETBUFFER_OVERFLOW_COUNT

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_PACKETBUFFER_OVERFLOW_COUNT, &value, 0, type)) < 0) {
    /* error handling */
}
```

3.7. FG_SYSTEMMONITOR_PACKETBUFFER_OVERFLOW_SOURCE

This parameter represents the port, which has overflows due to not correctly aligned package order.

Table 3.7. Parameter properties of FG_SYSTEMMONITOR_PACKETBUFFER_OVERFLOW_SOURCE

Property	Value
Name	**FG_SYSTEMMONITOR_PACKETBUFFER_OVERFLOW_SOURCE**
Display Name	**System Monitor Packet Buffer Overflow Source**
Type	**Unsigned Integer**
Access policy	**Read-Only**
Storage policy	**Persistent**
Allowed values	**Minimum 0** **Maximum 15** **Stepsize 1**

Example 3.7. Usage of FG_SYSTEMMONITOR_PACKETBUFFER_OVERFLOW_SOURCE

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_PACKETBUFFER_OVERFLOW_SOURCE, &value, 0, type)) < 0) {
    /* error handling */
}
```

3.8. FG_SYSTEMMONITOR_CXP_IMAGE_LINE_MODE

This parameter informs on the current transfer mode, used by the camera. The transfer can be an areascan (= 0) or linescan (= 1) image.

Table 3.8. Parameter properties of FG_SYSTEMMONITOR_CXP_IMAGE_LINE_MODE

Property	Value
Name	**FG_SYSTEMMONITOR_CXP_IMAGE_LINE_MODE**
Display Name	**System Monitor Cxp Image Line Mode**
Type	**Unsigned Integer**
Access policy	**Read-Only**
Storage policy	**Persistent**
Allowed values	**Minimum 0** **Maximum 1** **Stepsize 1**

Example 3.8. Usage of FG_SYSTEMMONITOR_CXP_IMAGE_LINE_MODE

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_CXP_IMAGE_LINE_MODE, &value, 0, type)) < 0) {
    /* error handling */
}
```

Chapter 4. Camera

This applet Acq_SingleCXP12Line for the CXP-12 Interface Card 1C acquires the sensor data of a line scan camera. When this is performed some sensor dimension depending information can be used to register an event based callback function.

4.1. Events

In programming or runtime environments, a callback function is a piece of executable code that is passed as an argument, which is expected to call back (execute) exactly that time an event is triggered. This applet can generate some software callback events based on applet-events as explained in the following section. These events are not related to a special camera functionality. Other event sources are described in additional sections of this document.

The Basler Framegrabber SDK enables an application to get these event notifications about certain state changes at the data flow from camera to RAM and the image and trigger processing as well. Please consult the Basler Framegrabber SDK documentation for more details concerning the implementation of this functionality.

4.1.1. FG_START_OF_FRAME_CAM_PORT_0

4.1.2. FG_END_OF_FRAME_CAM_PORT_0

4.1.3. FG_START_OF_LINE_CAM_PORT_0

This event is generated when the first pixel of camera line arrives at the framegrabber. Keep in mind that a high linerate can cause a critical high interrupt rate which might slow down the overall PC system. Even if the trigger setup will not use this line for a generated frame output this event will occour. This event can only occur if the acquisition is running.

4.1.4. FG_END_OF_LINE_CAM_PORT_0

This event is generated when the last pixel of camera line has arrives at the framegrabber. Keep in mind that a high linerate can cause a critical high interrupt rate which might slow down the overall PC system. This event can only occur if the acquisition is running.

Chapter 5. Sensor Geometry

Some operations, for example mirroring or tap sorting, require knowledge on the sensor dimension and orientation of the camera. The following parameters supply this kind of information.

5.1. FG_VANTAGEPOINT

This parameter defines the vantage point. Use this parameter to mirror the image. Note that when using this parameter for mirroring, the received sensor image is mirrored and not the selected ROI in the frame grabber. Therefore, to mirror the ROI in the frame grabber, ensure to set the correct offsets in the frame grabber.

If a horizontal mirroring is active, the parameter *FG_SENSORWIDTH* limits the maximum width. The parameter dependency will then be *FG_XOFFSET* + *FG_WIDTH* <= *FG_SENSORWIDTH*.

If a vertical mirroring is active , the parameter *FG_SENSORHEIGHT* limits the maximum height. The parameter dependency will then be *FG_YOFFSET* + *FG_HEIGHT* <= *FG_SENSORHEIGHT*.

Table 5.1. Parameter properties of FG_VANTAGEPOINT

Property	Value
Name	**FG_VANTAGEPOINT**
Display Name	**Vantage Point**
Type	**Enumeration**
Access policy	**Read/Write**
Storage policy	**Persistent**
Allowed values	**FG_VANTAGEPOINT_TOP_LEFT** Top Left **FG_VANTAGEPOINT_TOP_RIGHT** Top Right **FG_VANTAGEPOINT_BOTTOM_LEFT** Bottom Left **FG_VANTAGEPOINT_BOTTOM_RIGHT** Bottom Right
Default value	**FG_VANTAGEPOINT_TOP_LEFT**

Example 5.1. Usage of FG_VANTAGEPOINT

```
int result = 0;
int value = FG_VANTAGEPOINT_TOP_LEFT;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_VANTAGEPOINT, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_VANTAGEPOINT, &value, 0, type)) < 0) {
    /* error handling */
}
```

5.2. FG_SENSORWIDTH

To mirror the incoming data correctly, the parameter *FG_SENSORWIDTH* is required. The value of *FG_SENSORWIDTH* is ignored, if *FG_VANTAGEPOINT* = Top-**Left** or Bottom-**Left**. If also a vertical mirroring is used, the available DRAM and sensor height limit the maximum sensor width. This is so, because the sensor image needs to fit twice into the DRAM, because double buffering is used.

If No Mirroring Is Active, the Value of *FG_SENSORWIDTH* Is Not Used

If no mirroring is active, the value of the parameter *FG_SENSORWIDTH* is not used. Instead, the sum of *FG_XOFFSET* and *FG_WIDTH* is used. This makes the use of the module easier as an extra configuration is avoided, if defaults are used.

Table 5.2. Parameter properties of FG_SENSORWIDTH

Property	Value
Name	**FG_SENSORWIDTH**
Display Name	**Sensor Width**
Type	**Unsigned Integer**
Access policy	**Read/Write**
Storage policy	**Persistent**
Allowed values	**Minimum 8** **Maximum 32768** **Stepsize 4**
Default value	**1024**
Unit of measure	**pixel**

Example 5.2. Usage of FG_SENSORWIDTH

```
int result = 0;
unsigned int value = 1024;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_setParameterWithType(fg, FG_SENSORWIDTH, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_SENSORWIDTH, &value, 0, type)) < 0) {
    /* error handling */
}
```

5.3. FG_SENSORHEIGHT

For vertical mirroring or tap geometry sorting in vertical direction, the applet needs to be parameterized with the exact height transferred from the camera to the frame grabber. If you have set a region of interest in the camera, the parameter *FG_SENSORHEIGHT* needs to be set to the ROI size, otherwise use the sensor height.

If Only One Y-Zone Is Used and No Vertical Mirroring Is Active, the Value of *FG_SENSORHEIGHT* Is Not Used

If no vertical mirroring is configured the value of the parameter *FG_SENSORHEIGHT* is not used. Instead, the sum of *FG_YOFFSET* and *FG_HEIGHT* is used. This makes the use of the module easier as an extra configuration is avoided, if defaults are used.

Table 5.3. Parameter properties of FG_SENSORHEIGHT

Property	Value
Name	**FG_SENSORHEIGHT**
Display Name	**Sensor Height**
Type	**Unsigned Integer**
Access policy	**Read/Write**
Storage policy	**Persistent**
Allowed values	**Minimum 1** **Maximum 8388607** **Stepsize 1**
Default value	**1024**
Unit of measure	**pixel**

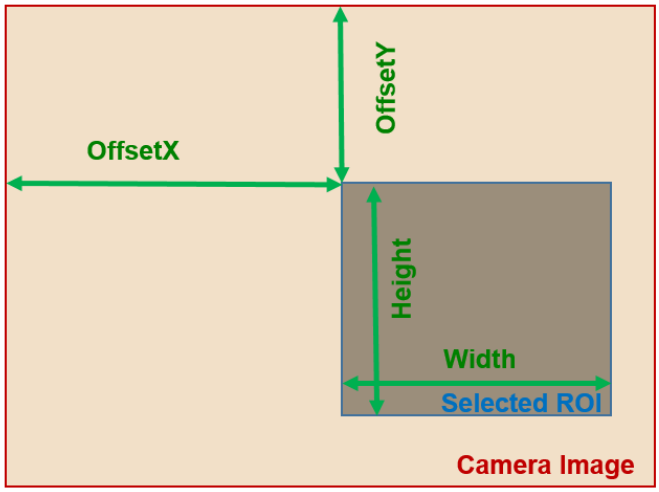
Example 5.3. Usage of FG_SENSORHEIGHT

```
int result = 0;
unsigned int value = 1024;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_setParameterWithType(fg, FG_SENSORHEIGHT, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_SENSORHEIGHT, &value, 0, type)) < 0) {
    /* error handling */
}
```

Chapter 6. ROI

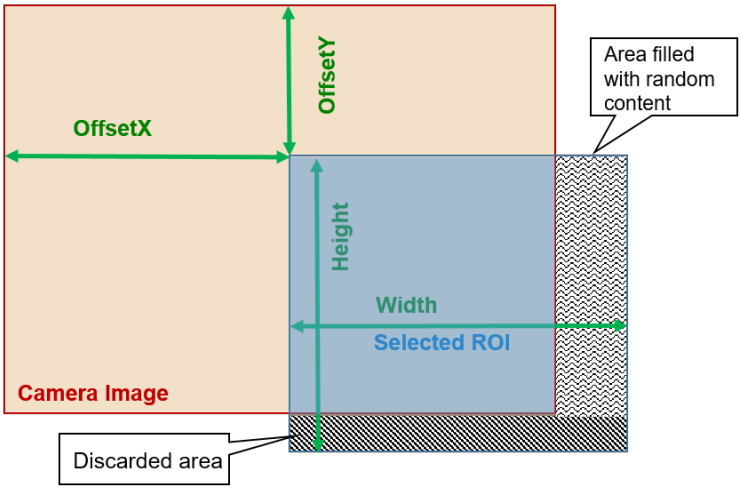
This module allows the definition of a region of interest (ROI), also called area of interest (AOI). A ROI allows the selection of a smaller subset pixel area from the input image. It is defined by using parameters *FG_XOFFSET*, *FG_WIDTH*, *FG_YOFFSET* and *FG_HEIGHT*. The following figure illustrates the parameters.

Figure 6.1. Region of Interest



As can be seen, the region of interest lies within the input image dimensions. Thus, if the image dimension provided by the camera is greater or equal to the specified ROI parameters, the applet will fully cut-out the ROI subset pixel area. However, if the image provided by the camera is smaller than the specified ROI, lines will be filled with random pixel content and the image height might be cut or filled with random image lines as illustrated in the following.

Figure 6.2. Region of Interest Selection Outside the Input Image Dimensions



Furthermore, mind that the image sent by the camera must not exceed the maximum allowed image dimensions. This applet allows a maximum image width of 32768 pixels and a maximum image height of 8388607 lines. The chosen ROI settings can have a direct influence on the maximum bandwidth of the applet as they define the image size and thus, define the amount of data.

The parameters have dynamic value ranges. For example an x-offset cannot be set if the sum of the offset and the image width will exceed the maximum image width. To set a high x-offset, the image width has to be reduced, first. Hence, the order of setting the parameters for this module is important. The return values of the function calls in the SDK should always be evaluated to check if changes were accepted.

Mind the minimum step size of the parameters. This applet has a minimum step size of 4 pixel for the width and the x-offset, while the step size for the height and the y-offset is 1.

The settings made in this module will define the display size and buffer size if the applet is used in microDisplay. If you use the applet in your own programs, ensure to define a sufficient buffer size for the DMA transfers in your PC memory.

All ROI parameters can only be changed if the acquisition is not started i.e. stopped.

Camera ROI

Most cameras allow the setting of a ROI inside the camera. The ROI settings described in this section are independent from the camera settings.

Influence on Bandwidth

A ROI might cause a strong reduction of the required bandwidth. If possible, the camera frame dimension should be reduced directly in the camera to the desired size instead of reducing the size in the applet. This will reduce the required bandwidth between the camera and the interface card.

6.1. FG_WIDTH

The parameter specifies the width of the ROI. The values of parameters *FG_WIDTH* + *FG_XOFFSET* must not exceed the maximum image width of 32768 pixels. If a horizontal mirroring is active the sensor width limits the maximum width (Width + XOffset). If furthermore vertical mirroring is active the maximum width is limited by the DRAM and sensor height (the sensor dimension needs to fit into the DRAM).

Table 6.1. Parameter properties of FG_WIDTH

Property	Value
Name	**FG_WIDTH**
Display Name	**Width**
Type	**Unsigned Integer**
Access policy	**Read/Write**
Storage policy	**Persistent**
Allowed values	**Minimum 8** **Maximum 32768** **Stepsize 4**
Default value	**1024**
Unit of measure	**pixel**

Example 6.1. Usage of FG_WIDTH

```
int result = 0;
unsigned int value = 1024;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_setParameterWithType(fg, FG_WIDTH, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_WIDTH, &value, 0, type)) < 0) {
    /* error handling */
}
```

6.2. FG_HEIGHT

The parameter specifies the height of the ROI. The values of parameters *FG_HEIGHT* + *FG_YOFFSET* must not exceed the maximum image height of 8388607 pixels. If a vertical mirroring is active the sensor height limits the maximum height (Height + YOffset). Furthermore he maximum height is limited by the DRAM and the sensor width (the sensor dimension needs to fit into the DRAM).

Table 6.2. Parameter properties of FG_HEIGHT

Property	Value
Name	**FG_HEIGHT**
Display Name	**Height**
Type	**Unsigned Integer**
Access policy	**Read/Write**
Storage policy	**Persistent**
Allowed values	**Minimum 1** **Maximum 8388607** **Stepsize 1**
Default value	**1024**
Unit of measure	**pixel**

Example 6.2. Usage of FG_HEIGHT

```
int result = 0;
unsigned int value = 1024;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_setParameterWithType(fg, FG_HEIGHT, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_HEIGHT, &value, 0, type)) < 0) {
    /* error handling */
}
```

6.3. FG_XOFFSET

The x-offset is defined by this parameter. If a horizontal mirroring is active the sensor width limits the maximum width (Width + XOffset). If furthermore vertical mirroring is active the maximum width is limited by the DRAM and the sensor height (the sensor dimension needs to fit into the DRAM).

Table 6.3. Parameter properties of FG_XOFFSET

Property	Value
Name	**FG_XOFFSET**
Display Name	**Offset X**
Type	**Unsigned Integer**
Access policy	**Read/Write**
Storage policy	**Persistent**
Allowed values	**Minimum 0** **Maximum 32760** **Stepsize 4**
Default value	**0**
Unit of measure	**pixel**

Example 6.3. Usage of FG_XOFFSET

```
int result = 0;
```

```
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_setParameterWithType(fg, FG_XOFFSET, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_XOFFSET, &value, 0, type)) < 0) {
    /* error handling */
}
```

6.4. FG_YOFFSET

The y-offset is defined by this parameter. If a vertical mirroring is active the sensor height limits the maximum height (Height + YOffset). Furthermore the maximum height is limited by the DRAM and the sensor width (the sensor dimension needs to fit into the DRAM).

Table 6.4. Parameter properties of FG_YOFFSET

Property	Value
Name	**FG_YOFFSET**
Display Name	**Offset Y**
Type	**Unsigned Integer**
Access policy	**Read/Write**
Storage policy	**Persistent**
Allowed values	**Minimum 0** **Maximum 8388606** **Stepsize 1**
Default value	**0**
Unit of measure	**pixel**

Example 6.4. Usage of FG_YOFFSET

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_setParameterWithType(fg, FG_YOFFSET, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_YOFFSET, &value, 0, type)) < 0) {
    /* error handling */
}
```

Chapter 7. Digital I/O

The frame grabber provides digital inputs and digital outputs for triggering, light synchronization, hardware control etc. This CXP-12 Interface Card 1C frame grabber has

- 4 front general purpose inputs (Front GPIs) using the connector on the frame grabber slot bracket.
- 2 front general purpose outputs using the connector on the frame grabber slot bracket.
- trigger over CXP cable function
- **GND**: Value set to GND, zero. For digital outputs check for possibly inverted outputs.
- **VCC**: Value set to VCC, one. For digital outputs check for possibly inverted outputs.
- **FG_SIGNAL_CAM0_EXSYNC**: The Exsync signal. Usually the line trigger signal used to trigger the camera. Check Chapter 8, '*Line Trigger / ExSync*' for more information.
- **FG_SIGNAL_CAM0_EXSYNC2**: The Exsync 2 signal a delayed exsync signal. Check *FG_LINETRIGGERDELAY* for more information.
- **FG_SIGNAL_CAM0_FLASH**: The flash signal. It is generated once at the start of each frame generated by the trigger module. Check Chapter 9, '*Image Trigger / Flash*' for more information.
- **FG_SIGNAL_CAM0_LVAL**: The line valid signal of the received camera or simulator image data. The signal is high for the duration of the line data transfer.
- **FG_SIGNAL_CAM0_FVAL**: The frame valid signal after the trigger module. The signal is high for the duration of the frame data transfer. Depending on the image trigger mode, the image dimension and timing the signal can vary. See Chapter 9, '*Image Trigger / Flash*' for more information.
- **FG_SIGNAL_FRONT_GPI_0** to **FG_SIGNAL_FRONT_GPI_1**: Direct mapping of the digital input signal after debouncing.
- **FG_SIGNAL_CAM0_LINE_START**: Line start pulse. Use for events and signal analyzer.
- **FG_SIGNAL_CAM0_LINE_END**: Line end pulse. Use for events and signal analyzer.
- **FG_SIGNAL_CAM0_FRAME_START**: Frame start pulse. Use for events and signal analyzer.
- **FG_SIGNAL_CAM0_FRAME_END**: Frame end pulse. Use for events and signal analyzer.

7.1. Camera

For CoaXPress triggering, packets are sent to the camera instead of signals. A trigger signal usually consists of a pulse of a certain pulse length defining, for example, the duration time of the exposure. The start of the pulse, i.e. the rising edge, defines the start of the exposure. For most cameras the moment of this rising edge of the pulse is used to send a CXP trigger on CXP LinkTigger0. At the time of the falling edge, the CXP LinkTrigger1 is used by many cameras to end the exposure in a trigger controlled mode.

Thus, you need to select the source signals for the CXP link triggers and define whether you want to use the rising or falling edge. You can do this with the following parameter. Note that the camera must match with these settings.

7.1.1. FG_TRIGGERCAMERA_SOURCE_CXP0

Table 7.1. Parameter properties of FG_TRIGGERCAMERA_SOURCE_CXP0

Property	Value	
Name	**FG_TRIGGERCAMERA_SOURCE_CXP0**	
Display Name	**CXP Link Trigger 0 Source**	
Type	**Enumeration**	
Access policy	**Read/Write/Change**	
Storage policy	**Persistent**	
Allowed values	**GND**	GND
	VCC	VCC
	FG_SIGNAL_CAM0_EXSYNC	Signal Exsync
	FG_SIGNAL_CAM0_EXSYNC2	Signal Exsync2
	FG_SIGNAL_CAM0_FLASH	Signal Flash
	FG_SIGNAL_CAM0_LVAL	Signal Line Valid
	FG_SIGNAL_CAM0_FVAL	Signal Frame Valid
	FG_SIGNAL_FRONT_GPI_0	Signal Front GPI 0
	FG_SIGNAL_FRONT_GPI_1	Signal Front GPI 1
Default value	**FG_SIGNAL_CAM0_EXSYNC**	

Example 7.1. Usage of FG_TRIGGERCAMERA_SOURCE_CXP0

```
int result = 0;
int value = FG_SIGNAL_CAM0_EXSYNC;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_TRIGGERCAMERA_SOURCE_CXP0, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_TRIGGERCAMERA_SOURCE_CXP0, &value, 0, type)) < 0) {
    /* error handling */
}
```

7.1.2. FG_TRIGGERCAMERA_SOURCE_EDGE_CXP0

Table 7.2. Parameter properties of FG_TRIGGERCAMERA_SOURCE_EDGE_CXP0

Property	Value	
Name	**FG_TRIGGERCAMERA_SOURCE_EDGE_CXP0**	
Display Name	**CXP Link Trigger 0 Source Edge**	
Type	**Enumeration**	
Access policy	**Read/Write/Change**	
Storage policy	**Persistent**	
Allowed values	**FG_RISING_EDGE**	Rising Edge
	FG_FALLING_EDGE	Falling Edge
Default value	**FG_RISING_EDGE**	

Example 7.2. Usage of FG_TRIGGERCAMERA_SOURCE_EDGE_CXP0

```
int result = 0;
int value = FG_RISING_EDGE;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_TRIGGERCAMERA_SOURCE_EDGE_CXP0, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_TRIGGERCAMERA_SOURCE_EDGE_CXP0, &value, 0, type)) < 0) {
    /* error handling */
}
```

7.1.3. FG_TRIGGERCAMERA_SOURCE_CXP1

Table 7.3. Parameter properties of FG_TRIGGERCAMERA_SOURCE_CXP1

Property	Value	
Name	**FG_TRIGGERCAMERA_SOURCE_CXP1**	
Display Name	**CXP Link Trigger 1 Source**	
Type	**Enumeration**	
Access policy	**Read/Write/Change**	
Storage policy	**Persistent**	
Allowed values	**GND**	GND
	VCC	VCC
	FG_SIGNAL_CAM0_EXSYNC	Signal Exsync
	FG_SIGNAL_CAM0_EXSYNC2	Signal Exsync2
	FG_SIGNAL_CAM0_FLASH	Signal Flash
	FG_SIGNAL_CAM0_LVAL	Signal Line Valid
	FG_SIGNAL_CAM0_FVAL	Signal Frame Valid
	FG_SIGNAL_FRONT_GPI_0	Signal Front GPI 0
	FG_SIGNAL_FRONT_GPI_1	Signal Front GPI 1
Default value	**FG_SIGNAL_CAM0_EXSYNC**	

Example 7.3. Usage of FG_TRIGGERCAMERA_SOURCE_CXP1

```
int result = 0;
int value = FG_SIGNAL_CAM0_EXSYNC;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_TRIGGERCAMERA_SOURCE_CXP1, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_TRIGGERCAMERA_SOURCE_CXP1, &value, 0, type)) < 0) {
    /* error handling */
}
```

7.1.4. FG_TRIGGERCAMERA_SOURCE_EDGE_CXP1

Table 7.4. Parameter properties of FG_TRIGGERCAMERA_SOURCE_EDGE_CXP1

Property	Value	
Name	**FG_TRIGGERCAMERA_SOURCE_EDGE_CXP1**	
Display Name	**CXP Link Trigger 1 Source Edge**	
Type	**Enumeration**	
Access policy	**Read/Write/Change**	
Storage policy	**Persistent**	
Allowed values	**FG_RISING_EDGE**	Rising Edge
	FG_FALLING_EDGE	Falling Edge
Default value	**FG_FALLING_EDGE**	

Example 7.4. Usage of FG_TRIGGERCAMERA_SOURCE_EDGE_CXP1

```
int result = 0;
int value = FG_FALLING_EDGE;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_TRIGGERCAMERA_SOURCE_EDGE_CXP1, &value, 0, type)) < 0) {
    /* error handling */
```

```
}

if ((result = Fg_getParameterWithType(fg, FG_TRIGGERCAMERA_SOURCE_EDGE_CXP1, &value, 0, type)) < 0) {
    /* error handling */
}
```

7.1.5. FG_TRIGGERCAMERA_SOURCE_CXP2

Table 7.5. Parameter properties of FG_TRIGGERCAMERA_SOURCE_CXP2

Property	Value	
Name	**FG_TRIGGERCAMERA_SOURCE_CXP2**	
Display Name	**CXP Link Trigger 2 Source**	
Type	**Enumeration**	
Access policy	**Read/Write/Change**	
Storage policy	**Persistent**	
Allowed values	**GND**	GND
	VCC	VCC
	FG_SIGNAL_CAM0_EXSYNC	Signal Exsync
	FG_SIGNAL_CAM0_EXSYNC2	Signal Exsync2
	FG_SIGNAL_CAM0_FLASH	Signal Flash
	FG_SIGNAL_CAM0_LVAL	Signal Line Valid
	FG_SIGNAL_CAM0_FVAL	Signal Frame Valid
	FG_SIGNAL_FRONT_GPI_0	Signal Front GPI 0
	FG_SIGNAL_FRONT_GPI_1	Signal Front GPI 1
Default value	**GND**	

Example 7.5. Usage of FG_TRIGGERCAMERA_SOURCE_CXP2

```
int result = 0;
int value = GND;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_TRIGGERCAMERA_SOURCE_CXP2, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_TRIGGERCAMERA_SOURCE_CXP2, &value, 0, type)) < 0) {
    /* error handling */
}
```

7.1.6. FG_TRIGGERCAMERA_SOURCE_EDGE_CXP2

Table 7.6. Parameter properties of FG_TRIGGERCAMERA_SOURCE_EDGE_CXP2

Property	Value	
Name	**FG_TRIGGERCAMERA_SOURCE_EDGE_CXP2**	
Display Name	**CXP Link Trigger 2 Source Edge**	
Type	**Enumeration**	
Access policy	**Read/Write/Change**	
Storage policy	**Persistent**	
Allowed values	**FG_RISING_EDGE**	Rising Edge
	FG_FALLING_EDGE	Falling Edge
Default value	**FG_RISING_EDGE**	

Example 7.6. Usage of FG_TRIGGERCAMERA_SOURCE_EDGE_CXP2

```
int result = 0;
int value = FG_RISING_EDGE;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_TRIGGERCAMERA_SOURCE_EDGE_CXP2, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_TRIGGERCAMERA_SOURCE_EDGE_CXP2, &value, 0, type)) < 0) {
    /* error handling */
}
```

7.1.7. FG_TRIGGERCAMERA_SOURCE_CXP3

Table 7.7. Parameter properties of FG_TRIGGERCAMERA_SOURCE_CXP3

Property	Value	
Name	**FG_TRIGGERCAMERA_SOURCE_CXP3**	
Display Name	**CXP Link Trigger 3 Source**	
Type	**Enumeration**	
Access policy	**Read/Write/Change**	
Storage policy	**Persistent**	
Allowed values	**GND**	GND
	VCC	VCC
	FG_SIGNAL_CAM0_EXSYNC	Signal Exsync
	FG_SIGNAL_CAM0_EXSYNC2	Signal Exsync2
	FG_SIGNAL_CAM0_FLASH	Signal Flash
	FG_SIGNAL_CAM0_LVAL	Signal Line Valid
	FG_SIGNAL_CAM0_FVAL	Signal Frame Valid
	FG_SIGNAL_FRONT_GPI_0	Signal Front GPI 0
	FG_SIGNAL_FRONT_GPI_1	Signal Front GPI 1
Default value	**GND**	

Example 7.7. Usage of FG_TRIGGERCAMERA_SOURCE_CXP3

```
int result = 0;
int value = GND;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_TRIGGERCAMERA_SOURCE_CXP3, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_TRIGGERCAMERA_SOURCE_CXP3, &value, 0, type)) < 0) {
    /* error handling */
}
```

7.1.8. FG_TRIGGERCAMERA_SOURCE_EDGE_CXP3

Table 7.8. Parameter properties of FG_TRIGGERCAMERA_SOURCE_EDGE_CXP3

Property	Value
Name	**FG_TRIGGERCAMERA_SOURCE_EDGE_CXP3**
Display Name	**CXP Link Trigger 3 Source Edge**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**FG_RISING_EDGE** Rising Edge **FG_FALLING_EDGE** Falling Edge
Default value	**FG_RISING_EDGE**

Example 7.8. Usage of FG_TRIGGERCAMERA_SOURCE_EDGE_CXP3

```
int result = 0;
int value = FG_RISING_EDGE;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_TRIGGERCAMERA_SOURCE_EDGE_CXP3, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_TRIGGERCAMERA_SOURCE_EDGE_CXP3, &value, 0, type)) < 0) {
    /* error handling */
}
```

7.2. GPO

7.2.1. FG_TRIGGEROUT_FRONT_GPO_0_SOURCE et al.

Note

This description applies also to the following parameters: FG_TRIGGEROUT_FRONT_GPO_1_SOURCE

Select the signal source of the Front General Purpose Output (Front GPO). For further explanation of the available sources see Chapter 7, '*Digital I/O*'.

You can change the polarity using parameter *FG_TRIGGEROUT_FRONT_GPO_0_POLARITY*.

Table 7.9. Parameter properties of FG_TRIGGEROUT_FRONT_GPO_0_SOURCE

Property	Value	
Name	**FG_TRIGGEROUT_FRONT_GPO_0_SOURCE**	
Display Name	**Trigger Out Front GPO 0 Source**	
Type	**Enumeration**	
Access policy	**Read/Write/Change**	
Storage policy	**Persistent**	
Allowed values	**GND**	GND
	VCC	VCC
	FG_SIGNAL_CAM0_EXSYNC	Signal Exsync
	FG_SIGNAL_CAM0_EXSYNC2	Signal Exsync2
	FG_SIGNAL_CAM0_FLASH	Signal Flash
	FG_SIGNAL_CAM0_LVAL	Signal Line Valid
	FG_SIGNAL_CAM0_FVAL	Signal Frame Valid
	FG_SIGNAL_FRONT_GPI_0	Signal Front GPI 0
	FG_SIGNAL_FRONT_GPI_1	Signal Front GPI 1
Default value	**FG_SIGNAL_CAM0_FLASH**	

Example 7.9. Usage of FG_TRIGGEROUT_FRONT_GPO_0_SOURCE

```
int result = 0;
int value = FG_SIGNAL_CAM0_FLASH;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_TRIGGEROUT_FRONT_GPO_0_SOURCE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_TRIGGEROUT_FRONT_GPO_0_SOURCE, &value, 0, type)) < 0) {
    /* error handling */
}
```

7.2.2. FG_TRIGGEROUT_FRONT_GPO_0_POLARITY et al.

Note

This description applies also to the following parameters: FG_TRIGGEROUT_FRONT_GPO_1_POLARITY

Select the output polarity the Front General Purpose Output (Front GPO). For further explanation of the available sources see Chapter 7, '*Digital I/O*'.

Table 7.10. Parameter properties of FG_TRIGGEROUT_FRONT_GPO_0_POLARITY

Property	Value	
Name	**FG_TRIGGEROUT_FRONT_GPO_0_POLARITY**	
Display Name	**Trigger Front Out GPO 0 Polarity**	
Type	**Enumeration**	
Access policy	**Read/Write/Change**	
Storage policy	**Persistent**	
Allowed values	**FG_LOW**	Low Active
	FG_HIGH	High Active
Default value	**FG_HIGH**	

Example 7.10. Usage of FG_TRIGGEROUT_FRONT_GPO_0_POLARITY

```
int result = 0;
int value = FG_HIGH;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_TRIGGEROUT_FRONT_GPO_0_POLARITY, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_TRIGGEROUT_FRONT_GPO_0_POLARITY, &value, 0, type)) < 0) {
    /* error handling */
}
```

7.3. GPI

7.3.1. FG_DIGIO_INPUT

Parameter *FG_DIGIO_INPUT* is used to monitor the digital inputs of the frame grabber. This AcquisitionApplets has 4 digital inputs. You can read the current state of these inputs using parameter *FG_DIGIO_INPUT*. Bit 0 of the read value represents input 0, bit 1 represents input 1 and so on. For example, if you obtain the value 37 or hexadecimal 0x25 the frame grabber will have high level on it's digital inputs 0, 2 and 5.

Table 7.11. Parameter properties of FG_DIGIO_INPUT

Property	Value
Name	**FG_DIGIO_INPUT**
Display Name	**Digital Input**
Type	**Unsigned Integer**
Access policy	**Read-Only**
Storage policy	**Persistent**
Allowed values	**Minimum 0** **Maximum 4095** **Stepsize 1**
Unit of measure	

Example 7.11. Usage of FG_DIGIO_INPUT

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_getParameterWithType(fg, FG_DIGIO_INPUT, &value, 0, type)) < 0) {
    /* error handling */
}
```

7.4. Event Source

7.4.1. FG_CUSTOM_SIGNAL_EVENT_0_SOURCE

Select the source for the custom signal event.

Table 7.12. Parameter properties of FG_CUSTOM_SIGNAL_EVENT_0_SOURCE

Property	Value	
Name	**FG_CUSTOM_SIGNAL_EVENT_0_SOURCE**	
Display Name	**Custom Signal Event 0 Source**	
Type	**Enumeration**	
Access policy	**Read/Write/Change**	
Storage policy	**Persistent**	
Allowed values	**GND**	GND
	VCC	VCC
	FG_SIGNAL_CAM0_EXSYNC	Signal Exsync
	FG_SIGNAL_CAM0_EXSYNC2	Signal Exsync2
	FG_SIGNAL_CAM0_FLASH	Signal Flash
	FG_SIGNAL_CAM0_LVAL	Signal Line Valid
	FG_SIGNAL_CAM0_FVAL	Signal Frame Valid
	FG_SIGNAL_CAM0_LINE_START	Signal Line Start
	FG_SIGNAL_CAM0_LINE_END	Cam0 Line Transfer End
	FG_SIGNAL_CAM0_FRAME_START	Signal Frame Start
	FG_SIGNAL_CAM0_FRAME_END	Signal Frame End
	FG_SIGNAL_FRONT_GPI_0	Signal Front GPI 0
	FG_SIGNAL_FRONT_GPI_1	Signal Front GPI 1
Default value	**FG_SIGNAL_CAM0_EXSYNC**	

Example 7.12. Usage of FG_CUSTOM_SIGNAL_EVENT_0_SOURCE

```
int result = 0;
int value = FG_SIGNAL_CAM0_EXSYNC;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_CUSTOM_SIGNAL_EVENT_0_SOURCE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_CUSTOM_SIGNAL_EVENT_0_SOURCE, &value, 0, type)) < 0) {
    /* error handling */
}
```

7.4.2. FG_CUSTOM_SIGNAL_EVENT_0_POLARITY

Select the polarity for the custom signal event.

Table 7.13. Parameter properties of FG_CUSTOM_SIGNAL_EVENT_0_POLARITY

Property	Value	
Name	**FG_CUSTOM_SIGNAL_EVENT_0_POLARITY**	
Display Name	**Custom Signal Event 0 Polarity**	
Type	**Enumeration**	
Access policy	**Read/Write/Change**	
Storage policy	**Persistent**	
Allowed values	**FG_LOW**	Low Active
	FG_HIGH	High Active
Default value	**FG_HIGH**	

Example 7.13. Usage of FG_CUSTOM_SIGNAL_EVENT_0_POLARITY

```
int result = 0;
```

```
int value = FG_HIGH;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_CUSTOM_SIGNAL_EVENT_0_POLARITY, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_CUSTOM_SIGNAL_EVENT_0_POLARITY, &value, 0, type)) < 0) {
    /* error handling */
}
```

7.4.3. FG_CUSTOM_SIGNAL_EVENT_1_SOURCE

Select the source for the custom signal event.

Table 7.14. Parameter properties of FG_CUSTOM_SIGNAL_EVENT_1_SOURCE

Property	Value	
Name	**FG_CUSTOM_SIGNAL_EVENT_1_SOURCE**	
Display Name	**Custom Signal Event 1 Source**	
Type	**Enumeration**	
Access policy	**Read/Write/Change**	
Storage policy	**Persistent**	
Allowed values	**GND**	GND
	VCC	VCC
	FG_SIGNAL_CAM0_EXSYNC	Signal Exsync
	FG_SIGNAL_CAM0_EXSYNC2	Signal Exsync2
	FG_SIGNAL_CAM0_FLASH	Signal Flash
	FG_SIGNAL_CAM0_LVAL	Signal Line Valid
	FG_SIGNAL_CAM0_FVAL	Signal Frame Valid
	FG_SIGNAL_CAM0_LINE_START	Signal Line Start
	FG_SIGNAL_CAM0_LINE_END	Cam0 Line Transfer End
	FG_SIGNAL_CAM0_FRAME_START	Signal Frame Start
	FG_SIGNAL_CAM0_FRAME_END	Signal Frame End
	FG_SIGNAL_FRONT_GPI_0	Signal Front GPI 0
	FG_SIGNAL_FRONT_GPI_1	Signal Front GPI 1
Default value	**FG_SIGNAL_CAM0_FLASH**	

Example 7.14. Usage of FG_CUSTOM_SIGNAL_EVENT_1_SOURCE

```
int result = 0;
int value = FG_SIGNAL_CAM0_FLASH;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_CUSTOM_SIGNAL_EVENT_1_SOURCE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_CUSTOM_SIGNAL_EVENT_1_SOURCE, &value, 0, type)) < 0) {
    /* error handling */
}
```

7.4.4. FG_CUSTOM_SIGNAL_EVENT_1_POLARITY

Select the polarity for the custom signal event.

Table 7.15. Parameter properties of FG_CUSTOM_SIGNAL_EVENT_1_POLARITY

Property	Value
Name	**FG_CUSTOM_SIGNAL_EVENT_1_POLARITY**
Display Name	**Custom Signal Event 1 Polarity**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**FG_LOW** Low Active **FG_HIGH** High Active
Default value	**FG_HIGH**

Example 7.15. Usage of FG_CUSTOM_SIGNAL_EVENT_1_POLARITY

```
int result = 0;
int value = FG_HIGH;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_CUSTOM_SIGNAL_EVENT_1_POLARITY, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_CUSTOM_SIGNAL_EVENT_1_POLARITY, &value, 0, type)) < 0) {
    /* error handling */
}
```

7.5. Events

In programming or runtime environments, a callback function is a piece of executable code that is passed as an argument, which is expected to call back (execute) exactly that time an event is triggered. This applet can generate some software callback events based on trigger inputs as explained in the following section. These events are not related to a special camera functionality. Other event sources are described in additional sections of this document.

Basler Framegrabber SDK enables an application to get these event notifications about certain state changes at the data flow from camera to RAM and the image and trigger processing as well. Please consult the Basler Framegrabber SDK documentation for more details concerning the implementation of this functionality.

7.5.1. FG_TRIGGER_INPUT0_RISING

This event is generated for each rising signal edge at trigger input 0. Except for the timestamp, the event has no additional data included. Keep in mind that fast changes of the input signal can cause high interrupt rates which might slow down the system. This event can occur independent of the acquisition status.

7.5.2. FG_TRIGGER_INPUT0_FALLING

This event is generated for each falling signal edge at trigger input 0. Except for the timestamp, the event has no additional data included. Keep in mind that fast changes of the input signal can cause high interrupt rates which might slow down the system. This event can occur independent of the acquisition status.

7.5.3. FG_CUSTOM_SIGNAL_EVENT_0

The event defined by *FG_CUSTOM_SIGNAL_EVENT_0_SOURCE* and *FG_CUSTOM_SIGNAL_EVENT_0_POLARITY*.

7.5.4. FG_CUSTOM_SIGNAL_EVENT_1

The event defined by *FG_CUSTOM_SIGNAL_EVENT_1_SOURCE* and *FG_CUSTOM_SIGNAL_EVENT_1_POLARITY*.

Chapter 8. Line Trigger / ExSync

The line trigger function block uses signals to control the line scan acquisition of the specific camera. A external synchronization signal or internal generated puls with fixed frequency being sent to the line scan camera is called ExSync. With the help of this signal it is possible to control the exposure of the connected camera.

The camera needs to be configured accordingly to use the ExSync as control signal. Furthermore the camera might expect the ExSync at a particular CC signal and/or polarity.

For CoaXPress the the exposure control is sent in two independent packets. A single start- and a single end-packet. The time in between is interpreted as pulse width. The timing of these is very precise.

An sensor exposure control based on pulse length/duration is very common. Please make sure that the exposure time is less than the period of the expected maximum line frequency. Consult the camera's manual for more details because these are device specific. More details concerning ExSync can be found in the parameter description of *FG_EXSYNCON*.

Basically two different generation modes for the ExSync signals are available,

- a simple periodical and
- an externally triggered generation.

Additionally, two variants of these are available,

- the first is independent from the image gate,
- and the second is gated by the image gate, which creates ExSync signals only during the actual acquisition.

All details can be found in the parameter description of *FG_LINETRIGGERMODE*.

For the mapping of the ExSync signals to the digital outputs check Chapter 7, '*Digital I/O*'.

8.1. FG_LINETRIGGERMODE

Please choose one of the line trigger modes described here. Make sure that the operation modes of the frame grabber and the camera are the same.

Image independent ExSync modes:

- **Grabber Controlled**

  For the grabber controlled line trigger, the ExSync signal is a simple periodical signal. Its period defines the line frequency and its active time is used by many cameras to define the exposure time.

- **External Trigger**

  The external trigger mode for ExSync generates a single ExSync pulse when the external trigger source becomes active. The ExSync defines the exposure time for the camera. During the exposure time is not possible to re-trigger the ExSync. If the camera needs an additional setup time, it is possible to extend the deadtime of the trigger - the time where no re-trigger is possible - beyond the exposure time. If you want to trigger fewer lines than pulses available at the trigger input, it is possible to downscale the trigger input, e.g. a downscaler of 2 will generate an ExSync every 2nd input pulse, a downscaler of 3 only every third of the input pulses, and so on.

Image gate dependent ExSync modes:

- **Grabber Controlled Gated**

For the grabber controlled gated line trigger, the ExSync signal is generated the very same way as for the grabber controlled mode described above. However, the generator for the ExSync is starting the rising image gate and stops with the image gate becoming inactive. This gives a smaller jitter for the time from the start of the image gate and the generation of the first ExSync, especially for very long ExSync periods.

- **External Trigger Gated**

  For the external trigger gated controlled line trigger, the ExSync signal is generated the very same way as for the external trigger mode described above. However, the generator for the ExSync is starting the rising image gate and stops with the image gate becoming inactive. For this mode two downscalers are available. The first is the downscaler from the beginning of the image gate to the first ExSync, it is called phase. The second is downscaling all succeeding input triggers and is the same as the downscaler used in external trigger mode described above. The options downscale and phase allow further adjustment of the camera trigger with respect to its external source, the trigger input. The value downscale determines the divisor of the input frequency, e.g. a downscale of 16 will produce an ExSync every 16 * n of the input trigger. Furthermore, the phase gives the possibility to shift the camera trigger. A phase shift of 90° is achieved when setting phase to 4, which produces a camera trigger at times 16 * n + 4 of the input trigger signal.

Table 8.1. Parameter properties of FG_LINETRIGGERMODE

Property	Value	
Name	**FG_LINETRIGGERMODE**	
Display Name	**Line Trigger Mode**	
Type	**Enumeration**	
Access policy	**Read/Write**	
Storage policy	**Persistent**	
Allowed values	**GRABBER_CONTROLLED** **ASYNC_TRIGGER** **GRABBER_CONTROLLED_GATED** **ASYNC_GATED**	Grabber Controlled Async External Trigger Grabber Controlled Gated Async Gated Trigger
Default value	**GRABBER_CONTROLLED**	

Example 8.1. Usage of FG_LINETRIGGERMODE

```
int result = 0;
int value = GRABBER_CONTROLLED;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_LINETRIGGERMODE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_LINETRIGGERMODE, &value, 0, type)) < 0) {
    /* error handling */
}
```

8.2. FG_EXSYNCON

This parameter enables the transmission of ExSync signals to the camera.

Please take care to first start the acquisition before setting this ExSyncOn parameter to On (**FG_ON**) if you want to acquire all lines being generated by the camera. The signal will be sent as soon as the ExSync has been started. As soon as the acquisition is started the used timeout parameter becomes valid independent of the ExSyncOn parameter being On (**FG_ON**) or Off (**FG_OFF**). By switching this parameter On (**FG_ON**) and Off (**FG_OFF**) during an acquisition you can check if the camera is configured to use this external signal for exposure start.

Whether the ExSync is really used by the camera is based on the settings of the camera. Consult the camera's manual for more details because these are device specific.

Table 8.2. Parameter properties of FG_EXSYNCON

Property	Value
Name	**FG_EXSYNCON**
Display Name	**ExSync On**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**FG_ON** On **FG_OFF** Off
Default value	**FG_ON**

Example 8.2. Usage of FG_EXSYNCON

```
int result = 0;
int value = FG_ON;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_EXSYNCON, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_EXSYNCON, &value, 0, type)) < 0) {
    /* error handling */
}
```

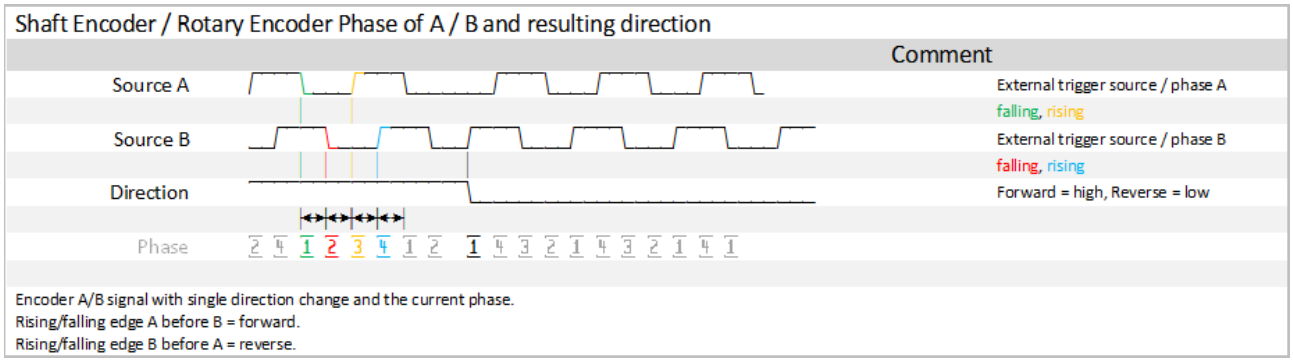
8.3. Line Trigger Input

In the line trigger input category of the line trigger module, the applet is configured for a possible external line trigger input. Here, debouncing times, downscales, polarities and a shaft encoder input are configured.

The external peripheral line trigger source will be in most cases a shaft encoder, also called a rotary encoder. These devices convert the objects movement over an angular motion into relative incremental pulses. The angular motion is taken from the motor axis or a wheel being connected to the translational motion of the scanned object. For most line scan applications it is relevant to get exact feedback of the relative motion between camera and object. By this a certain number of incremental pulses per distance is given to the frame grabber trigger input interface. Depending on the used incremental shaft encoders a certain number (500, 1000, ...) of incremental pulses per rotation is produced.

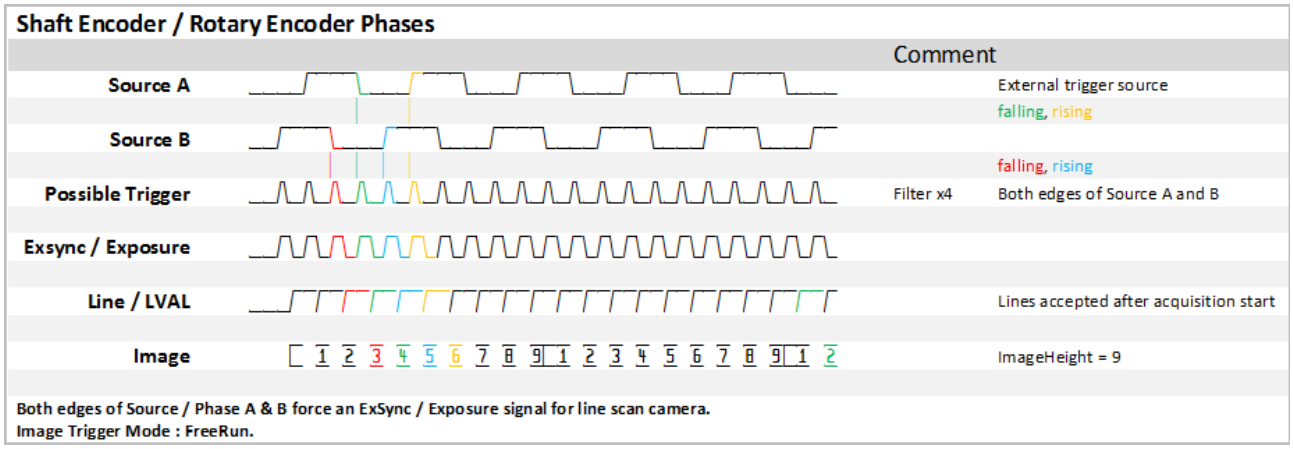
Most incremental shaft encoders provide 2 signals that are called A & B. By using these two signals the relative increments can be seen at the edges of these signals and a direction. In one direction the A-phase high state rises before the B-phase in the other direction, i.e. vice versa. If we do not need a direction for our application, only the A-phase is necessary. A combination of A & B may provide a higher resolution. Please see *FG_SHAFTENCODERMODE* and *FG_SHAFTENCODERON* for this.

Figure 8.1. Shaft Encoder, A & B phase, direction



During an acquisition the shaft encoder signals trigger the ExSync signals and force the sensor to perform an exposure. After the sensor exposure the line is read-out and transfered. The time between exposure and transfer is for most line scan cameras very short.

Figure 8.2. Shaft Encoder, A & B signal, acquisition



The different phases are defined as seen in the following table. A positive phase increment is forward direction, a negative means reverse. This induces rising/falling edge A before B equals forward direction and rising/falling edge B before A means reverse.

Table 8.3. Phases of an A/B Shaft Encoder

Phase	A-state	B-state
1	low	high
2	low	low
3	high	low
4	high	high

Some shaft encoders provide a third signal that is pulsed for each full rotation which is called Z or index. This signal Z could become interesting for an image trigger mode. For more details see Chapter 9, '*Image Trigger / Flash*'.

For most applications and several camera or line scan sensor types it is necessary to have the same resolution in X and Y direction of an image. Due to this the number of pixels per mm in sensor- and motion-direction needs to be the same. In case of an 1024 pixel line scan sensor looking at 10 cm we have 10.24 pixel per mm orthogonal to the web direction. In order to reach an 1:1 scaling we need 10.24 ExSync signals per mm. If a perfectly round object is scanned with an 1:1 scaling then it is exactly round in the image too. When the result becomes elliptic, the scaling is not perfect and some line scan sensor architectures (Bi/Tri-Linear, Dual-Line, ...) will show some additional artefacts.

8.3.1. FG_LINETRIGGERINSRC

This parameter specifies the digital signal source for phase A, which is used to trigger the ExSync signal. If an A/B shaft encoder is used, configure source B at *FG_SHAFTENCODERINSRC*, too. For more details consult the Framegrabber SDK manual.

It is possible to use the shaft encoder A phase only if the direction of scanning is not of interest in the target application. Concerning more details to the shaft encoder please consider the introduction of Section 8.3, 'Line Trigger Input'.

Table 8.4. Parameter properties of FG_LINETRIGGERINSRC

Property	Value
Name	**FG_LINETRIGGERINSRC**
Display Name	**Line Trigger In Source**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**TRGINSRC_FRONT_GPI_0** Trigger In Source Front GPI 0 **TRGINSRC_FRONT_GPI_1** Trigger In Source Front GPI 1 **TRGINSRC_FRONT_GPI_2** Trigger In Source Front GPI 2 **TRGINSRC_FRONT_GPI_3** Trigger In Source Front GPI 3
Default value	**TRGINSRC_FRONT_GPI_1**

Example 8.3. Usage of FG_LINETRIGGERINSRC

```
int result = 0;
int value = TRGINSRC_FRONT_GPI_1;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_LINETRIGGERINSRC, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_LINETRIGGERINSRC, &value, 0, type)) < 0) {
    /* error handling */
}
```

8.3.2. FG_LINETRIGGERINPOLARITY

The parameter defines the polarity of the external input trigger signal encoder source A and source B. When set to LowActive, the ExSync generator starts on a falling edge of the signal specified by the parameter *FG_LINETRIGGERINSRC*. Otherwise, the ExSync generation starts on a rising edge. This is only relevant if the *FG_LINETRIGGERMODE* is set to an external trigger.

Table 8.5. Parameter properties of FG_LINETRIGGERINPOLARITY

Property	Value
Name	**FG_LINETRIGGERINPOLARITY**
Display Name	**Line Trigger In Polarity**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**HIGH_ON_ZERO_LOW** Low Active **HIGH_ON_ZERO_HIGH** High Active
Default value	**HIGH_ACTIVE**

Example 8.4. Usage of FG_LINETRIGGERINPOLARITY

```
int result = 0;
int value = HIGH_ACTIVE;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_LINETRIGGERINPOLARITY, &value, 0, type)) < 0) {
    /* error handling */
}
```

```
if ((result = Fg_getParameterWithType(fg, FG_LINETRIGGERINPOLARITY, &value, 0, type)) < 0) {
    /* error handling */
}
```

8.3.3. FG_LINETRIGGERDEBOUNCING

This parameter specifies the debouncing time. This is the time for which the input line trigger signals must keep the same value to be detected as such. Fast signal changes within the debouncing time will be filtered out.

Table 8.6. Parameter properties of FG_LINETRIGGERDEBOUNCING

Property	Value
Name	**FG_LINETRIGGERDEBOUNCING**
Display Name	**Line Trigger Debouncing**
Type	**Double**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum 0.0032** **Maximum 26.0** **Stepsize 0.0032**
Default value	**0.112**
Unit of measure	**µs**

Example 8.5. Usage of FG_LINETRIGGERDEBOUNCING

```
int result = 0;
double value = 0.112;
const enum FgParamTypes type = FG_PARAM_TYPE_DOUBLE;

if ((result = Fg_setParameterWithType(fg, FG_LINETRIGGERDEBOUNCING, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_LINETRIGGERDEBOUNCING, &value, 0, type)) < 0) {
    /* error handling */
}
```

8.3.4. Downscale

8.3.4.1. FG_LINE_DOWNSCALE

Sets the value after how many pulses of the input trigger signal a single one is passed through as ExSync. For example, a value of 2 creates an ExSync pulse at each 2nd input trigger signal. This is only relevant if the *FG_LINETRIGGERMODE* is set to an external trigger mode. The parameter *FG_LINE_DOWNSCALEINIT* selects an initial delay of incoming pulses.

Figure 8.3. Downscale and Init phase behaviour

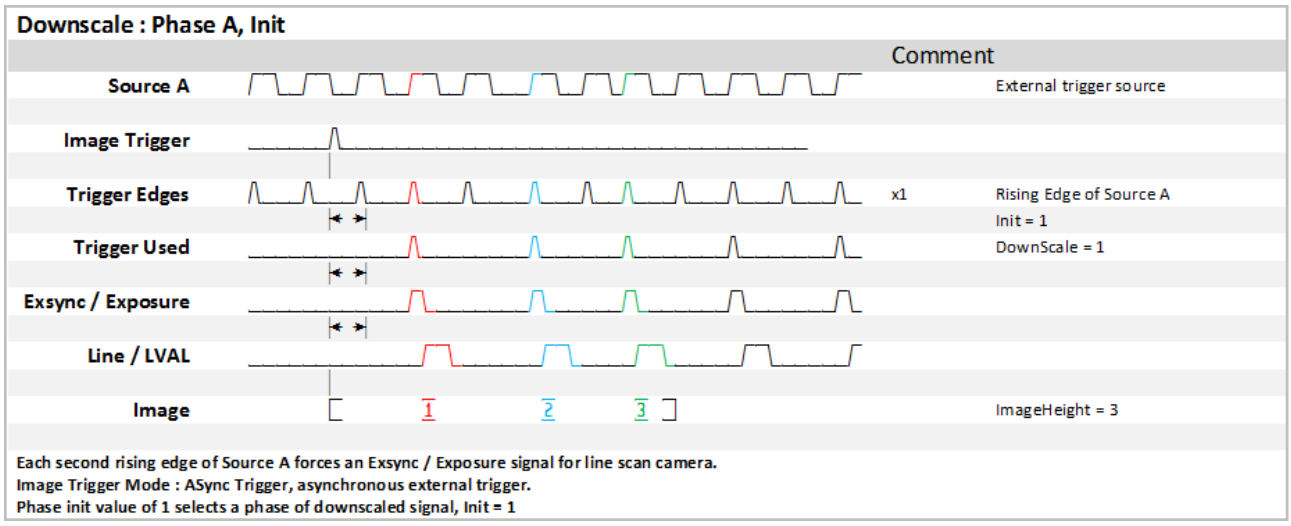


Table 8.7. Parameter properties of FG_LINE_DOWNSCALE

Property	Value
Name	**FG_LINE_DOWNSCALE**
Display Name	**Line Downscale**
Type	**Unsigned Integer**
Access policy	**Read/Write**
Storage policy	**Persistent**
Allowed values	**Minimum 1** **Maximum 255** **Stepsize 1**
Default value	**1**
Unit of measure	**pulses**

Example 8.6. Usage of FG_LINE_DOWNSCALE

```
int result = 0;
unsigned int value = 1;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_setParameterWithType(fg, FG_LINE_DOWNSCALE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_LINE_DOWNSCALE, &value, 0, type)) < 0) {
    /* error handling */
}
```

8.3.4.2. FG_LINE_DOWNSCALEINIT

In addition to the downscale value this parameter sets a phase position. This parameter specifies the number of external input trigger signals, which are needed to generate the first ExSync of a frame. This is only relevant if the *FG_LINETRIGGERMODE* is set to an image gate dependent ExSync mode. This value is applied after the image start pulse. The parameter *FG_LINE_DOWNSCALE* represents the number of possible steps and an explaining figure is found in its description (Init=1).

Table 8.8. Parameter properties of FG_LINE_DOWNSCALEINIT

Property	Value
Name	**FG_LINE_DOWNSCALEINIT**
Display Name	**Line Downscale Init**
Type	**Unsigned Integer**
Access policy	**Read/Write**
Storage policy	**Persistent**
Allowed values	**Minimum 1** **Maximum 255** **Stepsize 1**
Default value	**1**
Unit of measure	**pulses**

Example 8.7. Usage of FG_LINE_DOWNSCALEINIT

```
int result = 0;
unsigned int value = 1;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_setParameterWithType(fg, FG_LINE_DOWNSCALEINIT, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_LINE_DOWNSCALEINIT, &value, 0, type)) < 0) {
    /* error handling */
}
```

8.4. Shaft Encoder A/B Filter

With the support of signal A/B for shaft encoders it is possible to detect the rotary direction of an attached encoder and filter the encoder signals accordingly. Also a compensation is performed for up to 16,777,216 reverse encoder signals. A brief description about this feature is found in the shaft encoder documentation.

8.4.1. FG_SHAFTENCODERON

Switch the shaft encoder filter On or Off. This is only relevant if the *FG_LINETRIGGERMODE* is set to an external trigger mode. The functionalities of *FG_SHAFTENCODERMODE*, *FG_SHAFTENCODERINSRC*, *FG_SHAFTENCODERLEADING*, *FG_SHAFTENCODER_COMPENSATION_ENABLE*, *FG_SHAFTENCODER_COMPENSATION_COUNT* become relevant in the case this parameter is set to On = **FG_ON**. When enabling the shaft encoder, a reset of the encoder compensation is performed. If this filter is switched on an correct A & B encoder signal is expected and necessary for correct functionality. Please be aware that the input signal at *FG_SHAFTENCODERINSRC* is interpreted as phase B and the input signal at *FG_LINETRIGGERINSRC* as phase A. A sketch of the signal can be found in the description of parameter *FG_LINETRIGGERINSRC*.

Table 8.9. Parameter properties of FG_SHAFTENCODERON

Property	Value
Name	**FG_SHAFTENCODERON**
Display Name	**Shaft Encoder On**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**FG_ON** On **FG_OFF** Off
Default value	**FG_OFF**

Example 8.8. Usage of FG_SHAFTENCODERON

```
int result = 0;
int value = FG_OFF;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_SHAFTENCODERON, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_SHAFTENCODERON, &value, 0, type)) < 0) {
    /* error handling */
}
```

8.4.2. FG_SHAFTENCODERMODE

The shaft encoder mode can be run in three operation modes. Please choose the according operation mode for your application. This feature can be used if *FG_SHAFTENCODERON* is switched on. It enables you to adjust the number of increments per rotation of the shaft encoder. Together with the parameter *FG_LINE_DOWNSCALE* you can adjust the increment re-scaling.

The following modes are available:

- Filter x1

  ExSync is generated for a forward rotation of the shaft encoder in single resolution, i.e. a trigger pulse for rising edge of Source A.

- Filter x2

  ExSync is generated for a forward rotation of the shaft encoder in double resolution, i.e. a trigger pulse for a rising and falling edge of Source A, edges of Source B are not used.

- Filter x4

  ExSync is generated for a forward rotation of the shaft encoder in quad resolution, i.e. a trigger pulse for a rising and falling edge of Source A and a rising and falling edge of Source B.

Figure 8.4. Shaft Encoder Mode : Filter x4, x2, x1

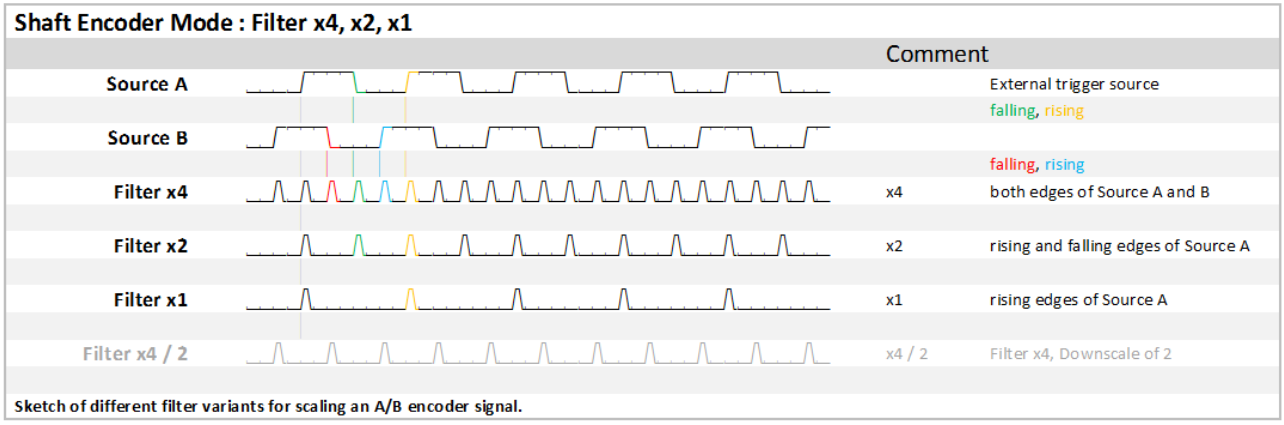


Table 8.10. Parameter properties of FG_SHAFTENCODERMODE

Property	Value
Name	**FG_SHAFTENCODERMODE**
Display Name	**Shaft Encoder Mode**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**FILTER_X1** Filter X1 **FILTER_X2** Filter X2 **FILTER_X4** Filter X4
Default value	**FILTER_X1**

Example 8.9. Usage of FG_SHAFTENCODERMODE

```
int result = 0;
int value = FILTER_X1;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_SHAFTENCODERMODE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_SHAFTENCODERMODE, &value, 0, type)) < 0) {
    /* error handling */
}
```

8.4.3. FG_SHAFTENCODERINSRC

Specifies the input signal source / phase B for the shaft encoder filter. Signal source B of the shaft encoder is 90 degree phase shifted to source / phase A. In this document you can get more explanations regarding the input pins in the context of parameter *FG_LINETRIGGERINSRC* and concerning the shaft encoder in the introduction of Section 8.3, 'Line Trigger Input'. Check the hardware documentation of the microEnable trigger board and the Framegrabber SDK manual for more details.

Table 8.11. Parameter properties of FG_SHAFTENCODERINSRC

Property	Value
Name	**FG_SHAFTENCODERINSRC**
Display Name	**Shaft Encoder Input Source**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**TRGINSRC_FRONT_GPI_0** Trigger In Source Front GPI 0 **TRGINSRC_FRONT_GPI_1** Trigger In Source Front GPI 1 **TRGINSRC_FRONT_GPI_2** Trigger In Source Front GPI 2 **TRGINSRC_FRONT_GPI_3** Trigger In Source Front GPI 3
Default value	**TRGINSRC_FRONT_GPI_2**

Example 8.10. Usage of FG_SHAFTENCODERINSRC

```
int result = 0;
int value = TRGINSRC_FRONT_GPI_2;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_SHAFTENCODERINSRC, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_SHAFTENCODERINSRC, &value, 0, type)) < 0) {
    /* error handling */
}
```

8.4.4. FG_SHAFTENCODERLEADING

This parameter defines the leading signal (= direction) of the shaft encoder filter. This induces rising/falling edge A before B equals forward direction and rising/falling edge B before A means reverse. The default setting is A as the leading signal. Flipping the input pins or their polarity will have the same effect as changing this to B as the leading signal. It simply defines the valid direction of the scan. An explanation of the direction detection based on an encoder A / B signal is found in Section 8.3, 'Line Trigger Input'.

Table 8.12. Parameter properties of FG_SHAFTENCODERLEADING

Property	Value
Name	**FG_SHAFTENCODERLEADING**
Display Name	**Shaft Encoder Leading**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**SOURCE_A** Source A **SOURCE_B** Source B
Default value	**SOURCE_A**

Example 8.11. Usage of FG_SHAFTENCODERLEADING

```
int result = 0;
int value = SOURCE_A;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_SHAFTENCODERLEADING, &value, 0, type)) < 0) {
    /* error handling */
```

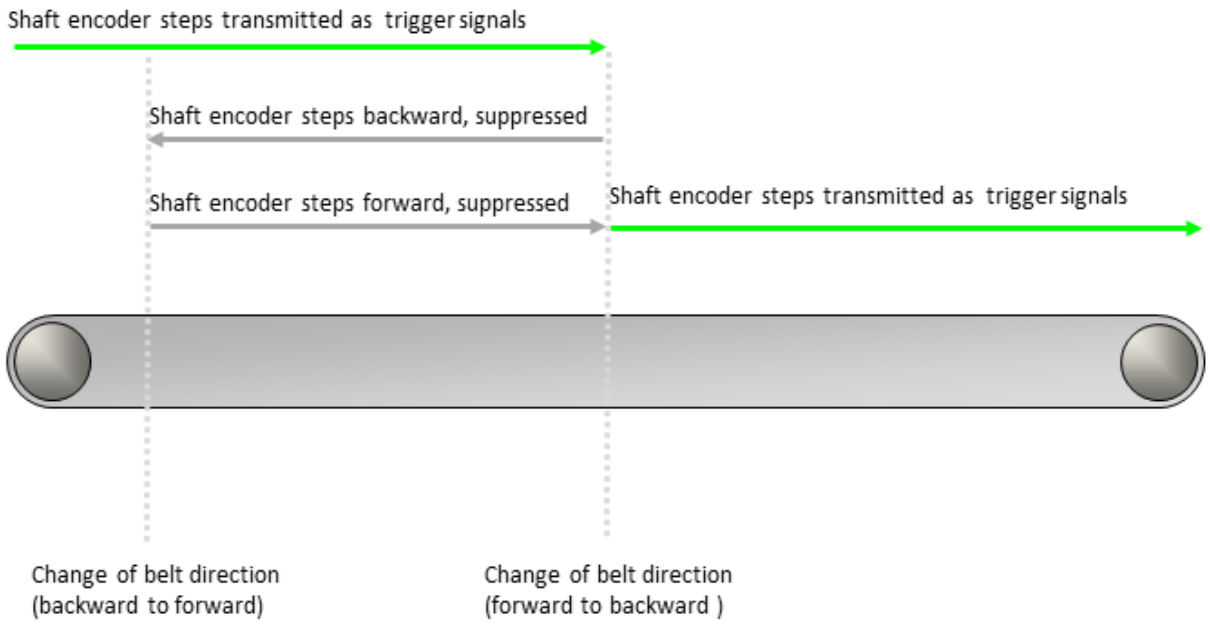
```
}

if ((result = Fg_getParameterWithType(fg, FG_SHAFTENCODERLEADING, &value, 0, type)) < 0) {
    /* error handling */
}
```

8.4.5. FG_SHAFTENCODER_COMPENSATION_ENABLE

The shaft encoder analyzer includes a rollback compensation. In case the rollback compensation is enabled, the module will compensate the reverse movement so that no object is scanned twice. The module will count the number of reverse pulses and will suppress all reverse and forward pulses until position of maximum progress is reached again. If switched to ON, in case of shaft encoder backward movement, the operator counts how many shaft encoder steps the shaft encoder moves backwards. When the shaft encoder moves forwards again, this number of shaft encoder steps (now forward direction) is not transmitted as external trigger signals. Only after the transportation belt is back to the place where the backward movement started, the shaft encoder steps (forward direction) are transmitted as external trigger signals again.

Parameter *FG_SHAFTENCODER_COMPENSATION_ENABLE* switched ON:

Figure 8.5. Shaft Encoder Compensation Enable = ON



In case the rollback compensation is disabled, the shaft encoder analyzer will only suppress reverse pulses but use all forward pulses. If switched to OFF, the operator simply doesn't transmit any trigger signals as long as the transportation belt moves backwards. As soon as the transport belt starts to move forwards again, the operator transmits the shaft encoder steps (forward direction) as trigger signals.

Parameter *FG_SHAFTENCODER_COMPENSATION_ENABLE* switched OFF:

Figure 8.6. Shaft Encoder Compensation Enable = OFF

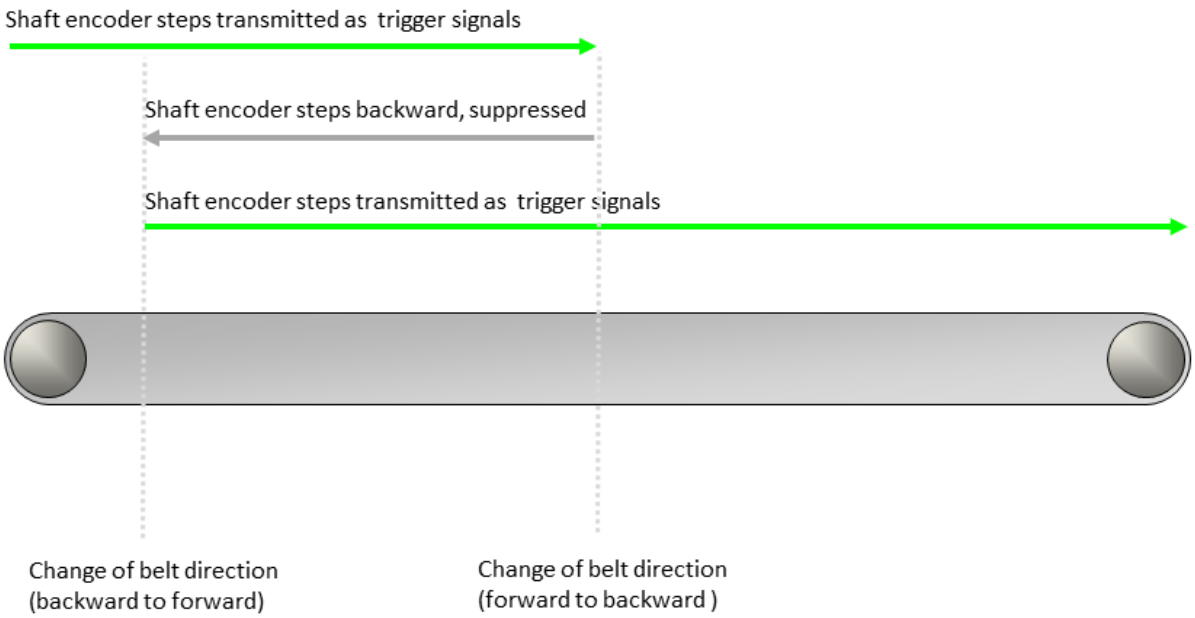


Table 8.13. Parameter properties of FG_SHAFTENCODER_COMPENSATION_ENABLE

Property	Value
Name	**FG_SHAFTENCODER_COMPENSATION_ENABLE**
Display Name	**Shaft Encoder Compensation Enable**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**FG_ON** On **FG_OFF** Off
Default value	**FG_ON**

Example 8.12. Usage of FG_SHAFTENCODER_COMPENSATION_ENABLE

```
int result = 0;
int value = FG_ON;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_SHAFTENCODER_COMPENSATION_ENABLE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_SHAFTENCODER_COMPENSATION_ENABLE, &value, 0, type)) < 0) {
    /* error handling */
}
```

8.4.6. FG_SHAFTENCODER_COMPENSATION_COUNT

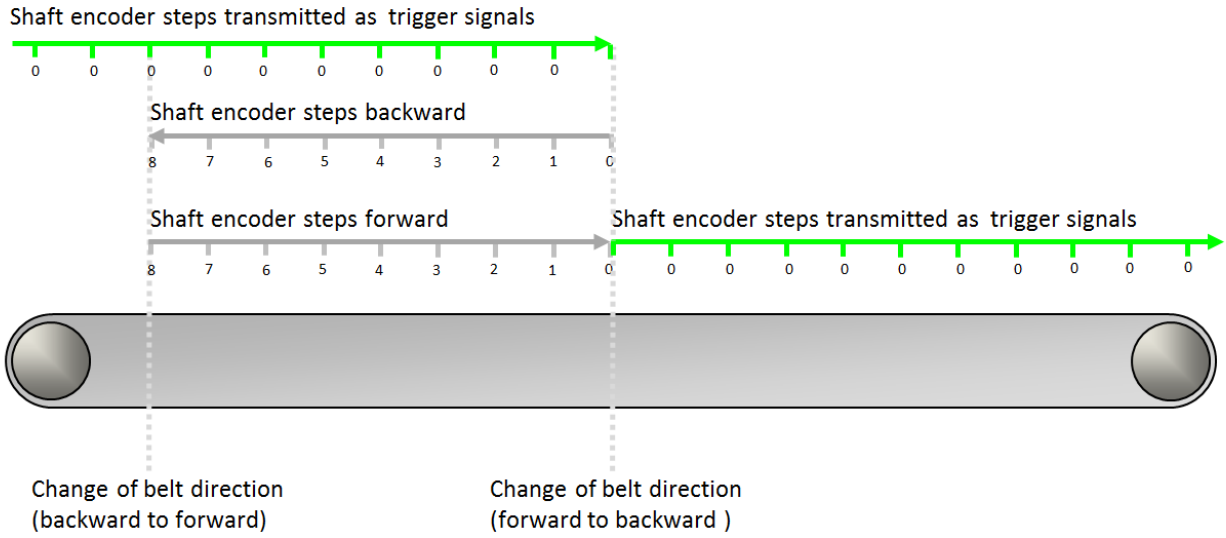
Using this parameter you can read and write the current shaft encoder rollback compensation counter. A compensation value zero indicates that currently no compensation is made. Therefore, you can reset the compensation by writing value zero to this parameter. Any other value will set a new compensation value. By knowing the distance / resolution for every encoder pulse, the compensation distance can be set. Concerning the shaft encoder find some more details in the introduction of Section 8.3, 'Line Trigger Input'.

It is based on a 20bit counter enabling a backward movement of up to 1048575 encoder pulses. An overflow of this value will not occur since it will skip all additional pulses for a compensation state of more than 1048575. By this the count of the rollback compensation is limited by 2 to the power of 20 pulses, what is enough for most applications in practice. As an example we could use a pretty high resolution of 20 pulses per mm, what is already sufficient for a maximum rollback distance of more than 50 meters.

Basic Conditions

If parameter *FG_SHAFTENCODER_COMPENSATION_ENABLE* is set to ON, an internal counter counts the shaft encoder steps the transportation belt moves backwards. This is necessary to be able to compensate the exact number of shaft encoder steps when the transportation belt starts moving forwards again:

Figure 8.7. Shaft Encoder Compensation Enable = ON



The internal counter counts forwards as long as the transportation belt moves backwards. (In figure 8.7, from 0 to 8.)

The internal counter counts backwards while the transportation belt moves forwards. (In figure 8.7, from 8 to 0.)

When the internal counter holds the value 0, the shaft encoder steps are transmitted as trigger signals.

The value the internal counter holds at a given moment is the value of parameter *FG_SHAFTENCODER_COMPENSATION_COUNT*. Only if this value is 0, encoder steps are transmitted as trigger signals. If the value of parameter *FG_SHAFTENCODER_COMPENSATION_COUNT* is not 0, the shaft encoder steps are not transmitted as trigger signals and the value keeps changing with every encoder step until it reaches the value 0 again.

Reading the Parameter

The parameter *FG_SHAFTENCODER_COMPENSATION_COUNT* is a read/write parameter. Therefore, at any given moment, you can always read out the value the counter holds at a given moment.

Defining an Offset

On the other hand, you can always modify the parameter value since you have write access during acquisition. If you need to define an offset to the standard encoder compensation, you can use this parameter to enter the number of steps you need the offset to be.

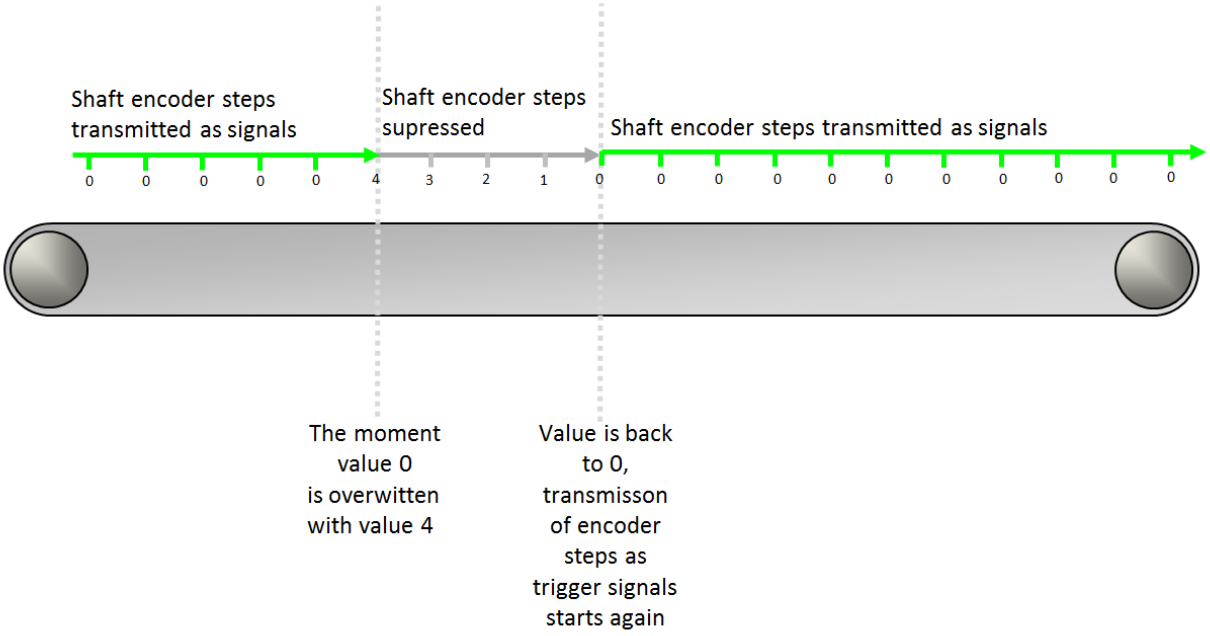
As soon as you enter a value for *FG_SHAFTENCODER_COMPENSATION_COUNT*, this value overwrites the value the parameter holds before.

In the following let's look at some examples for overwriting the current value of *FG_SHAFTENCODER_COMPENSATION_COUNT*:

Example 1:

The transportation belt is moving forward, the shaft encoder steps are transmitted as trigger signals, and the value of *FG_SHAFTENCODER_COMPENSATION_COUNT* is 0. Then, the value 0 of *FG_SHAFTENCODER_COMPENSATION_COUNT* is overwritten by value 4. Result: 4 shaft encoder steps are not transmitted as trigger signals.

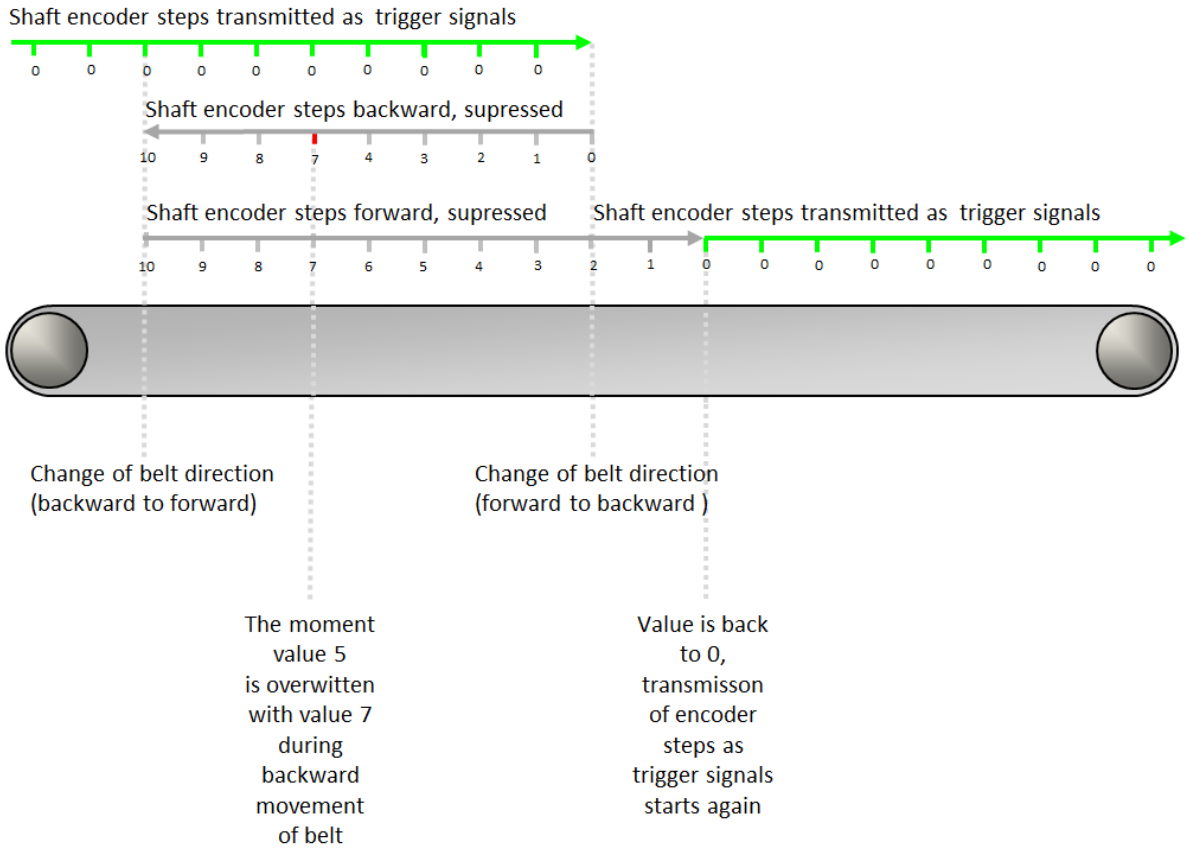
Figure 8.8. Shaft Encoder Compensation Count Example 1



Example 2:

The transportation belt is moving backward, the (backward) shaft encoder steps are suppressed, and the value of *FG_SHAFTENCODER_COMPENSATION_COUNT* is not 0. Then, during backward movement of the transportation belt, the value 5 of *FG_SHAFTENCODER_COMPENSATION_COUNT* is overwritten by value 7. Result: Offset of 2 shaft encoder steps.

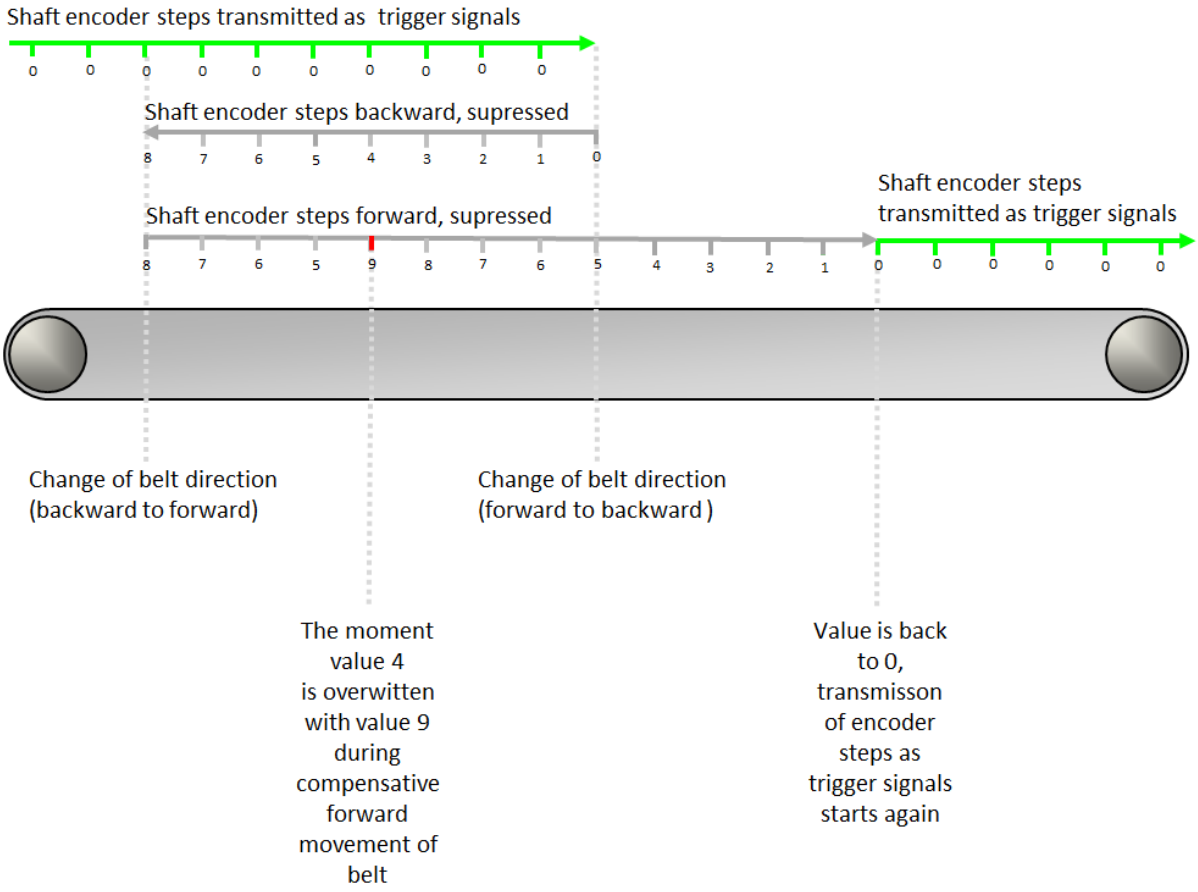
Figure 8.9. Shaft Encoder Compensation Count Example 2



Example 3:

The transportation belt is moving forward during compensation, the (forward) shaft encoder steps are suppressed, and the value of *FG_SHAFTENCODER_COMPENSATION_COUNT* is not 0. Then, during compensative forward movement of the transportation belt, the value 4 of *FG_SHAFTENCODER_COMPENSATION_COUNT* is overwritten with value 9. Result: Offset of 5 shaft encoder steps.

Figure 8.10. Shaft Encoder Compensation Count Example 3



Example 4:

The transportation belt is moving forward during compensation, the (forward) shaft encoder steps are suppressed, and the value of *FG_SHAFTENCODER_COMPENSATION_COUNT* is not 0. Then, during compensative forward movement of the transportation belt, the value 4 of *FG_SHAFTENCODER_COMPENSATION_COUNT* is overwritten with a smaller value, in our case with value 3. Result: Negative offset of -1 shaft encoder step.

Figure 8.11. Shaft Encoder Compensation Count Example 4

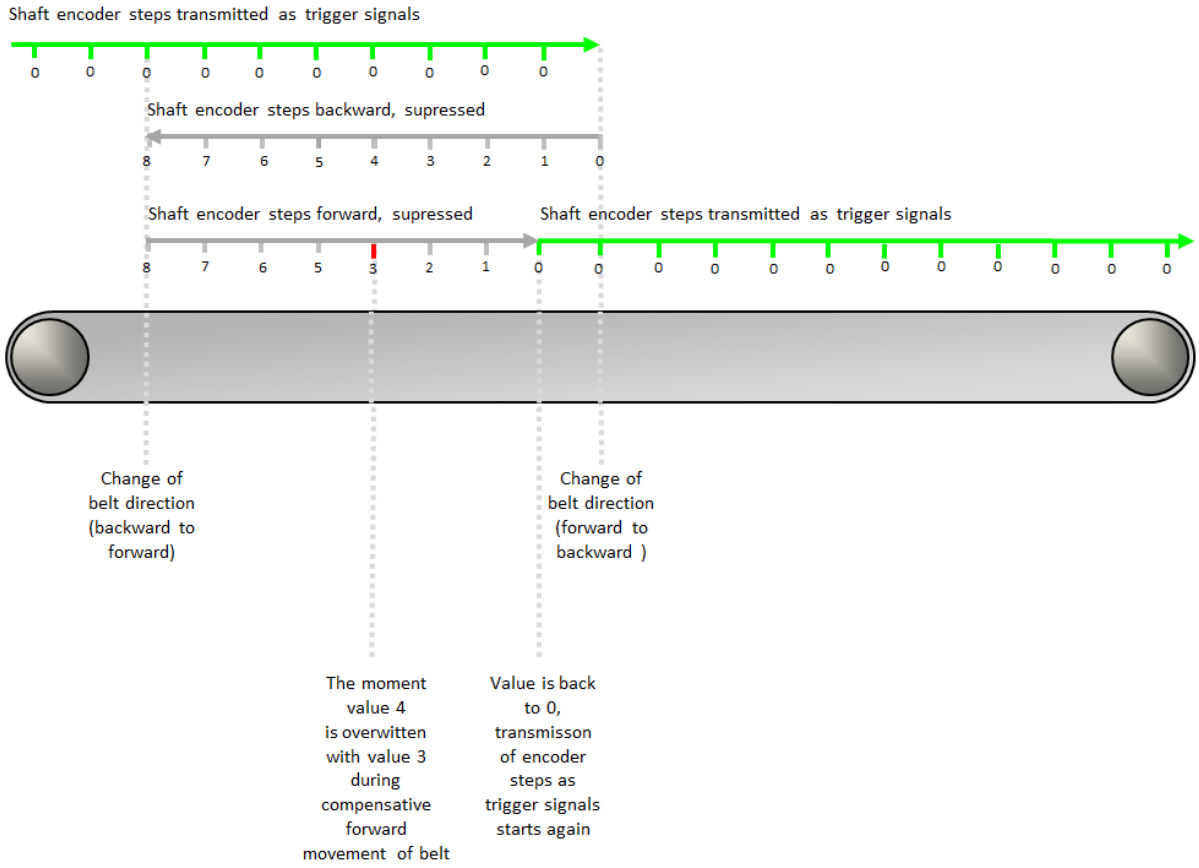


Table 8.14. Parameter properties of FG_SHAFTENCODER_COMPENSATION_COUNT

Property	Value
Name	**FG_SHAFTENCODER_COMPENSATION_COUNT**
Display Name	**Shaft Encoder Compensation Count**
Type	**Unsigned Integer**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum 0** **Maximum 1048575** **Stepsize 1**
Default value	**0**
Unit of measure	**pulses**

Example 8.13. Usage of FG_SHAFTENCODER_COMPENSATION_COUNT

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_setParameterWithType(fg, FG_SHAFTENCODER_COMPENSATION_COUNT, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_SHAFTENCODER_COMPENSATION_COUNT, &value, 0, type)) < 0) {
    /* error handling */
}
```

8.5. ExSync Output

This category includes parameters to specify and parameterize the generated ExSync output signals.

8.5.1. FG_LINEPERIODE

This parameter specifies the period of the ExSync signal. Therefore, it defines the line frequency when using the grabber controlled mode to trigger the connected camera. This period is of interest if a grabber controlled line trigger mode is used; more details for this can be found at *FG_LINETRIGGERMODE*. The line period is not allowed to be shorter than the minimum period - maximum line frequency - being supported by the camera, or in other words:

Please do not try to trigger the camera at a higher frequency than possible.

This maximum frequency is limited by the exposure time and the line scan sensor maximum speed. Please consider the camera manual for more details.

The following equations are mentioned in order to support the setup process if no period for *FG_LINEPERIODE* is mentioned:

- **Frequency**

  The period **T** is the duration of time of one cycle in a repeating event, so the period is the reciprocal of the frequency **f**.

  Equation 8.1. Frequency to Period

$$T = \frac{1}{f}$$

  Equation 8.2. Example: 17.6 kHz to Period

$$\begin{aligned} T &= \frac{1}{F} = \frac{1}{17.6kHz} = \frac{1}{17600Hz} \\ T &= 0.0000568s = 0.0568ms = 56.8\mu s \end{aligned}$$

- **Velocity and Pixel / mm**

  The period **T** is the duration of time of one cycle in a repeating event. At a velocity **v** and a given number **n** of pixels / mm together with the number **n** of pixels / mm being based on the resolution count **r** of the line scan sensor pixels and the width of view **w** in mm the following equations are valid.

  Equation 8.3. Velocity and Resolution to Period

$$\begin{aligned} n &= \frac{r}{w} \\ v &= \frac{distance}{time} \\ f &= v * n \\ T &= \frac{1}{f} \end{aligned}$$

Equation 8.4. Example: v = 53.4 m/min, r = 4096 pixels, w = 19.2 cm Wide Web to Period

$$
\begin{aligned}
n &= \frac{r}{w} = \frac{4096}{19.2cm} = \frac{4096}{192mm} = \frac{21.33}{mm} \\
v &= \frac{distance}{time} = \frac{53.4m}{min} = \frac{53.4m}{60s} = 0.89\frac{m}{s} \\
f &= v * n = 0.89\frac{m}{s} * \frac{21.33}{mm} = 890\frac{mm}{s} * \frac{21.33}{mm} \\
&= \frac{890 * 21.33}{s} = \frac{18983.7}{s} = 18983.7Hz = 18.9837kHz \\
T &= \frac{1}{f} \\
&= \frac{1}{18983.7Hz} = 52.68\mu s
\end{aligned}
$$

Table 8.15. Parameter properties of FG_LINEPERIODE

Property	Value
Name	**FG_LINEPERIODE**
Display Name	**Line Period**
Type	**Double**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum 0.2048** **Maximum 838.8576** **Stepsize 0.0032**
Default value	**200.0**
Unit of measure	**µs**

Example 8.14. Usage of FG_LINEPERIODE

```
int result = 0;
double value = 200.0;
const enum FgParamTypes type = FG_PARAM_TYPE_DOUBLE;

if ((result = Fg_setParameterWithType(fg, FG_LINEPERIODE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_LINEPERIODE, &value, 0, type)) < 0) {
    /* error handling */
}
```

8.5.2. FG_LINEEXPOSURE

This parameter specifies the pulse width of the ExSync signal, which can be used by many cameras to specify the exposure time. It is possible to adjust the exposure time via software, even while grabbing. The value is set in microseconds and may not exceed the period time of the ExSync *FG_LINEPERIODE*. In order to check the polarity simply increase this value and the resulting frame should become brighter. If this behaves in an opposite way check the polarity using *FG_EXSYNCPOLARITY*.

Table 8.16. Parameter properties of FG_LINEEXPOSURE

Property	Value
Name	**FG_LINEEXPOSURE**
Display Name	**Line Exposure**
Type	**Double**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum** 0.2048 **Maximum** 419.4272 **Stepsize** 0.0032
Default value	**19.0**
Unit of measure	**µs**

Example 8.15. Usage of FG_LINEEXPOSURE

```
int result = 0;
double value = 19.0;
const enum FgParamTypes type = FG_PARAM_TYPE_DOUBLE;

if ((result = Fg_setParameterWithType(fg, FG_LINEEXPOSURE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_LINEEXPOSURE, &value, 0, type)) < 0) {
    /* error handling */
}
```

8.5.3. FG_EXSYNCPOLARITY

The parameter adjusts the polarity of the ExSync signal generator. Use Low Active, if the camera opens the shutter on a falling edge, otherwise use High Active. For the mapping of the ExSync signals to the digital outputs check Chapter 7, '*Digital I/O*'.

Table 8.17. Parameter properties of FG_EXSYNCPOLARITY

Property	Value
Name	**FG_EXSYNCPOLARITY**
Display Name	**ExSync Polarity**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**FG_LOW** Low Active **FG_HIGH** High Active
Default value	**FG_HIGH**

Example 8.16. Usage of FG_EXSYNCPOLARITY

```
int result = 0;
int value = FG_HIGH;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_EXSYNCPOLARITY, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_EXSYNCPOLARITY, &value, 0, type)) < 0) {
    /* error handling */
}
```

8.5.4. FG_LINETRIGGERDELAY

This parameter specifies the delay between the generated ExSync and ExSync2 signals with respect to an external trigger input. Therefore, the ExSync2 signal is a delayed clone of the ExSync (polarity, period, etc. are the same as for ExSync). For the mapping of the ExSync signals to the digital outputs check Chapter 7, '*Digital I/O*'.

Please note that the line trigger delay needs to be less than the line trigger period. You might need to increase the line period first before increasing the line delay. This constraint also applies for external line trigger modes.

Table 8.18. Parameter properties of FG_LINETRIGGERDELAY

Property	Value
Name	**FG_LINETRIGGERDELAY**
Display Name	**Line Trigger Delay**
Type	**Double**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum 0.0** **Maximum 419.4272** **Stepsize 0.0032**
Default value	**0.0**
Unit of measure	**µs**

Example 8.17. Usage of FG_LINETRIGGERDELAY

```
int result = 0;
double value = 0.0;
const enum FgParamTypes type = FG_PARAM_TYPE_DOUBLE;

if ((result = Fg_setParameterWithType(fg, FG_LINETRIGGERDELAY, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_LINETRIGGERDELAY, &value, 0, type)) < 0) {
    /* error handling */
}
```

Chapter 9. Image Trigger / Flash

The image trigger for line-scan cameras is in charge to generate an internal signal called image gate. Lines sent by the camera are only accepted if this image gate is active = open. Therefore, with help of the Image Gate it is possible to define frames by grouping all lines that belong to the same image gate into one frame.

This AcquisitionApplets supports three distinct operation modes of the image trigger:

- Free run

  In free run mode the image gate basically remains active all time. Therefore, all lines sent by the camera are grabbed. Moreover, it cuts the input lines into frames of the height specified by parameter *FG_HEIGHT* of the display module. Also, offsets defined by *FG_YOFFSET* are covered and removed from the camera transfers for each image.

- Async Trigger

  For the external trigger mode of the image trigger, the image gate is inactive = closed until an external trigger signal activates the image gate for *FG_HEIGHT* + *FG_YOFFSET* lines. Therefore, for each external trigger event, the frame grabber records a frame of the specified height.

- Async Trigger Multi Buffer

  For the external trigger mode of the image trigger, the image gate is inactive = closed until an external trigger signal activates the image gate. In contrast to the **ASYNC_TRIGGER** mode, the gate is open for *FG_IMGTRIGGER_ASYNC_HEIGHT* lines while this image is split into smaller chunks of *FG_HEIGHT* lines. Therefore, for each external trigger event, the frame grabber records a frame of a large specified height and split the large image into smaller chunks. The purpose of the mode is to start processing in PC while the image is still recorded.

  The parameter value of *FG_YOFFSET* is without influence in this mode.

- Gated, Trigger

  For the external gated mode of the image trigger, the image gate is active as long as the external trigger source is active, but is becoming inactive when *FG_HEIGHT* + *FG_YOFFSET* lines have been grabbed. Therefore, during an external trigger phase the frame grabber records a frame with a height depending on the duration of active time of the external trigger signal, but is not exceeding an image height of *FG_HEIGHT* + *FG_YOFFSET* lines.

- Gated Multi Buffer, Triggered

  Equal to the 'Gated Trigger' mode, for the 'Gated Multi Buffer Trigger' the image gate is active as long as the external trigger source is active. In contrast, it does not limit the height to *FG_HEIGHT* lines. It will cut the image after *FG_HEIGHT* lines and start a new frame. Thus, for each gate, multiple frames are generated when a gate is active for more lines than defined by *FG_HEIGHT*.

  All images of a generated sequence will have a height of *FG_HEIGHT* lines. However, the last image of each sequence might have a lower number of lines in the image.

  To detect the last image of a sequence in your software. Parameter **FG_IMAGE_TAG** can be used. This parameter is of type unsigned 32 bit integer. The most significant bit i.e. bit 31 includes a flag which is set to one if the respective image is the last image of a multi buffer sequence.

```
uint32_t imageTag = 0;
int returnCode = Fg_getParameterEx(fg, FG_IMAGE_TAG, &imageTag, 0, pmem0, imageNumber);
bool isLastImageOfSequence = imageTagRAW >> 31;
```

  All other bits of parameter **FG_IMAGE_TAG** are fixed to value 0. The image tag parameter does not output the image number as available for older AcquisitionApplets.

Note that the value of parameter *FG_YOFFSET* is not considered if the 'Gated Multi Buffer Trigger' mode is used. An y-offset cannot be set in the applet.

9.1. FG_IMGTRIGGERMODE

Choose one of the image trigger modes described above. Please make sure that the operation mode of frame grabber and camera is the same.

Table 9.1. Parameter properties of FG_IMGTRIGGERMODE

Property	Value	
Name	**FG_IMGTRIGGERMODE**	
Display Name	**Image Trigger Mode**	
Type	**Enumeration**	
Access policy	**Read/Write**	
Storage policy	**Persistent**	
Allowed values	**FREE_RUN**	Free Run
	ASYNC_TRIGGER	Async External Trigger
	ASYNC_TRIGGER_MULTIFRAME	Async External Trigger Multiframe
	ASYNC_GATED	Async Gated Trigger
	ASYNC_GATED_MULTIFRAME	Async Gated Trigger Multiframe
Default value	**FREE_RUN**	

Example 9.1. Usage of FG_IMGTRIGGERMODE

```
int result = 0;
int value = FREE_RUN;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_IMGTRIGGERMODE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_IMGTRIGGERMODE, &value, 0, type)) < 0) {
    /* error handling */
}
```

9.2. FG_IMGTRIGGERON

The generation of image triggers can be switched on or off by use of this parameter. When the image trigger is disabled and the image trigger is not running in free-run mode, the image acquisition is terminated. If the image trigger is enabled, the acquisition will start immediately.

Table 9.2. Parameter properties of FG_IMGTRIGGERON

Property	Value	
Name	**FG_IMGTRIGGERON**	
Display Name	**Image Trigger On**	
Type	**Enumeration**	
Access policy	**Read/Write/Change**	
Storage policy	**Persistent**	
Allowed values	**FG_ON**	On
	FG_OFF	Off
Default value	**FG_ON**	

Example 9.2. Usage of FG_IMGTRIGGERON

```
int result = 0;
int value = FG_ON;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_IMGTRIGGERON, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_IMGTRIGGERON, &value, 0, type)) < 0) {
    /* error handling */
}
```

9.3. FG_FLASHON

To enable the flash output use this parameter.

For the mapping of the flash signal to the digital IO check Chapter 7, '*Digital I/O*'.

Table 9.3. Parameter properties of FG_FLASHON

Property	Value
Name	**FG_FLASHON**
Display Name	**Flash On**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**FG_ON** On **FG_OFF** Off
Default value	**FG_ON**

Example 9.3. Usage of FG_FLASHON

```
int result = 0;
int value = FG_ON;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_FLASHON, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_FLASHON, &value, 0, type)) < 0) {
    /* error handling */
}
```

9.4. FG_IMGTRIGGER_ASYNC_HEIGHT

This parameter only has influcence in the image trigger mode *FG_IMGTRIGGERMODE* Async Trigger Multi Frame **ASYNC_TRIGGER_MULTIFRAME**. The value is used to define the image height of the frame after the trigger pulse. Whereas parameter *FG_HEIGHT* defines the chunk height.

If the value of *FG_IMGTRIGGER_ASYNC_HEIGHT* is less than *FG_HEIGHT*, the frame is not split into multiple frames an will result in a smaller output frame.

Table 9.4. Parameter properties of FG_IMGTRIGGER_ASYNC_HEIGHT

Property	Value
Name	**FG_IMGTRIGGER_ASYNC_HEIGHT**
Display Name	**Image Trigger Async Height**
Type	**Unsigned Integer**
Access policy	**Read/Write**
Storage policy	**Persistent**
Allowed values	**Minimum 1** **Maximum 16777216** **Stepsize 1**
Default value	**1024**
Unit of measure	**lines**

Example 9.4. Usage of FG_IMGTRIGGER_ASYNC_HEIGHT

```
int result = 0;
unsigned int value = 1024;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_setParameterWithType(fg, FG_IMGTRIGGER_ASYNC_HEIGHT, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_IMGTRIGGER_ASYNC_HEIGHT, &value, 0, type)) < 0) {
    /* error handling */
}
```

9.5. FG_IMGTRIGGER_IS_BUSY

The image trigger is busy if the current requested frame from the camera has not been completely transferred to the grabber. This parameter can be used to check if the camera can accept a new software trigger pulse.

Table 9.5. Parameter properties of FG_IMGTRIGGER_IS_BUSY

Property	Value
Name	**FG_IMGTRIGGER_IS_BUSY**
Display Name	**Image Trigger is Busy**
Type	**Enumeration**
Access policy	**Read-Only**
Storage policy	**Transient**
Allowed values	**IS_BUSY** Busy **IS_NOT_BUSY** Not Busy

Example 9.5. Usage of FG_IMGTRIGGER_IS_BUSY

```
int result = 0;
int value = IS_NOT_BUSY;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_getParameterWithType(fg, FG_IMGTRIGGER_IS_BUSY, &value, 0, type)) < 0) {
    /* error handling */
}
```

9.6. Image Trigger Input

This category includes parameters to specify and control the image trigger inputs. The input can either be input pins of the frame grabber's trigger connector or trigger pulses generated by software register accesses.

9.6.1. FG_IMGTRIGGERINSRC

This parameter specifies the signal source, which is used to trigger the image acquisition gate. If a software image trigger has to be used select option **TRGINSOFTWARE**.

Table 9.6. Parameter properties of FG_IMGTRIGGERINSRC

Property	Value	
Name	**FG_IMGTRIGGERINSRC**	
Display Name	**Image Trigger Input Source**	
Type	**Enumeration**	
Access policy	**Read/Write/Change**	
Storage policy	**Persistent**	
Allowed values	**TRGINSRC_FRONT_GPI_0**	Trigger In Source Front GPI 0
	TRGINSRC_FRONT_GPI_1	Trigger In Source Front GPI 1
	TRGINSRC_FRONT_GPI_2	Trigger In Source Front GPI 2
	TRGINSRC_FRONT_GPI_3	Trigger In Source Front GPI 3
	TRGINSOFTWARE	Software Trigger
Default value	**TRGINSRC_FRONT_GPI_0**	

Example 9.6. Usage of FG_IMGTRIGGERINSRC

```
int result = 0;
int value = TRGINSRC_FRONT_GPI_0;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_IMGTRIGGERINSRC, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_IMGTRIGGERINSRC, &value, 0, type)) < 0) {
    /* error handling */
}
```

9.6.2. FG_IMGTRIGGERINPOLARITY

The parameter defines the polarity of the external input trigger signal.

Table 9.7. Parameter properties of FG_IMGTRIGGERINPOLARITY

Property	Value	
Name	**FG_IMGTRIGGERINPOLARITY**	
Display Name	**Image Trigger Input Polarity**	
Type	**Enumeration**	
Access policy	**Read/Write/Change**	
Storage policy	**Persistent**	
Allowed values	**HIGH_ON_ZERO_LOW**	Low Active
	HIGH_ON_ZERO_HIGH	High Active
Default value	**HIGH_ACTIVE**	

Example 9.7. Usage of FG_IMGTRIGGERINPOLARITY

```
int result = 0;
int value = HIGH_ACTIVE;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_IMGTRIGGERINPOLARITY, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_IMGTRIGGERINPOLARITY, &value, 0, type)) < 0) {
```

```
    /* error handling */
}
```

9.6.3. FG_IMGTRIGGERGATEDELAY

With this parameter, a delay of lines can be configured before the activation of the image gate. This delays the start of the image acquisition. The parameter y-offest (as in free run mode) rejects the first lines from the camera. Delay and y-offset seem to have the same effect, however the difference is, that y-offset doesn't affect the image gate, which is relevant while using the gated line trigger mode.

Table 9.8. Parameter properties of FG_IMGTRIGGERGATEDELAY

Property	Value
Name	**FG_IMGTRIGGERGATEDELAY**
Display Name	**Image Trigger Gate Delay**
Type	**Unsigned Integer**
Access policy	**Read/Write**
Storage policy	**Persistent**
Allowed values	**Minimum 0** **Maximum 65535** **Stepsize 1**
Default value	**0**
Unit of measure	**lines**

Example 9.8. Usage of FG_IMGTRIGGERGATEDELAY

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_setParameterWithType(fg, FG_IMGTRIGGERGATEDELAY, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_IMGTRIGGERGATEDELAY, &value, 0, type)) < 0) {
    /* error handling */
}
```

9.6.4. FG_IMGTRIGGERDEBOUNCING

This parameter specifies the debouncing time. This is the time for which the input image trigger signal must keep the same value to be detected as such. Fast signal changes within the debounce time will be filtered out.

Table 9.9. Parameter properties of FG_IMGTRIGGERDEBOUNCING

Property	Value
Name	**FG_IMGTRIGGERDEBOUNCING**
Display Name	**Image Trigger Debouncing**
Type	**Double**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum 0.0032** **Maximum 26.0** **Stepsize 0.0032**
Default value	**0.112**
Unit of measure	**µs**

Example 9.9. Usage of FG_IMGTRIGGERDEBOUNCING

```
int result = 0;
double value = 0.112;
const enum FgParamTypes type = FG_PARAM_TYPE_DOUBLE;

if ((result = Fg_setParameterWithType(fg, FG_IMGTRIGGERDEBOUNCING, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_IMGTRIGGERDEBOUNCING, &value, 0, type)) < 0) {
    /* error handling */
}
```

9.6.5. FG_STROBEPULSEDELAY

This parameter specifies the delay of the generated flash signal with respect to an external trigger input. Therefore, it is possible to synchronize the flash to the external trigger input. The delay is set in image line ticks.

Table 9.10. Parameter properties of FG_STROBEPULSEDELAY

Property	Value
Name	**FG_STROBEPULSEDELAY**
Display Name	**Strobe Pulse Delay**
Type	**Unsigned Integer**
Access policy	**Read/Write**
Storage policy	**Persistent**
Allowed values	**Minimum 0** **Maximum 65535** **Stepsize 1**
Default value	**0**
Unit of measure	**lines**

Example 9.10. Usage of FG_STROBEPULSEDELAY

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_setParameterWithType(fg, FG_STROBEPULSEDELAY, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_STROBEPULSEDELAY, &value, 0, type)) < 0) {
    /* error handling */
}
```

9.6.6. Flash

9.6.6.1. FG_FLASH_POLARITY

The polarity of the generated flash signal can be changed with this parameter. For the mapping of the flash signal to the digital outputs check Chapter 7, '*Digital I/O*'.

Table 9.11. Parameter properties of FG_FLASH_POLARITY

Property	Value
Name	**FG_FLASH_POLARITY**
Display Name	**Flash Polarity**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**FG_LOW** Low Active **FG_HIGH** High Active
Default value	**FG_HIGH**

Example 9.11. Usage of FG_FLASH_POLARITY

```
int result = 0;
int value = FG_HIGH;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_FLASH_POLARITY, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_FLASH_POLARITY, &value, 0, type)) < 0) {
    /* error handling */
}
```

9.6.7. Software Trigger

For the image trigger it is possible to use a software generated trigger signal to replace the external trigger input.

The software trigger control modules allows the to either generate a software trigger pulse or allows to set the state of the software trigger signal to generate a gate i.e. for gated image trigger mode.

To enable the software trigger set parameter *FG_IMGTRIGGERINSRC* to software trigger.

9.6.7.1. FG_SENDSOFTWARETRIGGER

A software trigger pulse can be sent by use of this parameter. Ensure to enable the software trigger by *FG_IMGTRIGGERINSRC*.

Table 9.12. Parameter properties of FG_SENDSOFTWARETRIGGER

Property	Value
Name	**FG_SENDSOFTWARETRIGGER**
Display Name	**Send Software Trigger**
Type	**Unsigned Integer**
Access policy	**Read/Write/Change**
Storage policy	**Transient**
Allowed values	**Minimum** 1 **Maximum** 1 **Stepsize** 1
Default value	**1**

Example 9.12. Usage of FG_SENDSOFTWARETRIGGER

```
int result = 0;
```

```
unsigned int value = 1;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_setParameterWithType(fg, FG_SENDSOFTWARETRIGGER, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_SENDSOFTWARETRIGGER, &value, 0, type)) < 0) {
    /* error handling */
}
```

9.6.7.2. FG_SETSOFTWARETRIGGER

The software trigger state can be set to zero = inactive = low or one = active = high. Ensure to enable the software trigger by *FG_IMGTRIGGERINSRC*.

Table 9.13. Parameter properties of FG_SETSOFTWARETRIGGER

Property	Value
Name	**FG_SETSOFTWARETRIGGER**
Display Name	**Set Software Trigger**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**FG_LOW** Low Active **FG_HIGH** High Active
Default value	

Example 9.13. Usage of FG_SETSOFTWARETRIGGER

```
int result = 0;
int value = FG_ZERO;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_SETSOFTWARETRIGGER, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_SETSOFTWARETRIGGER, &value, 0, type)) < 0) {
    /* error handling */
}
```

Chapter 10. Signal Analyzer

The signal analyzer module computes some information on a signal source. These are

- Pulse Count
- Period (current, min, max)
- Difference between two pulse counters

The module is used to detect unexpected behaviors of the trigger system. For example a bouncing encode signal resulting in overtriggering of the camera. Another example is the detection of trigger lost signals or corrupted camera data which can result in extra lines.

Simply select the analyzer source signal and polarity. The measurement values can be obtained using read-only parameters. All measurements can be cleared synchronously.

Note that the module is available only once for the applet. All cameras share the same module. The camera/DMA index in the setParameter and getParameter functions has no influence.

10.1. FG_SIGNAL_ANALYZER_0_SOURCE et al.

Note

This description applies also to the following parameters: FG_SIGNAL_ANALYZER_1_SOURCE

Select the source signal for the trigger analyzer. For further explanation of the available sources see Chapter 7, '*Digital I/O*'. In addition, the line/frame start/end pulses can be used as signal sources, too.

Table 10.1. Parameter properties of FG_SIGNAL_ANALYZER_0_SOURCE

Property	Value	
Name	**FG_SIGNAL_ANALYZER_0_SOURCE**	
Display Name	**Signal Analyzer 0 Source**	
Type	**Enumeration**	
Access policy	**Read/Write/Change**	
Storage policy	**Persistent**	
Allowed values	**GND**	GND
	VCC	VCC
	FG_SIGNAL_CAM0_EXSYNC	Signal Exsync
	FG_SIGNAL_CAM0_EXSYNC2	Signal Exsync2
	FG_SIGNAL_CAM0_FLASH	Signal Flash
	FG_SIGNAL_CAM0_LVAL	Signal Line Valid
	FG_SIGNAL_CAM0_FVAL	Signal Frame Valid
	FG_SIGNAL_CAM0_LINE_START	Signal Line Start
	FG_SIGNAL_CAM0_LINE_END	Cam0 Line Transfer End
	FG_SIGNAL_CAM0_FRAME_START	Signal Frame Start
	FG_SIGNAL_CAM0_FRAME_END	Signal Frame End
	FG_SIGNAL_FRONT_GPI_0	Signal Front GPI 0
	FG_SIGNAL_FRONT_GPI_1	Signal Front GPI 1
Default value	**FG_SIGNAL_CAM0_EXSYNC**	

Example 10.1. Usage of FG_SIGNAL_ANALYZER_0_SOURCE

```
int result = 0;
```

```
int value = FG_SIGNAL_CAM0_EXSYNC;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_SIGNAL_ANALYZER_0_SOURCE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_SIGNAL_ANALYZER_0_SOURCE, &value, 0, type)) < 0) {
    /* error handling */
}
```

10.2. FG_SIGNAL_ANALYZER_0_POLARITY et al.

Note

This description applies also to the following parameters: FG_SIGNAL_ANALYZER_1_POLARITY

Select the polarity for the signal analyzer of the selected source. With this parameter you can invert the signal. The signal analyzer module will only measure on rising edges.

Table 10.2. Parameter properties of FG_SIGNAL_ANALYZER_0_POLARITY

Property	Value
Name	**FG_SIGNAL_ANALYZER_0_POLARITY**
Display Name	**Signal Analyzer 0 Polarity**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**FG_LOW** Low Active **FG_HIGH** High Active
Default value	**FG_HIGH**

Example 10.2. Usage of FG_SIGNAL_ANALYZER_0_POLARITY

```
int result = 0;
int value = FG_HIGH;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_SIGNAL_ANALYZER_0_POLARITY, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_SIGNAL_ANALYZER_0_POLARITY, &value, 0, type)) < 0) {
    /* error handling */
}
```

10.3. FG_SIGNAL_ANALYZER_0_PERIOD_CURRENT et al.

Note

This description applies also to the following parameters: FG_SIGNAL_ANALYZER_1_PERIOD_CURRENT

This read-only parameter returns the last measured period of the selected signal source. Keep in mind that the module requires two rising edges to obtain a measurement result. Selecting a new source or changing the acquisition states can result in very long periods.

Table 10.3. Parameter properties of FG_SIGNAL_ANALYZER_0_PERIOD_CURRENT

Property	Value
Name	**FG_SIGNAL_ANALYZER_0_PERIOD_CURRENT**
Display Name	**Signal Analyzer 0 Current Period**
Type	**Double**
Access policy	**Read-Only**
Storage policy	**Persistent**
Allowed values	**Minimum 0.0032** **Maximum 1.3743895344E7** **Stepsize 0.0032**
Unit of measure	**µs**

Example 10.3. Usage of FG_SIGNAL_ANALYZER_0_PERIOD_CURRENT

```
int result = 0;
double value = 0.0;
const enum FgParamTypes type = FG_PARAM_TYPE_DOUBLE;

if ((result = Fg_getParameterWithType(fg, FG_SIGNAL_ANALYZER_0_PERIOD_CURRENT, &value, 0, type)) < 0) {
    /* error handling */
}
```

10.4. FG_SIGNAL_ANALYZER_0_PERIOD_MAX et al.

Note

This description applies also to the following parameters: FG_SIGNAL_ANALYZER_1_PERIOD_MAX

This read-only parameter returns the maximum measured period after the last reset. Keep in mind that selecting a new source or changing the acquisition states can result in very long periods.

Table 10.4. Parameter properties of FG_SIGNAL_ANALYZER_0_PERIOD_MAX

Property	Value
Name	**FG_SIGNAL_ANALYZER_0_PERIOD_MAX**
Display Name	**Signal Analyzer 0 Max Period**
Type	**Double**
Access policy	**Read-Only**
Storage policy	**Persistent**
Allowed values	**Minimum 0.0032** **Maximum 1.3743895344E7** **Stepsize 0.0032**
Unit of measure	**µs**

Example 10.4. Usage of FG_SIGNAL_ANALYZER_0_PERIOD_MAX

```
int result = 0;
double value = 0.0;
const enum FgParamTypes type = FG_PARAM_TYPE_DOUBLE;

if ((result = Fg_getParameterWithType(fg, FG_SIGNAL_ANALYZER_0_PERIOD_MAX, &value, 0, type)) < 0) {
    /* error handling */
}
```

10.5. FG_SIGNAL_ANALYZER_0_PERIOD_MIN et al.

Note

This description applies also to the following parameters: FG_SIGNAL_ANALYZER_1_PERIOD_MIN

This read-only parameter returns the minimum measured period after the last reset.

Table 10.5. Parameter properties of FG_SIGNAL_ANALYZER_0_PERIOD_MIN

Property	Value
Name	**FG_SIGNAL_ANALYZER_0_PERIOD_MIN**
Display Name	**Signal Analyzer 0 Min Period**
Type	**Double**
Access policy	**Read-Only**
Storage policy	**Persistent**
Allowed values	**Minimum 0.0032** **Maximum 1.3743895344E7** **Stepsize 0.0032**
Unit of measure	**µs**

Example 10.5. Usage of FG_SIGNAL_ANALYZER_0_PERIOD_MIN

```
int result = 0;
double value = 0.0;
const enum FgParamTypes type = FG_PARAM_TYPE_DOUBLE;

if ((result = Fg_getParameterWithType(fg, FG_SIGNAL_ANALYZER_0_PERIOD_MIN, &value, 0, type)) < 0) {
    /* error handling */
}
```

10.6. FG_SIGNAL_ANALYZER_0_PULSE_COUNT et al.

Note

This description applies also to the following parameters: FG_SIGNAL_ANALYZER_1_PULSE_COUNT

Returns the counter value of the selected source. For each rising edge the counter is increased. This, after the first pulse, the counter value will be one. On counter overflow, it will start from 0 again.

Table 10.6. Parameter properties of FG_SIGNAL_ANALYZER_0_PULSE_COUNT

Property	Value
Name	**FG_SIGNAL_ANALYZER_0_PULSE_COUNT**
Display Name	**Signal Analyzer 0 Pulse Count**
Type	**Unsigned Integer**
Access policy	**Read-Only**
Storage policy	**Persistent**
Allowed values	**Minimum 0** **Maximum 4294967295** **Stepsize 1**
Unit of measure	**pulses**

Example 10.6. Usage of FG_SIGNAL_ANALYZER_0_PULSE_COUNT

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;
```

```
if ((result = Fg_getParameterWithType(fg, FG_SIGNAL_ANALYZER_0_PULSE_COUNT, &value, 0, type)) < 0) {
    /* error handling */
}
```

10.7. FG_SIGNAL_ANALYZER_PULSE_COUNT_DIFFERENCE

Use this read only parameter to check the difference of the signal analyzer 0 and 1 pulse counter values (Analyzer 0 - Analyzer 1 value). This can be used to check for trigger lost signals if analyzer 0 will count the exsyc pulses and analyzer 1 the returned camera lines. In this case the difference is between 0 and 1 for single line cameras with no extra delay. If the difference exceeds 1, the camera did not return a line for all trigger pulses i.e. a trigger is lost or ignored due to overtriggering. If the difference is less than 0 an additional camera line was generated and received by the frame grabber. The reason for this can be a noisy trigger cable which added extra spikes or a corrupted data transfer which split the data into several parts.

Table 10.7. Parameter properties of FG_SIGNAL_ANALYZER_PULSE_COUNT_DIFFERENCE

Property	Value
Name	**FG_SIGNAL_ANALYZER_PULSE_COUNT_DIFFERENCE**
Display Name	**Signal Analyzer Pulse Count Difference**
Type	**Signed Integer (64 Bit)**
Access policy	**Read-Only**
Storage policy	**Persistent**
Allowed values	**Minimum -4294967296** **Maximum 4294967295** **Stepsize 1**
Unit of measure	**pulses**

Example 10.7. Usage of FG_SIGNAL_ANALYZER_PULSE_COUNT_DIFFERENCE

```
int result = 0;
int64_t value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_INT64_T;

if ((result = Fg_getParameterWithType(fg, FG_SIGNAL_ANALYZER_PULSE_COUNT_DIFFERENCE, &value, 0, type)) < 0) {
    /* error handling */
}
```

10.8. FG_SIGNAL_ANALYZER_CLEAR

To clear all signal analyzer measurement results and counters use this parameter. All counters will be reset synchronously and are ready to restart immediately.

Table 10.8. Parameter properties of FG_SIGNAL_ANALYZER_CLEAR

Property	Value
Name	**FG_SIGNAL_ANALYZER_CLEAR**
Display Name	**Signal Analyzer Clear**
Type	**Unsigned Integer**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum 1** **Maximum 1** **Stepsize 1**
Default value	**1**

Example 10.8. Usage of FG_SIGNAL_ANALYZER_CLEAR

```
int result = 0;
unsigned int value = 1;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_setParameterWithType(fg, FG_SIGNAL_ANALYZER_CLEAR, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_SIGNAL_ANALYZER_CLEAR, &value, 0, type)) < 0) {
    /* error handling */
}
```

Chapter 11. Overflow

The applet processes image data as fast as possible. Any image data sent by the camera is immediately processed and sent to the PC. The latency is minimal. In general, only one concurrent image line is stored and processed in the interface card. However, the transfer bandwidth to the PC via DMA channel can vary caused by interrupts, other hardware and the current CPU load. Furthermore, if operated in **selective mode**, it is possible to queue buffer slower than the camera offers new images and therefore generate an overflow condition on the frame grabber. Also, the camera frame rate can vary due to an fluctuating trigger. For these cases, the applet is equipped with a memory to buffer the input frames. The fill level of the buffer can be obtained by reading from parameter *FG_FILLLEVEL*.

In normal operation conditions the buffer will always remain almost empty. For fluctuating camera bandwidths or for short and fast acquisitions, the buffer can easily fill up quickly. Of course, the input bandwidth must not exceed the maximum bandwidth of the applet. Check Section 1.2, 'Bandwidth' for more information.

If the buffer's fill level reaches 100%, the applet is in overflow condition, as no more data can be buffered and camera data will be discarded. This can result in two different behaviors:

- Corrupted Frames:

  The transfer of a current frame is interrupted by an overflow. This means, the first pixels or lines of the frame were transfered into the buffer, but not the full frame. The output of the applet i.e. the DMA transfer will be shorter. The output image will not have it's full height. These images will be marked incomplete in the **FG_IMAGE_TAG** (bit 30 is set to '1').

- Lost Frames:

  A full camera frame was discarded due to a full buffer memory. No DMA transfer will exist for the discarded frame. This means the number of applet output images can differ from the number of applet input images.

The buffer overflow threshold *FG_OVERFLOW_ON_THRESHOLD* and *FG_OVERFLOW_ON_SYNC_THRESHOLD* default ensures that under normal conditions frames can be completed or will be fully dropped so that corrupted frames are avoide

A way to detect the overflows is to read parameter *FG_OVERFLOW* or check for event *FG_OVERFLOW_CAM0*. Reading from the parameter will provide information about an overflow condition. As soon as the parameter is read, it will reset. Using the parameter an overflow condition can be detect, but it is not possible to obtain the exact image number and the moment. For this, the overflow event can be used.

11.1. FG_FILLLEVEL

The fill-level of the interface card buffers used in this applet can be read-out by use of this parameter. The value allows to check if the mean input bandwidth of the camera is to high to be processed with the applet.

Table 11.1. Parameter properties of FG_FILLLEVEL

Property	Value
Name	**FG_FILLLEVEL**
Display Name	**Fill Level**
Type	**Unsigned Integer**
Access policy	**Read-Only**
Storage policy	**Transient**
Allowed values	**Minimum 0** **Maximum 100** **Stepsize 1**
Unit of measure	**%**

Example 11.1. Usage of FG_FILLLEVEL

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_getParameterWithType(fg, FG_FILLLEVEL, &value, 0, type)) < 0) {
    /* error handling */
}
```

11.2. FG_OVERFLOW

If the applet runs into overflow, a value "1" can be read by the use of this parameter. Note that an overflow results in loss of images. To avoid overflows reduce the mean input bandwidth.

The parameter is reset at each readout cycle. The program microDisplayX will continuously poll the value, thus the occurrence of an overflow might not be visible in microDisplayX.

A more effective and robust way is to detect overflows is the use of the event system.

Table 11.2. Parameter properties of FG_OVERFLOW

Property	Value
Name	**FG_OVERFLOW**
Display Name	**Buffer overflow**
Type	**Unsigned Integer**
Access policy	**Read-Only**
Storage policy	**Transient**
Allowed values	**Minimum 0** **Maximum 1** **Stepsize 1**

Example 11.2. Usage of FG_OVERFLOW

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_getParameterWithType(fg, FG_OVERFLOW, &value, 0, type)) < 0) {
    /* error handling */
}
```

11.3. FG_OVERFLOW_OFF_THRESHOLD

The Overflow state will be deactivated once the buffer Fillevel (*FG_FILLLEVEL*) will fall below this value. As long as the applet remains in overflow state all images arriving will be discarded. This will result in Overflow events with a set "lost" flag.

Table 11.3. Parameter properties of FG_OVERFLOW_OFF_THRESHOLD

Property	Value
Name	**FG_OVERFLOW_OFF_THRESHOLD**
Display Name	**Overflow Off Threshold**
Type	**Double**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum 0.0** **Maximum 100.0** **Stepsize 0.5**
Default value	**50.0**

Example 11.3. Usage of FG_OVERFLOW_OFF_THRESHOLD

```
int result = 0;
double value = 50.0;
const enum FgParamTypes type = FG_PARAM_TYPE_DOUBLE;

if ((result = Fg_setParameterWithType(fg, FG_OVERFLOW_OFF_THRESHOLD, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_OVERFLOW_OFF_THRESHOLD, &value, 0, type)) < 0) {
    /* error handling */
}
```

11.4. FG_OVERFLOW_ON_THRESHOLD

The applet will enter Overflow state once the buffer Fillevel exceeds this filllevel (*FG_FILLLEVEL*). If the overflow state is active images will be stopped imidiately. This may lead to an incomplete frame. Incomplete frames are marked incomplete in the image Tag and an overflow event can be generated.

Table 11.4. Parameter properties of FG_OVERFLOW_ON_THRESHOLD

Property	Value
Name	**FG_OVERFLOW_ON_THRESHOLD**
Display Name	**Overflow On Threshold**
Type	**Double**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum 0.0** **Maximum 100.0** **Stepsize 0.5**
Default value	**99.5**

Example 11.4. Usage of FG_OVERFLOW_ON_THRESHOLD

```
int result = 0;
double value = 99.5;
const enum FgParamTypes type = FG_PARAM_TYPE_DOUBLE;

if ((result = Fg_setParameterWithType(fg, FG_OVERFLOW_ON_THRESHOLD, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_OVERFLOW_ON_THRESHOLD, &value, 0, type)) < 0) {
    /* error handling */
}
```

11.5. FG_OVERFLOW_ON_SYNC_THRESHOLD

The applet will enter Overflow state once the buffer fillevel (*FG_FILLLEVEL*) exceeds this filllevel and the currently arriving frame is stored to the buffer. If the applet remains in overflow state frames might be droped. If the buffer falls below this fillevel frames are accepted again. There is no hysteresis for this threshold.

Table 11.5. Parameter properties of FG_OVERFLOW_ON_SYNC_THRESHOLD

Property	Value
Name	**FG_OVERFLOW_ON_SYNC_THRESHOLD**
Display Name	**Overflow Sync On Threshold**
Type	**Double**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum 0.0** **Maximum 100.0** **Stepsize 0.5**
Default value	**80.0**

Example 11.5. Usage of FG_OVERFLOW_ON_SYNC_THRESHOLD

```
int result = 0;
double value = 80.0;
const enum FgParamTypes type = FG_PARAM_TYPE_DOUBLE;

if ((result = Fg_setParameterWithType(fg, FG_OVERFLOW_ON_SYNC_THRESHOLD, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_OVERFLOW_ON_SYNC_THRESHOLD, &value, 0, type)) < 0) {
    /* error handling */
}
```

11.6. FG_OVERFLOW_EVENT_SELECT

The *FG_OVERFLOW_CAM0* Event. Allows to generate events if one of the following conditions is meet.

Table 11.6. Event select for *FG_OVERFLOW_CAM0*

Value	Description
FG_OVERFLOW_EVENT_INCOMPLETE	Each incomplete frame will generate an Event containing the information that the frame is incomplete and the frameID
FG_OVERFLOW_EVENT_LOST	Each lost frame will generate an Event containing the information that the frame is lost and the frameID
FG_OVERFLOW_EVENT_INCOMPLETE_LOST	Each lost or incomplete frame will generate an Event containing the information that the frame is lost/ incomplete and the frameID
FG_OVERFLOW_EVENT_OK	Each correct frame will generate an Event containing the information that the frame is transfered correct and the frameID of the frame
FG_OVERFLOW_EVENT_OK_INCOMPLETE	Each incomplete or correct frame will generate an Event containing the information that the frame is correct or incomplete and the frameID
FG_OVERFLOW_EVENT_OK_LOST	Each lost or correct frame will generate an Event containing the information that the frame is correct or lost and the frameID
FG_OVERFLOW_EVENT_ALL	Each frame will generate an Event containing the status(lost, incomplete or correct) of the frame and the frameID

Table 11.7. Parameter properties of FG_OVERFLOW_EVENT_SELECT

Property	Value
Name	**FG_OVERFLOW_EVENT_SELECT**
Display Name	**Overflow Event Select**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**FG_OVERFLOW_EVENT_INCOMPLETE** Incomplete **FG_OVERFLOW_EVENT_LOST** Lost **FG_OVERFLOW_EVENT_INCOMPLETE_LOST** Incomplete Lost **FG_OVERFLOW_EVENT_OK** OK **FG_OVERFLOW_EVENT_OK_INCOMPLETE** Incomplete OK **FG_OVERFLOW_EVENT_OK_LOST** Lost OK **FG_OVERFLOW_EVENT_ALL** All
Default value	**FG_OVERFLOW_EVENT_INCOMPLETE_LOST**

Example 11.6. Usage of FG_OVERFLOW_EVENT_SELECT

```
int result = 0;
int value = FG_OVERFLOW_EVENT_INCOMPLETE_LOST;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_OVERFLOW_EVENT_SELECT, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_OVERFLOW_EVENT_SELECT, &value, 0, type)) < 0) {
    /* error handling */
}
```

11.7. Events

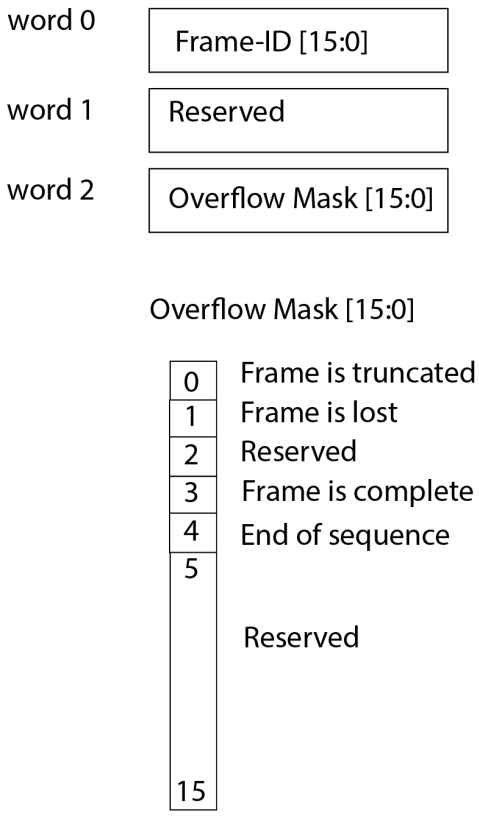
In programming or runtime environments, a callback function is a piece of executable code that is passed as an argument, which is expected to call back (execute) exactly that time an event is triggered. This applet can generate some software callback events based on the memory overflow condition as explained in the following section. These events are not related to a special camera functionality. Other event sources are described in additional sections of this document.

The Basler Framegrabber SDK and pylon SDK via GenTL enables an application to get these event notifications about certain state changes at the data flow from camera to RAM and the image and trigger processing as well. Please consult the Basler Framegrabber SDK, pylon SDK or GenTL documentation for more details concerning the implementation of this functionality.

11.7.1. FG_OVERFLOW_CAM0

Overflow events are generated for each truncated, lost or complete frame. The selection can be done using *FG_OVERFLOW_EVENT_SELECT*. The overflow event contains data, namely the type of overflow, the image number and the timestamp. The following figure illustrates the event data. Data is contained in a 64-bit data packet. The first 16 bits contain the frame-ID from the camera. Bits 32 to 47 provide an overflow mask.

Figure 11.1. Illustration of Overflow Data Packet



Note that the frame-ID is taken from the camera stream. See Section 1.5, 'Frame ID' for more information. The frame-ID is a 16-bit value. If its maximum is reached, the frame-ID starts at zero again. If the **frame truncated** flag is set, the frame with the frame-ID in the event is truncated i.e. it doesn't have its full length but is still transferred via DMA channel. If the **frame lost** flag is set, the frame with the frame-ID in the event was fully discarded. No DMA transfer exists for this frame. The **truncated frame** flag and the **frame lost** flag never occur for the same event.

Chapter 12. Image Selector

The Image Selector allows the user to cut out a period of *p* images from the image stream and select a particular image *n* from it.

The following example will explain the settings of *p* and *n* which represent the frame grabber parameters *FG_IMG_SELECT_PERIOD* and *FG_IMG_SELECT*. Suppose two frame grabbers being connected to a camera signal multiplexer, providing all camera images to both devices. Grabber 0 is required to process all even frames, while grabber 1 is required to process all odd frames. The settings will then be:

1. Grabber 0:
   - *FG_IMG_SELECT_PERIOD* = 2

     FG_IMG_SELECT = 0
2. Grabber 1:
   - *FG_IMG_SELECT_PERIOD* = 2

     FG_IMG_SELECT = 1

Ensure that both grabbers are used synchronously. This is possible with a triggered camera. To do so, initialize and configure both frame grabbers. Configure the camera for external trigger and the trigger system of master grabber which is directly connected to the camera.

12.1. FG_IMG_SELECT_PERIOD

This parameter specifies the period length *p*. The parameter can be changed at any time. However, changing during acquisition can result in an asynchronous switching which will result in the loss of a synchronous grabbing. It is recommended to change the parameter only when the acquisition is stopped.

The parameter's value has to be greater than *FG_IMG_SELECT*.

Table 12.1. Parameter properties of FG_IMG_SELECT_PERIOD

Property	Value
Name	**FG_IMG_SELECT_PERIOD**
Display Name	**Image Select Period**
Type	**Unsigned Integer**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum 1** **Maximum 256** **Stepsize 1**
Default value	**1**
Unit of measure	**image**

Example 12.1. Usage of FG_IMG_SELECT_PERIOD

```
int result = 0;
unsigned int value = 1;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_setParameterWithType(fg, FG_IMG_SELECT_PERIOD, &value, 0, type)) < 0) {
    /* error handling */
}
```

```
if ((result = Fg_getParameterWithType(fg, FG_IMG_SELECT_PERIOD, &value, 0, type)) < 0) {
    /* error handling */
}
```

12.2. FG_IMG_SELECT

The parameter *FG_IMG_SELECT* specifies a particular image from the image set defined by *FG_IMG_SELECT_PERIOD*. This parameter can be changed at any time. However, changing during acquisition can result in an asynchronous switching which will result in the loss of a synchronous grabbing. It is recommended to change the parameter only when the acquisition is stopped.

The parameter's value has to be less than *FG_IMG_SELECT_PERIOD*.

Table 12.2. Parameter properties of FG_IMG_SELECT

Property	Value
Name	**FG_IMG_SELECT**
Display Name	**Image Select**
Type	**Unsigned Integer**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum 0** **Maximum 255** **Stepsize 1**
Default value	**0**
Unit of measure	**image**

Example 12.2. Usage of FG_IMG_SELECT

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_setParameterWithType(fg, FG_IMG_SELECT, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_IMG_SELECT, &value, 0, type)) < 0) {
    /* error handling */
}
```

Chapter 13. White Balance

The applet enables a spectral adaptation of the image to the lighting situation of the application. The color values for the red, green and blue components can be individually enhanced or reduced by a scaling factor to adjust the spectral sensibility of the camera sensor.

13.1. FG_SCALINGFACTOR_GREEN

Table 13.1. Parameter properties of FG_SCALINGFACTOR_GREEN

Property	Value
Name	**FG_SCALINGFACTOR_GREEN**
Display Name	**Scaling Factor Green**
Type	**Double**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum 0.0** **Maximum 3.9990234375** **Stepsize 9.765625E-4**
Default value	**1.0**

Example 13.1. Usage of FG_SCALINGFACTOR_GREEN

```
int result = 0;
double value = 1.0;
const enum FgParamTypes type = FG_PARAM_TYPE_DOUBLE;

if ((result = Fg_setParameterWithType(fg, FG_SCALINGFACTOR_GREEN, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_SCALINGFACTOR_GREEN, &value, 0, type)) < 0) {
    /* error handling */
}
```

13.2. FG_SCALINGFACTOR_RED

Table 13.2. Parameter properties of FG_SCALINGFACTOR_RED

Property	Value
Name	**FG_SCALINGFACTOR_RED**
Display Name	**Scaling Factor Red**
Type	**Double**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum 0.0** **Maximum 3.9990234375** **Stepsize 9.765625E-4**
Default value	**1.0**

Example 13.2. Usage of FG_SCALINGFACTOR_RED

```
int result = 0;
double value = 1.0;
```

```
const enum FgParamTypes type = FG_PARAM_TYPE_DOUBLE;

if ((result = Fg_setParameterWithType(fg, FG_SCALINGFACTOR_RED, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_SCALINGFACTOR_RED, &value, 0, type)) < 0) {
    /* error handling */
}
```

13.3. FG_SCALINGFACTOR_BLUE

Table 13.3. Parameter properties of FG_SCALINGFACTOR_BLUE

Property	Value
Name	**FG_SCALINGFACTOR_BLUE**
Display Name	**Scaling Factor Blue**
Type	**Double**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum 0.0** **Maximum 3.9990234375** **Stepsize 9.765625E-4**
Default value	**1.0**

Example 13.3. Usage of FG_SCALINGFACTOR_BLUE

```
int result = 0;
double value = 1.0;
const enum FgParamTypes type = FG_PARAM_TYPE_DOUBLE;

if ((result = Fg_setParameterWithType(fg, FG_SCALINGFACTOR_BLUE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_SCALINGFACTOR_BLUE, &value, 0, type)) < 0) {
    /* error handling */
}
```

Chapter 14. Color Converter

The color converter module is used to convert the input pixel format to an output pixel format. The conversion is performed post to the Bayer de-mosicing and just before the lookup table.

This applet can perform the following conversions.

Table 14.1. Color Conversion

Input Format	Mono	RGB	Bayer	YCbCr
Output Format				
Mono	yes	yes	yes	N/A
RGB	yes	yes	yes	N/A
Bayer	N/A	N/A	yes	N/A
YCbCr	N/A	N/A	N/A	yes

By setting the input and output format the conversion is automatically applied if a conversion is possible. Otherwise the applet will output unchanged values. See *FG_PIXELFORMAT* and *FG_FORMAT*.

Chapter 15. Output Format

The following parameter can be used to configure the applet's image output format i.e. the format and bit alignment.

15.1. FG_FORMAT

Parameter *FG_FORMAT* is used to set and determine the output formats of the DMA channels. An output format value specifies the number of bits and the color format of the output.

This applet has an internal processing bit width of 16 bits. Any selected camera pixel format is mapped to this internal bit width. Check the camera parameter section to learn about the mapping of the camera bits to the internal bit width. For a definition on how to map the internal bits to the output bits, check parameter *FG_BITALIGNMENT*.

Moreover, the color converter of this applet can convert between different color formats of the input and output. Check Chapter 14, '*Color Converter*' for more information.

This applet supports the following output formats:

- **FG_BGR8** and **FG_RGB8**: 24 bit BGR/RGB color format with 8 bit/component.
- **FG_BGRA8** and **FG_RGBA8**: Color format with 8 bit/component. Component "a" has value zero.
- **FG_BGR10** and **FG_RGB10**: 30 bit BGR/RGB color format with 10 bit/component.

30 Bit Output Format

Note that in the 30 bit output format 1 pixel and its 3 color components are distributed over multiple bytes. Also, two successive pixel might share one byte. The pixel are directly aligned in memory. Thus 8 successive color components are stored in 10 byte. The DMA transfer might be filled with random content for the last bytes.

- **FG_BGR12** and **FG_RGB12**: 36 bit BGR/RGB color format with 12 bit/component.

36 Bit Output Format

Note that in the 36 bit output format 1 pixel and its 3 color components are distributed over multiple bytes. Also, two successive pixel might share one byte. The pixel are directly aligned in memory. Thus 2 successive color components are stored in 3 byte or two pixel in 9 Byte. The DMA transfer might be filled with random content for the last bytes.

- **FG_BGR14** and **FG_RGB14**: 42 bit BGR/RGB color format with 14 bit/component.

42 Bit Output Format

Note that in the 42 bit output format 1 pixel and its 3 color components are distributed over multiple bytes. Also, two successive pixel might share one byte. The pixel are directly aligned in memory. Thus 4 successive color components are stored in 7 byte or four pixel in 21 Byte. The DMA transfer might be filled with random content for the last bytes.

- **FG_BGR16** and **FG_RGB16**: 48 bit BGR/RGB color format with 16 bit/component.

BGR vs. RGB Memory Alignement

Note that the color components are either written to the PC buffer in the common blue, green, red (BGR) or red, green, blue order. So either the blue or red color component is at the lower memory address.

- **FG_MONO8**: 8 bit grayscale format
- **FG_MONO10**: 10 bit grayscale format

10 Bit Output Format

Note that in the 10 bit output format 1 pixel is distributed over more than one byte. Also, two successive pixel share one byte. The pixel are directly aligned in memory. Thus 8 successive pixel are stored in 10 byte. The DMA transfer might be filled with random content for the last bytes.

- **FG_MONO12**: 12 bit grayscale format

12 Bit Output Format

Note that in the 12 bit output format 1 pixel is distributed over more than one byte. Also, two successive pixel share the same byte. The pixel are directly aligned in memory. Thus 2 successive pixel are stored in 3 byte. The DMA transfer might be filled with random content for the last bytes.

- **FG_MONO14**: 14 bit grayscale format

14 Bit Output Format

Note that in the 14 bit output format 1 pixel is distributed over more than one byte. Also, two successive pixel share the same byte. The pixel are directly aligned in memory. Thus 12 successive pixel are stored in 21 byte. The DMA transfer might be filled with random content for the last bytes.

- **FG_MONO16**: 16 bit grayscale format

DMA Bandwidth

Keep in mind that for the 16 bit output mode, the DMA bandwidth might not be sufficient to process the camera input data. Check Section 1.2, 'Bandwidth' for more information.

- **FG_BAYERGR8**, **FG_BAYERRG8**, **FG_BAYERGB8** and **FG_BAYERBG8**: 8 bit Bayer format Green-followed-by-Red, Red-followed-by-Green, Green-followed-by-Blue and Blue-followed-by-Green.
- **FG_BAYERGR10**, **FG_BAYERRG10**, **FG_BAYERGB10** and **FG_BAYERBG10**: 10 bit Bayer format Green-followed-by-Red, Red-followed-by-Green, Green-followed-by-Blue and Blue-followed-by-Green.

10 Bit Output Format

Note that in the 10 bit output format 1 pixel is distributed over more than one byte. Also, two successive pixel share one byte. The pixel are directly aligned in memory. Thus 8 successive pixel are stored in 10 byte. The DMA transfer might be filled with random content for the last bytes.

- **FG_BAYERGR12**, **FG_BAYERRG12**, **FG_BAYERGB12** and **FG_BAYERBG12**: 12 bit Bayer format Green-followed-by-Red, Red-followed-by-Green, Green-followed-by-Blue and Blue-followed-by-Green.

12 Bit Output Format

Note that in the 12 bit output format 1 pixel is distributed over more than one byte. Also, two successive pixel share the same byte. The pixel are directly aligned in memory. Thus 2 successive pixel are stored in 3 byte. The DMA transfer might be filled with random content for the last bytes.

- **FG_BAYERGR14**, **FG_BAYERRG14**, **FG_BAYERGB14** and **FG_BAYERBG14**: 14 bit Bayer format Green-followed-by-Red, Red-followed-by-Green, Green-followed-by-Blue and Blue-followed-by-Green.

14 Bit Output Format

Note that in the 14 bit output format 1 pixel is distributed over more than one byte. Also, two successive pixel share the same byte. The pixel are directly aligned in memory. Thus 12 successive pixel are stored in 21 byte. The DMA transfer might be filled with random content for the last bytes.

- **FG_BAYERGR16**, **FG_BAYERRG16**, **FG_BAYERGB16** and **FG_BAYERBG16**: 16 bit Bayer format Green-followed-by-Red, Red-followed-by-Green, Green-followed-by-Blue and Blue-followed-by-Green.

DMA Bandwidth

Keep in mind that for the 16 bit output mode, the DMA bandwidth might not be sufficient to process the camera input data. Check Section 1.2, 'Bandwidth' for more information.

- **FG_YUV422_8**: YUV 422 output in 8 bit per component.

Table 15.1. Parameter properties of FG_FORMAT

Property	Value
Name	**FG_FORMAT**
Display Name	**Output Format**
Type	**Enumeration**
Access policy	**Read/Write**
Storage policy	**Persistent**
Allowed values	**FG_MONO8** Mono 8 **FG_MONO10** Mono 10p **FG_MONO12** Mono 12p **FG_MONO14** Mono 14p **FG_MONO16** Mono 16 **FG_BGR8** BGR 8bit **FG_BGR10** BGR 10bit **FG_BGR12** BGR 12bit **FG_BGR14** BGR 14p **FG_BGR16** BGR 16bit **FG_RGB8** RGB 8 **FG_RGB10** RGB 10p **FG_RGB12** RGB 12p **FG_RGB14** RGB 14p **FG_RGB16** RGB 16 **FG_BGRA8** BGRA 8 **FG_RGBA8** RGBA 8 **FG_YUV422_8** YCbCr422_8
Default value	**FG_MONO8**

Example 15.1. Usage of FG_FORMAT

```
int result = 0;
int value = FG_MONO8;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_FORMAT, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_FORMAT, &value, 0, type)) < 0) {
    /* error handling */
}
```

15.2. FG_BITALIGNMENT

The bit alignment is used to map the pixel bits of the internal processing with a depth of 16 bit to the configured DMA output bit depth defined by parameter *FG_FORMAT*.

You can select three different modes: Left aligned, right aligned and a custom shift mode. If you select left aligned, the applet will map the upper bits of the internal processing bit width to the available output bits. If you select right aligned, the applet will map the lower bits of the internal processing bit width to the available output bits. If you want to define a custom bit shift, you'll need to set the parameter to CustomBitShift and use parameter *FG_CUSTOM_BIT_SHIFT_RIGHT* to define the bit shift.

Keep in mind that the internal processing bit width has nothing to do with the camera pixel format. Check the camera parameter section to learn about the mapping of the camera bits to the internal bit width.

Table 15.2. Parameter properties of FG_BITALIGNMENT

Property	Value	
Name	**FG_BITALIGNMENT**	
Display Name	**Bit Alignment**	
Type	**Enumeration**	
Access policy	**Read/Write/Change**	
Storage policy	**Persistent**	
Allowed values	**FG_LEFT_ALIGNED** **FG_RIGHT_ALIGNED** **FG_CUSTOM_BIT_SHIFT_MODE**	Left Aligned Right Aligned Custom Bit Shift
Default value	**FG_LEFT_ALIGNED**	

Example 15.2. Usage of FG_BITALIGNMENT

```
int result = 0;
int value = FG_LEFT_ALIGNED;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_BITALIGNMENT, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_BITALIGNMENT, &value, 0, type)) < 0) {
    /* error handling */
}
```

15.3. FG_PIXELDEPTH

The pixel depth read-only parameter is used to determine the number of bits used to process a pixel in the applet. It represents the internal bit width.

Table 15.3. Parameter properties of FG_PIXELDEPTH

Property	Value
Name	**FG_PIXELDEPTH**
Display Name	**Pixel Depth**
Type	**Unsigned Integer**
Access policy	**Read-Only**
Storage policy	**Transient**
Allowed values	**Minimum 0** **Maximum 128** **Stepsize 1**
Unit of measure	**bit**

Example 15.3. Usage of FG_PIXELDEPTH

```
int result = 0;
unsigned int value = 8;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_getParameterWithType(fg, FG_PIXELDEPTH, &value, 0, type)) < 0) {
    /* error handling */
}
```

15.4. FG_CUSTOM_BIT_SHIFT_RIGHT

This parameter can only be used if parameter *FG_BITALIGNMENT* is set to **FG_CUSTOM_BIT_SHIFT_MODE**. If it is enabled, you can define a custom right bit shift value for the DMA output of the interface card. A shift of 0 means that the most significant bits (MSB) of the internal processing bit width are mapped to the output MSB. For example, if the applet has an internal processing bit width of 12 bit and you select a 10 bit output, the upper 10 bits are mapped to the output. If you select however a bit width of two, the lower 10 bits are mapped to the output. Note that this applet has an internal bit width of 16 bits.

Table 15.4. Parameter properties of FG_CUSTOM_BIT_SHIFT_RIGHT

Property	Value
Name	**FG_CUSTOM_BIT_SHIFT_RIGHT**
Display Name	**Bit Shift Right**
Type	**Unsigned Integer**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum 0** **Maximum 15** **Stepsize 1**
Default value	**0**
Unit of measure	**bit**

Example 15.4. Usage of FG_CUSTOM_BIT_SHIFT_RIGHT

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_setParameterWithType(fg, FG_CUSTOM_BIT_SHIFT_RIGHT, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_CUSTOM_BIT_SHIFT_RIGHT, &value, 0, type)) < 0) {
```

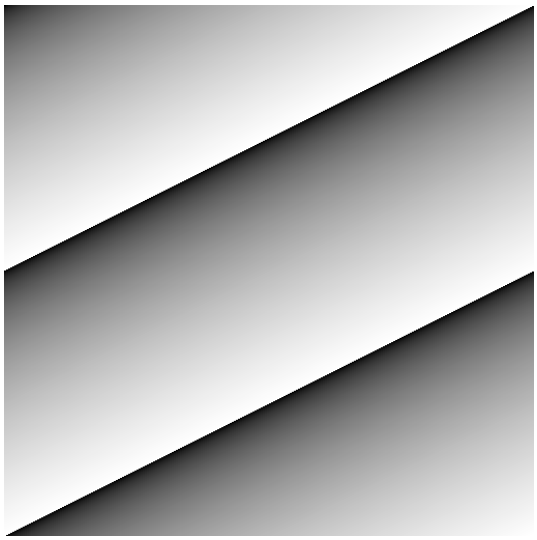
```
    /* error handling */
}
```

Chapter 16. Camera Simulator

The camera simulator is a convenient way to simulate cameras for first time applet tests. If the simulator is enabled it generates pattern frames of specified size and speed. The image data is replaced at the position of the camera i.e. all applet processing functionalities are applied to the generated images. Note that camera specific settings of the applet will not have any functionality.

The generated images are horizontal, diagonal or vertical grayscale patterns, such as the one shown in the following figure.

Figure 16.1. Generator Pattern



No Sub-Sensor sorting in Generated Images

The camera simulator will generate a simple grayscale pattern. If the camera or this applet uses sub sensor pixel sorting (sensor correction), the simulator will not generate images which represent the camera sensor.

16.1. FG_CAMERASIMULATOR_ENABLE

The camera simulator is enabled with this parameter. When you switch between camera mode and simulator, the applet will finalize the current frame before switching to the other input. Note that an activated simulator will have effect on parameter *FG_CAMSTATUS*.

Only 8bit support

The camera simulator will produce valid 8bit values only for 8bit pixel format. All other pixel formats will consist of packed 8bit data inside the packed format.

This will cause strange images in the simulation for higher bit depth than 8bit. Since this function is not related to productive usage this should be acceptable.

Table 16.1. Parameter properties of FG_CAMERASIMULATOR_ENABLE

Property	Value
Name	**FG_CAMERASIMULATOR_ENABLE**
Display Name	**Image Source**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**FG_CAMPORT** Camera **FG_CAMERASIMULATOR** Simulator
Default value	**FG_CAMPORT**

Example 16.1. Usage of FG_CAMERASIMULATOR_ENABLE

```
int result = 0;
int value = FG_CAMPORT;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_CAMERASIMULATOR_ENABLE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_CAMERASIMULATOR_ENABLE, &value, 0, type)) < 0) {
    /* error handling */
}
```

16.2. FG_CAMERASIMULATOR_WIDTH

The width of the generated frame is set with this parameter. You can enter any value. The applet will automatically round up to the next valid value limited due to internal processing granularity.

The range of the width depends on other parameters and is automatically determined from the applet. Decrease the speed for extending the range of the width value.

Table 16.2. Parameter properties of FG_CAMERASIMULATOR_WIDTH

Property	Value
Name	**FG_CAMERASIMULATOR_WIDTH**
Display Name	**Width**
Type	**Unsigned Integer**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum** 1 **Maximum** 65536 **Stepsize** 1
Default value	**1024**
Unit of measure	**pixel**

Example 16.2. Usage of FG_CAMERASIMULATOR_WIDTH

```
int result = 0;
unsigned int value = 1024;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;
```

```
if ((result = Fg_setParameterWithType(fg, FG_CAMERASIMULATOR_WIDTH, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_CAMERASIMULATOR_WIDTH, &value, 0, type)) < 0) {
    /* error handling */
}
```

16.3. FG_CAMERASIMULATOR_LINE_GAP

The simulator will generate a gap between the lines. The length of the gap is defined by this parameter. So the time of the gap depends on the pixel clock and the value.

You can enter any value. The applet will automatically round up to the next valid value.

The range of the line gap depends on other parameters and is automatically determined from the applet. Decrease the speed for extending the range of the line gap value.

The parameter can only be changed if *FG_CAMERASIMULATOR_SELECT_MODE* is set to **FG_PIXEL_FREQUENCY**.

Table 16.3. Parameter properties of FG_CAMERASIMULATOR_LINE_GAP

Property	Value
Name	**FG_CAMERASIMULATOR_LINE_GAP**
Display Name	**Line Gap**
Type	**Unsigned Integer**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum 0** **Maximum 65536** **Stepsize 1**
Default value	**0**
Unit of measure	**pixel**

Example 16.3. Usage of FG_CAMERASIMULATOR_LINE_GAP

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_setParameterWithType(fg, FG_CAMERASIMULATOR_LINE_GAP, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_CAMERASIMULATOR_LINE_GAP, &value, 0, type)) < 0) {
    /* error handling */
}
```

16.4. FG_CAMERASIMULATOR_HEIGHT

The height of the generated frame is set with this parameter.

The range of the height depends on other parameters and is automatically determined from the applet. Decrease the speed for extending the range of the height value.

Table 16.4. Parameter properties of FG_CAMERASIMULATOR_HEIGHT

Property	Value
Name	**FG_CAMERASIMULATOR_HEIGHT**
Display Name	**Height**
Type	**Unsigned Integer**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum 1** **Maximum 65536** **Stepsize 1**
Default value	**1024**
Unit of measure	**pixel**

Example 16.4. Usage of FG_CAMERASIMULATOR_HEIGHT

```
int result = 0;
unsigned int value = 1024;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_setParameterWithType(fg, FG_CAMERASIMULATOR_HEIGHT, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_CAMERASIMULATOR_HEIGHT, &value, 0, type)) < 0) {
    /* error handling */
}
```

16.5. FG_CAMERASIMULATOR_FRAME_GAP

The simulator will generate a gap between the frames. The length of the gap is defined by this parameter. So the time of the gap depends on the line rate and the value.

The range of the frame gap depends on other parameters and is automatically determined from the applet. Decrease the speed for extending the range of the frame gap value.

The parameter can not be changed if parameter *FG_CAMERASIMULATOR_SELECT_MODE* is set to **FG_FRAMERATE**.

Table 16.5. Parameter properties of FG_CAMERASIMULATOR_FRAME_GAP

Property	Value
Name	**FG_CAMERASIMULATOR_FRAME_GAP**
Display Name	**Frame Gap**
Type	**Unsigned Integer**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum 0** **Maximum 65536** **Stepsize 1**
Default value	**0**
Unit of measure	**pixel**

Example 16.5. Usage of FG_CAMERASIMULATOR_FRAME_GAP

```
int result = 0;
```

```
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_setParameterWithType(fg, FG_CAMERASIMULATOR_FRAME_GAP, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_CAMERASIMULATOR_FRAME_GAP, &value, 0, type)) < 0) {
    /* error handling */
}
```

16.6. FG_CAMERASIMULATOR_PATTERN

The simulator will generate pixel value ramps from 0 to 255. As this applet is capable of using monochrome or RGB inputs.

The following three types of patterns can be generated and selected by this parameter.

- **FG_HORIZONTAL**

  A horizontal pattern. Values are increased by 1 in x-direction.

- **FG_VERTICAL**

  A vertical pattern. Values are increased by 1 in y-direction.

- **FG_DIAGONAL**

  A diagonal pattern. Values are increased by 1 in x and y-direction.

Table 16.6. Parameter properties of FG_CAMERASIMULATOR_PATTERN

Property	Value
Name	**FG_CAMERASIMULATOR_PATTERN**
Display Name	**Pattern**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**FG_HORIZONTAL** Horizontal **FG_VERTICAL** Vertical **FG_DIAGONAL** Diagonal
Default value	**FG_DIAGONAL**

Example 16.6. Usage of FG_CAMERASIMULATOR_PATTERN

```
int result = 0;
int value = FG_DIAGONAL;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_CAMERASIMULATOR_PATTERN, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_CAMERASIMULATOR_PATTERN, &value, 0, type)) < 0) {
    /* error handling */
}
```

16.7. FG_CAMERASIMULATOR_PATTERN_OFFSET

Using this parameter, an offset value can be added to to the generated patterns. After acquisition start, the offset is added. For example, the very first pixel of an image will start with the offset value instead of 0.

Table 16.7. Parameter properties of FG_CAMERASIMULATOR_PATTERN_OFFSET

Property	Value
Name	**FG_CAMERASIMULATOR_PATTERN_OFFSET**
Display Name	**Pattern Offset**
Type	**Unsigned Integer**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum** 0 **Maximum** 255 **Stepsize** 1
Default value	**0**
Unit of measure	**pixel value**

Example 16.7. Usage of FG_CAMERASIMULATOR_PATTERN_OFFSET

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_setParameterWithType(fg, FG_CAMERASIMULATOR_PATTERN_OFFSET, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_CAMERASIMULATOR_PATTERN_OFFSET, &value, 0, type)) < 0) {
    /* error handling */
}
```

16.8. FG_CAMERASIMULATOR_ROLL

The generated pattern can be 'rolled'. With every new frame, all pattern pixels are increased by value one. At the wrap-around value 256, the pixel will get value 0. The generated images look like a moving (rolling) image.

Table 16.8. Parameter properties of FG_CAMERASIMULATOR_ROLL

Property	Value
Name	**FG_CAMERASIMULATOR_ROLL**
Display Name	**Roll**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**FG_ON** On **FG_OFF** Off
Default value	**FG_ON**

Example 16.8. Usage of FG_CAMERASIMULATOR_ROLL

```
int result = 0;
int value = FG_ON;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_CAMERASIMULATOR_ROLL, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_CAMERASIMULATOR_ROLL, &value, 0, type)) < 0) {
    /* error handling */
}
```

16.9. FG_CAMERASIMULATOR_SELECT_MODE

The simulator will generate the images with a certain speed. Users are allowed to select whether they want to set the pixel frequency, line rate or frame rate to control the speed. This parameter selects the mode.

Table 16.9. Parameter properties of FG_CAMERASIMULATOR_SELECT_MODE

Property	Value	
Name	**FG_CAMERASIMULATOR_SELECT_MODE**	
Display Name	**Speed Mode**	
Type	**Enumeration**	
Access policy	**Read/Write/Change**	
Storage policy	**Persistent**	
Allowed values	**FG_PIXEL_FREQUENCY** **FG_LINERATE** **FG_FRAMERATE**	Pixel Frequency Line Rate Frame Rate
Default value	**FG_LINERATE**	

Example 16.9. Usage of FG_CAMERASIMULATOR_SELECT_MODE

```
int result = 0;
int value = FG_LINERATE;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_CAMERASIMULATOR_SELECT_MODE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_CAMERASIMULATOR_SELECT_MODE, &value, 0, type)) < 0) {
    /* error handling */
}
```

16.10. FG_CAMERASIMULATOR_PIXEL_FREQUENCY

This parameter sets the pixel frequency. Note that the generator only simulates cameras. It is made for a first time use of the applet and user SDK verification. The camera simulator cannot reflect the exact timings and frequencies of cameras.

To set the pixel frequency, you will need to set parameter *FG_CAMERASIMULATOR_SELECT_MODE* to **FG_PIXEL_FREQUENCY**.

Any floating point value can be inserted to the parameter. However, the applet will round the value to the next valid value. Read the parameter value to find out the new rounded value.

Table 16.10. Parameter properties of FG_CAMERASIMULATOR_PIXEL_FREQUENCY

Property	Value	
Name	**FG_CAMERASIMULATOR_PIXEL_FREQUENCY**	
Display Name	**Pixel Frequency**	
Type	**Double**	
Access policy	**Read/Write/Change**	
Storage policy	**Persistent**	
Allowed values	**Minimum** **Maximum** **Stepsize**	**0.31249999999999994** **1250.0** **0.625**
Default value	**39.375**	
Unit of measure	**MHz**	

Example 16.10. Usage of FG_CAMERASIMULATOR_PIXEL_FREQUENCY

```
int result = 0;
double value = 39.375;
const enum FgParamTypes type = FG_PARAM_TYPE_DOUBLE;

if ((result = Fg_setParameterWithType(fg, FG_CAMERASIMULATOR_PIXEL_FREQUENCY, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_CAMERASIMULATOR_PIXEL_FREQUENCY, &value, 0, type)) < 0) {
    /* error handling */
}
```

16.11. FG_CAMERASIMULATOR_LINERATE

This parameter sets the line rate of the generated images.

To set the line rate, you will need to set parameter *FG_CAMERASIMULATOR_SELECT_MODE* to **FG_LINERATE**.

In line rate mode, the pixel frequency is set to the maximum.

Any floating point value can be inserted to the parameter. However, the applet will round the value to the next valid value. Read the parameter value to find out the new rounded value.

Table 16.11. Parameter properties of FG_CAMERASIMULATOR_LINERATE

Property	Value
Name	**FG_CAMERASIMULATOR_LINERATE**
Display Name	**Line Rate**
Type	**Double**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum 0.15** **Maximum 4.464285714285714E7** **Stepsize 7.0E-11**
Default value	**10240.0**
Unit of measure	**Hz**

Example 16.11. Usage of FG_CAMERASIMULATOR_LINERATE

```
int result = 0;
double value = 10240.0;
const enum FgParamTypes type = FG_PARAM_TYPE_DOUBLE;

if ((result = Fg_setParameterWithType(fg, FG_CAMERASIMULATOR_LINERATE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_CAMERASIMULATOR_LINERATE, &value, 0, type)) < 0) {
    /* error handling */
}
```

16.12. FG_CAMERASIMULATOR_FRAMERATE

This parameter sets the frame rate of the generated images.

To set the frame rate, you will need to set parameter *FG_CAMERASIMULATOR_SELECT_MODE* to **FG_FRAMERATE**.

In frame rate mode, the pixel frequency is set to the maximum and the line gap is set to zero.

Any floating point value can be inserted to the parameter. However, the applet will round the value to the next valid value. Read the parameter value to find out the new rounded value.

Table 16.12. Parameter properties of FG_CAMERASIMULATOR_FRAMERATE

Property	Value
Name	**FG_CAMERASIMULATOR_FRAMERATE**
Display Name	**Framerate**
Type	**Double**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum 0.15** **Maximum 4.464285714285714E7** **Stepsize 7.0E-11**
Default value	**10.0**
Unit of measure	**Hz**

Example 16.12. Usage of FG_CAMERASIMULATOR_FRAMERATE

```
int result = 0;
double value = 10.0;
const enum FgParamTypes type = FG_PARAM_TYPE_DOUBLE;

if ((result = Fg_setParameterWithType(fg, FG_CAMERASIMULATOR_FRAMERATE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_CAMERASIMULATOR_FRAMERATE, &value, 0, type)) < 0) {
    /* error handling */
}
```

16.13. FG_CAMERASIMULATOR_TRIGGER_MODE

You can either use the camera simulator in free run mode or the simulator can be triggered by the output of the trigger module of this applet. As this applet uses a CoaxPress camera interface, the CXP trigger output of the respective camera port is used as camera simulator trigger input. The rising edge of the trigger will be used.

You can choose between line trigger and frame trigger mode. In line trigger mode, a rising edge at the input will output a line from the camera simulator. For frame trigger mode, the input will trigger the output of a frame.

Trigger frequency must not exceed the speed of the camera simulator

Same as for real cameras, it is very important that the frequency of the trigger pulses do not exceed the maximum speed of the camera simulator. Set the camera simulator to a sufficiently large speed to avoid line or frame lost.

Table 16.13. Parameter properties of FG_CAMERASIMULATOR_TRIGGER_MODE

Property	Value
Name	**FG_CAMERASIMULATOR_TRIGGER_MODE**
Display Name	**Trigger Mode**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**SIMULATION_FREE_RUN** Free Run **RISING_EDGE_TRIGGERS_LINE** Rising Edge Triggers Line **RISING_EDGE_TRIGGERS_FRAME** Rising Edge Triggers Frame
Default value	**SIMULATION_FREE_RUN**

Example 16.13. Usage of FG_CAMERASIMULATOR_TRIGGER_MODE

```
int result = 0;
int value = SIMULATION_FREE_RUN;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_CAMERASIMULATOR_TRIGGER_MODE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_CAMERASIMULATOR_TRIGGER_MODE, &value, 0, type)) < 0) {
    /* error handling */
}
```

16.14. FG_CAMERASIMULATOR_ACTIVE

Table 16.14. Parameter properties of FG_CAMERASIMULATOR_ACTIVE

Property	Value
Name	**FG_CAMERASIMULATOR_ACTIVE**
Display Name	**Active Parts**
Type	**Unsigned Integer**
Access policy	**Read-Only**
Storage policy	**Persistent**
Allowed values	**Minimum** 1 **Maximum** 2000 **Stepsize** 1
Unit of measure	**pixel**

Example 16.14. Usage of FG_CAMERASIMULATOR_ACTIVE

```
int result = 0;
unsigned int value = 1024;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_getParameterWithType(fg, FG_CAMERASIMULATOR_ACTIVE, &value, 0, type)) < 0) {
    /* error handling */
}
```

16.15. FG_CAMERASIMULATOR_PASSIVE

Table 16.15. Parameter properties of FG_CAMERASIMULATOR_PASSIVE

Property	Value
Name	**FG_CAMERASIMULATOR_PASSIVE**
Display Name	**Passive Parts**
Type	**Unsigned Integer**
Access policy	**Read-Only**
Storage policy	**Persistent**
Allowed values	**Minimum 1** **Maximum 2000** **Stepsize 1**
Unit of measure	**pixel**

Example 16.15. Usage of FG_CAMERASIMULATOR_PASSIVE

```
int result = 0;
unsigned int value = 1024;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_getParameterWithType(fg, FG_CAMERASIMULATOR_PASSIVE, &value, 0, type)) < 0) {
    /* error handling */
}
```

Chapter 17. Miscellaneous

The miscellaneous module category summarizes other read and write parameters such as the camera status, buffer fill levels, DMA transfer lengths, time stamps and buffer fill-levels.

17.1. FG_SYSTEMMONITOR_CHANNEL_CURRENT

Returns the channel current in Ampere.

Table 17.1. Parameter properties of FG_SYSTEMMONITOR_CHANNEL_CURRENT

Property	Value
Name	**FG_SYSTEMMONITOR_CHANNEL_CURRENT**
Display Name	**Channel Current**
Type	**Double Field**
Field Size	**1**
Access policy	**Read-Only**
Storage policy	**Transient**
Unit of measure	**A**

Example 17.1. Usage of FG_SYSTEMMONITOR_CHANNEL_CURRENT

```
int result = 0;

FieldParameterDouble access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMDOUBLE;

    if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_CHANNEL_CURRENT, &access, 0, type)) < 0) {
        /* error handling */
    }
}
```

17.2. FG_SYSTEMMONITOR_CHANNEL_VOLTAGE

Returns the channel voltage.

Table 17.2. Parameter properties of FG_SYSTEMMONITOR_CHANNEL_VOLTAGE

Property	Value
Name	**FG_SYSTEMMONITOR_CHANNEL_VOLTAGE**
Display Name	**Channel Voltage**
Type	**Double Field**
Field Size	**1**
Access policy	**Read-Only**
Storage policy	**Transient**
Unit of measure	**V**

Example 17.2. Usage of FG_SYSTEMMONITOR_CHANNEL_VOLTAGE

```
int result = 0;
```

```
FieldParameterDouble access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMDOUBLE;

    if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_CHANNEL_VOLTAGE, &access, 0, type)) < 0) {
        /* error handling */
    }
}
```

17.3. FG_SYSTEMMONITOR_POWER_OVER_CXP_STATE

Shows the current power over CXP state.

Table 17.3. Parameter properties of FG_SYSTEMMONITOR_POWER_OVER_CXP_STATE

Property	Value
Name	**FG_SYSTEMMONITOR_POWER_OVER_CXP_STATE**
Display Name	**Power Over CXP State**
Type	**Unsigned Integer Field**
Field Size	**1**
Access policy	**Read-Only**
Storage policy	**Transient**

Example 17.3. Usage of FG_SYSTEMMONITOR_POWER_OVER_CXP_STATE

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

    if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_POWER_OVER_CXP_STATE, &access, 0, type)) < 0) {
        /* error handling */
    }
}
```

17.4. FG_SYSTEMMONITOR_POWER_OVER_CXP_CONTROLLER_ENABLED

Returns, whether the Power over CXP controller is enabled: Yes or No

Table 17.4. Parameter properties of FG_SYSTEMMONITOR_POWER_OVER_CXP_CONTROLLER_ENABLED

Property	Value
Name	**FG_SYSTEMMONITOR_POWER_OVER_CXP_CONTROLLER_ENABLED**
Display Name	**Power Over CXP Enabled**
Type	**Unsigned Integer Field**
Field Size	**1**
Access policy	**Read-Only**
Storage policy	**Transient**

Example 17.4. Usage of FG_SYSTEMMONITOR_POWER_OVER_CXP_CONTROLLER_ENABLED

```
int result = 0;

FieldParameterInt access;
```

```
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

    if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_POWER_OVER_CXP_CONTROLLER_ENABLED, &access, 0, type)) < 0) {
        /* error handling */
    }
}
```

17.5. FG_SYSTEMMONITOR_DECODER_8B_10B_ERROR

Decoder 8b 10b errors

Table 17.5. Parameter properties of FG_SYSTEMMONITOR_DECODER_8B_10B_ERROR

Property	Value
Name	**FG_SYSTEMMONITOR_DECODER_8B_10B_ERROR**
Display Name	**Decoder 8b10b Error**
Type	**Unsigned Integer Field (64 Bit)**
Field Size	**1**
Access policy	**Read-Only**
Storage policy	**Transient**

Example 17.5. Usage of FG_SYSTEMMONITOR_DECODER_8B_10B_ERROR

```
int result = 0;

FieldParameterAccess access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMACCESS;

    if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_DECODER_8B_10B_ERROR, &access, 0, type)) < 0) {
        /* error handling */
    }
}
```

17.6. FG_SYSTEMMONITOR_BYTE_ALIGNMENT_8B_10B_LOCKED

Byte Alignment 8b10b locked

Table 17.6. Parameter properties of FG_SYSTEMMONITOR_BYTE_ALIGNMENT_8B_10B_LOCKED

Property	Value
Name	**FG_SYSTEMMONITOR_BYTE_ALIGNMENT_8B_10B_LOCKED**
Display Name	**Byte Alignment 8B 10 B Locked**
Type	**Unsigned Integer Field**
Field Size	**1**
Access policy	**Read-Only**
Storage policy	**Transient**

Example 17.6. Usage of FG_SYSTEMMONITOR_BYTE_ALIGNMENT_8B_10B_LOCKED

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

    if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_BYTE_ALIGNMENT_8B_10B_LOCKED, &access, 0, type)) < 0) {
        /* error handling */
```

```
    }
}
```

17.7. FG_SYSTEMMONITOR_PORT_BIT_RATE

Returns the port bit rate.

Table 17.7. Parameter properties of FG_SYSTEMMONITOR_PORT_BIT_RATE

Property	Value
Name	**FG_SYSTEMMONITOR_PORT_BIT_RATE**
Display Name	**Port Bit Rate**
Type	**Double Field**
Field Size	**1**
Access policy	**Read-Only**
Storage policy	**Transient**
Unit of measure	**Gb/s**

Example 17.7. Usage of FG_SYSTEMMONITOR_PORT_BIT_RATE

```
int result = 0;

FieldParameterDouble access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMDOUBLE;

    if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_PORT_BIT_RATE, &access, 0, type)) < 0) {
        /* error handling */
    }
}
```

17.8. FG_SYSTEMMONITOR_STREAM_PACKET_SIZE

Returns the Stream Packet Size in Bytes. Value range is between 4 and 65535 Bytes in steps of 4 Bytes.

Table 17.8. Parameter properties of FG_SYSTEMMONITOR_STREAM_PACKET_SIZE

Property	Value
Name	**FG_SYSTEMMONITOR_STREAM_PACKET_SIZE**
Display Name	**Stream Packet Size**
Type	**Unsigned Integer Field**
Field Size	**1**
Access policy	**Read-Only**
Storage policy	**Transient**

Example 17.8. Usage of FG_SYSTEMMONITOR_STREAM_PACKET_SIZE

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

    if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_STREAM_PACKET_SIZE, &access, 0, type)) < 0) {
        /* error handling */
    }
```

```
}
```

17.9. FG_SYSTEMMONITOR_CXP_STANDARD

Returns the CXP Standard.

Table 17.9. CXP standard enumerator

CXP standard		
CXP_1_0		
CXP_1_1_1		
CXP_2_0		
unknown		

Table 17.10. Parameter properties of FG_SYSTEMMONITOR_CXP_STANDARD

Property	Value
Name	**FG_SYSTEMMONITOR_CXP_STANDARD**
Display Name	**CXP Standard**
Type	**Unsigned Integer Field**
Field Size	**1**
Access policy	**Read-Only**
Storage policy	**Transient**

Example 17.9. Usage of FG_SYSTEMMONITOR_CXP_STANDARD

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

    if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_CXP_STANDARD, &access, 0, type)) < 0) {
        /* error handling */
    }
}
```

17.10. FG_SYSTEMMONITOR_RX_STREAM_INCOMPLETE_COUNT

Returns the number of incomplete stream counts. Value range is between 0 and 8191 in steps of 1.

Table 17.11. Parameter properties of FG_SYSTEMMONITOR_RX_STREAM_INCOMPLETE_COUNT

Property	Value
Name	**FG_SYSTEMMONITOR_RX_STREAM_INCOMPLETE_COUNT**
Display Name	**Stream Incomplete Count**
Type	**Unsigned Integer Field**
Field Size	**1**
Access policy	**Read-Only**
Storage policy	**Transient**

Example 17.10. Usage of FG_SYSTEMMONITOR_RX_STREAM_INCOMPLETE_COUNT

```
int result = 0;
```

```
FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

    if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_RX_STREAM_INCOMPLETE_COUNT, &access, 0, type)) < 0) {
        /* error handling */
    }
}
```

17.11. FG_SYSTEMMONITOR_RX_UNKNOWN_DATA_RECEIVED_COUNT

Returns the number of incomplete stream counts. Value range is between 0 and 8191 in steps of 1.

Table 17.12. Parameter properties of FG_SYSTEMMONITOR_RX_UNKNOWN_DATA_RECEIVED_COUNT

Property	Value
Name	**FG_SYSTEMMONITOR_RX_UNKNOWN_DATA_RECEIVED_COUNT**
Display Name	**Unknown Data received Count**
Type	**Unsigned Integer Field**
Field Size	**1**
Access policy	**Read-Only**
Storage policy	**Transient**

Example 17.11. Usage of FG_SYSTEMMONITOR_RX_UNKNOWN_DATA_RECEIVED_COUNT

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

    if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_RX_UNKNOWN_DATA_RECEIVED_COUNT, &access, 0, type)) < 0) {
        /* error handling */
    }
}
```

17.12. FG_SYSTEMMONITOR_RX_PACKET_CRC_ERROR_COUNT

Returns the number of Packet CRC Errors. Value range is between 0 and 8191 in steps of 1.

Table 17.13. Parameter properties of FG_SYSTEMMONITOR_RX_PACKET_CRC_ERROR_COUNT

Property	Value
Name	**FG_SYSTEMMONITOR_RX_PACKET_CRC_ERROR_COUNT**
Display Name	**Packet CRC Error Count**
Type	**Unsigned Integer Field**
Field Size	**1**
Access policy	**Read-Only**
Storage policy	**Transient**

Example 17.12. Usage of FG_SYSTEMMONITOR_RX_PACKET_CRC_ERROR_COUNT

```
int result = 0;

FieldParameterInt access;
```

```
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

    if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_RX_PACKET_CRC_ERROR_COUNT, &access, 0, type)) < 0) {
        /* error handling */
    }
}
```

17.13. FG_SYSTEMMONITOR_RX_UNSUPPORTED_PACKET_COUNT

Returns the number of unsupported packets. Value range is between 0 and 8191 in steps of 1.

Table 17.14. Parameter properties of FG_SYSTEMMONITOR_RX_UNSUPPORTED_PACKET_COUNT

Property	Value
Name	**FG_SYSTEMMONITOR_RX_UNSUPPORTED_PACKET_COUNT**
Display Name	**Unsupported Packet Count**
Type	**Unsigned Integer Field**
Field Size	**1**
Access policy	**Read-Only**
Storage policy	**Transient**

Example 17.13. Usage of FG_SYSTEMMONITOR_RX_UNSUPPORTED_PACKET_COUNT

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

    if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_RX_UNSUPPORTED_PACKET_COUNT, &access, 0, type)) < 0) {
        /* error handling */
    }
}
```

17.14. FG_SYSTEMMONITOR_RX_LENGTH_ERROR_COUNT

Returns the number of length errors. Value range is between 0 and 8191 in steps of 1.

Table 17.15. Parameter properties of FG_SYSTEMMONITOR_RX_LENGTH_ERROR_COUNT

Property	Value
Name	**FG_SYSTEMMONITOR_RX_LENGTH_ERROR_COUNT**
Display Name	**Length Error Count**
Type	**Unsigned Integer Field**
Field Size	**1**
Access policy	**Read-Only**
Storage policy	**Transient**

Example 17.14. Usage of FG_SYSTEMMONITOR_RX_LENGTH_ERROR_COUNT

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

    if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_RX_LENGTH_ERROR_COUNT, &access, 0, type)) < 0) {
        /* error handling */
    }
```

```
}
```

17.15. FG_TIMEOUT

This parameter is used to set a timeout for DMA transfers. After a timeout the acquisition is stopped. But it is only a internal value that should not be used directly. Use the timeout value described in the Framegrabber API or microDisplay for acquisition in order to handle the functionality correctly.

Table 17.16. Parameter properties of FG_TIMEOUT

Property	Value
Name	**FG_TIMEOUT**
Display Name	**Timeout**
Type	**Unsigned Integer**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum 2** **Maximum 2147483646** **Stepsize 1**
Default value	**1000000**
Unit of measure	**seconds**

Example 17.15. Usage of FG_TIMEOUT

```
int result = 0;
unsigned int value = 1000000;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_setParameterWithType(fg, FG_TIMEOUT, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_TIMEOUT, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.16. FG_APPLET_VERSION

This parameter represents the version number of the applet. Please report this value for any support of the applet.

Table 17.17. Parameter properties of FG_APPLET_VERSION

Property	Value
Name	**FG_APPLET_VERSION**
Display Name	**Applet version**
Type	**Unsigned Integer**
Access policy	**Read-Only**
Storage policy	**Transient**
Allowed values	**Minimum 0** **Maximum 256** **Stepsize 1**

Example 17.16. Usage of FG_APPLET_VERSION

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_getParameterWithType(fg, FG_APPLET_VERSION, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.17. FG_APPLET_REVISION

This parameter represents the revision number of the applet. Please report this value for any support case with the applet.

Table 17.18. Parameter properties of FG_APPLET_REVISION

Property	Value
Name	**FG_APPLET_REVISION**
Display Name	**Applet revision**
Type	**Unsigned Integer**
Access policy	**Read-Only**
Storage policy	**Transient**
Allowed values	**Minimum 0** **Maximum 256** **Stepsize 1**

Example 17.17. Usage of FG_APPLET_REVISION

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_getParameterWithType(fg, FG_APPLET_REVISION, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.18. FG_APPLET_ID

This parameter returns the unique applet id of the applet as a string parameter.

Table 17.19. Parameter properties of FG_APPLET_ID

Property	Value
Name	**FG_APPLET_ID**
Display Name	**Applet Id**
Type	**String**
Access policy	**Read-Only**
Storage policy	**Transient**

Example 17.18. Usage of FG_APPLET_ID

```
int result = 0;
char* value = "";
const enum FgParamTypes type = FG_PARAM_TYPE_CHAR_PTR;

if ((result = Fg_getParameterWithType(fg, FG_APPLET_ID, &value, 0, type)) < 0) {
```

```
    /* error handling */
}
```

17.19. FG_APPLET_BUILD_TIME

This string parameter returns the hardware applet (HAP) build timestamp. To obtain the build time of the applet, check the DLL / SO file details. Mainly, this parameter is required for internal usage only.

Table 17.20. Parameter properties of FG_APPLET_BUILD_TIME

Property	Value
Name	**FG_APPLET_BUILD_TIME**
Display Name	**Build Time**
Type	**String**
Access policy	**Read-Only**
Storage policy	**Transient**

Example 17.19. Usage of FG_APPLET_BUILD_TIME

```
int result = 0;
char* value = "";
const enum FgParamTypes type = FG_PARAM_TYPE_CHAR_PTR;

if ((result = Fg_getParameterWithType(fg, FG_APPLET_BUILD_TIME, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.20. FG_HAP_FILE

The name of the Hardware-Applet (HAP) file on which this applet is based. Please report this read-only string parameter for any support case of the applet.

Table 17.21. Parameter properties of FG_HAP_FILE

Property	Value
Name	**FG_HAP_FILE**
Display Name	**HAP file**
Type	**String**
Access policy	**Read-Only**
Storage policy	**Transient**

Example 17.20. Usage of FG_HAP_FILE

```
int result = 0;
char* value = "";
const enum FgParamTypes type = FG_PARAM_TYPE_CHAR_PTR;

if ((result = Fg_getParameterWithType(fg, FG_HAP_FILE, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.21. FG_DMASTATUS

Using this parameter the status of a DMA channel can be obtained. Value "1" represents a started DMA i.e. a started acquisition. Value "0" represents a stopped acquisition.

Table 17.22. Parameter properties of FG_DMASTATUS

Property	Value
Name	**FG_DMASTATUS**
Display Name	**DMA Status**
Type	**Unsigned Integer**
Access policy	**Read-Only**
Storage policy	**Transient**
Allowed values	**Minimum 0** **Maximum 1** **Stepsize 1**

Example 17.21. Usage of FG_DMASTATUS

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_getParameterWithType(fg, FG_DMASTATUS, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.22. FG_CAMSTATUS

For CoaXPress, this parameter is not used. Please use the SDK's GenICam functions to identify camera detection state. More details on this can be found in the Basler Framegrabber SDK documentation.

Table 17.23. Parameter properties of FG_CAMSTATUS

Property	Value
Name	**FG_CAMSTATUS**
Display Name	**Camera Status**
Type	**Unsigned Integer**
Access policy	**Read-Only**
Storage policy	**Transient**
Allowed values	**Minimum 0** **Maximum 1** **Stepsize 1**

Example 17.22. Usage of FG_CAMSTATUS

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_getParameterWithType(fg, FG_CAMSTATUS, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.23. FG_CAMSTATUS_EXTENDED

This parameter provides extended information on the pixel clock from the camera, LVAL and FVAL, as well as the camera trigger signals, external trigger signals, buffer overflow status and buffer status. Each bit of the eight bit output word represents one parameter listed in the following:

- 0 = CameraClk, currently NOT supported by CoaXPress interface.

- 1 = CameraLval, currently NOT supported by CoaXPress interface.
- 2 = CameraFval, currently NOT supported by CoaXPress interface.
- 3 = Camera CC1 Signal, currently NOT supported by CoaXPress interface.
- 4 = ExTrg / external trigger, currently NOT supported by CoaXPress interface.
- 5 = BufferOverflow
- 6 = BufStatus, LSB
- 7 = BufStatus, MSB

Table 17.24. Parameter properties of FG_CAMSTATUS_EXTENDED

Property	Value
Name	**FG_CAMSTATUS_EXTENDED**
Display Name	**Camera Status Extended**
Type	**Unsigned Integer**
Access policy	**Read-Only**
Storage policy	**Transient**
Allowed values	**Minimum 0** **Maximum 255** **Stepsize 1**

Example 17.23. Usage of FG_CAMSTATUS_EXTENDED

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_getParameterWithType(fg, FG_CAMSTATUS_EXTENDED, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.24. FG_SYSTEMMONITOR_FPGA_TEMPERATURE

Returns the current FGPA die temperature.

Table 17.25. Parameter properties of FG_SYSTEMMONITOR_FPGA_TEMPERATURE

Property	Value
Name	**FG_SYSTEMMONITOR_FPGA_TEMPERATURE**
Display Name	**FGPA Temperature**
Type	**Double**
Access policy	**Read-Only**
Storage policy	**Transient**
Allowed values	**Minimum 0.0** **Maximum 1000.0** **Stepsize 0.0**
Unit of measure	**Celsius**

Example 17.24. Usage of FG_SYSTEMMONITOR_FPGA_TEMPERATURE

```
int result = 0;
double value = 0.0;
const enum FgParamTypes type = FG_PARAM_TYPE_DOUBLE;
```

```
if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_FPGA_TEMPERATURE, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.25. FG_SYSTEMMONITOR_FPGA_VCC_INT

Returns the current FPGA internal voltage.

Table 17.26. Parameter properties of FG_SYSTEMMONITOR_FPGA_VCC_INT

Property	Value
Name	**FG_SYSTEMMONITOR_FPGA_VCC_INT**
Display Name	**FGPA Vcc Int**
Type	**Double**
Access policy	**Read-Only**
Storage policy	**Transient**
Allowed values	**Minimum -1000.0** **Maximum 1000.0** **Stepsize 0.0**
Unit of measure	**V**

Example 17.25. Usage of FG_SYSTEMMONITOR_FPGA_VCC_INT

```
int result = 0;
double value = 0.0;
const enum FgParamTypes type = FG_PARAM_TYPE_DOUBLE;

if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_FPGA_VCC_INT, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.26. FG_SYSTEMMONITOR_FPGA_VCC_AUX

Returns the current FPGA Vcc auxiliary voltage.

Table 17.27. Parameter properties of FG_SYSTEMMONITOR_FPGA_VCC_AUX

Property	Value
Name	**FG_SYSTEMMONITOR_FPGA_VCC_AUX**
Display Name	**FGPA Vcc Aux**
Type	**Double**
Access policy	**Read-Only**
Storage policy	**Transient**
Allowed values	**Minimum -1000.0** **Maximum 1000.0** **Stepsize 0.0**
Unit of measure	**V**

Example 17.26. Usage of FG_SYSTEMMONITOR_FPGA_VCC_AUX

```
int result = 0;
double value = 0.0;
const enum FgParamTypes type = FG_PARAM_TYPE_DOUBLE;

if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_FPGA_VCC_AUX, &value, 0, type)) < 0) {
```

```
    /* error handling */
}
```

17.27. FG_SYSTEMMONITOR_FPGA_VCC_BRAM

Returns the current FPGA Vcc of the BlockRAM voltage.

Table 17.28. Parameter properties of FG_SYSTEMMONITOR_FPGA_VCC_BRAM

Property	Value
Name	**FG_SYSTEMMONITOR_FPGA_VCC_BRAM**
Display Name	**FGPA Vcc BRAM**
Type	**Double**
Access policy	**Read-Only**
Storage policy	**Transient**
Allowed values	**Minimum -1000.0** **Maximum 1000.0** **Stepsize 0.0**
Unit of measure	**V**

Example 17.27. Usage of FG_SYSTEMMONITOR_FPGA_VCC_BRAM

```
int result = 0;
double value = 0.0;
const enum FgParamTypes type = FG_PARAM_TYPE_DOUBLE;

if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_FPGA_VCC_BRAM, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.28. FG_SYSTEMMONITOR_CURRENT_LINK_SPEED

Returns the current link width of the frame grabber representing the number of PCIe lanes being used for data transfer. This is a value that should correspond to the number of hardware lanes the frame grabber is requiring, otherwise the possible maximum of DMA bandwidth can be reduced drastically.

Table 17.29. Parameter properties of FG_SYSTEMMONITOR_CURRENT_LINK_SPEED

Property	Value
Name	**FG_SYSTEMMONITOR_CURRENT_LINK_SPEED**
Display Name	**Current Link Speed**
Type	**Double**
Access policy	**Read-Only**
Storage policy	**Transient**
Allowed values	**Minimum 0.0** **Maximum 1000.0** **Stepsize 0.0**
Unit of measure	**GB/s**

Example 17.28. Usage of FG_SYSTEMMONITOR_CURRENT_LINK_SPEED

```
int result = 0;
double value = 0.0;
const enum FgParamTypes type = FG_PARAM_TYPE_DOUBLE;

if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_CURRENT_LINK_SPEED, &value, 0, type)) < 0) {
```

```
    /* error handling */
}
```

17.29. FG_SYSTEMMONITOR_PCIE_TRAINED_PAYLOAD_SIZE

Returns the PCIe packet size that was evaluated during the training period at boot-time.

Table 17.30. Parameter properties of FG_SYSTEMMONITOR_PCIE_TRAINED_PAYLOAD_SIZE

Property	Value
Name	**FG_SYSTEMMONITOR_PCIE_TRAINED_PAYLOAD_SIZE**
Display Name	**PCIe Trained Payload Size**
Type	**Unsigned Integer**
Access policy	**Read-Only**
Storage policy	**Transient**
Allowed values	**Minimum 0** **Maximum 1024** **Stepsize 0**
Unit of measure	**byte**

Example 17.29. Usage of FG_SYSTEMMONITOR_PCIE_TRAINED_PAYLOAD_SIZE

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_PCIE_TRAINED_PAYLOAD_SIZE, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.30. FG_SYSTEMMONITOR_PCIE_TRAINED_REQUEST_SIZE

Size in bytes of the PCIe packets payload used for the data transmission between the framegrabber and the PCIe bridge.

Table 17.31. Parameter properties of FG_SYSTEMMONITOR_PCIE_TRAINED_REQUEST_SIZE

Property	Value
Name	**FG_SYSTEMMONITOR_PCIE_TRAINED_REQUEST_SIZE**
Display Name	**PCIe Trained Request Size**
Type	**Unsigned Integer**
Access policy	**Read-Only**
Storage policy	**Transient**
Allowed values	**Minimum 0** **Maximum 4096** **Stepsize 0**
Unit of measure	**byte**

Example 17.30. Usage of FG_SYSTEMMONITOR_PCIE_TRAINED_REQUEST_SIZE

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_PCIE_TRAINED_REQUEST_SIZE, &value, 0, type)) < 0) {
    /* error handling */
```

```
}
```

17.31. FG_SYSTEMMONITOR_FPGA_DNA_LOW

The parameter *FG_SYSTEMMONITOR_FPGA_DNA_LOW* provides the lower 57 bit unique FPGA DNA.

Table 17.32. Parameter properties of FG_SYSTEMMONITOR_FPGA_DNA_LOW

Property	Value
Name	**FG_SYSTEMMONITOR_FPGA_DNA_LOW**
Display Name	**FPGA DNA Low**
Type	**Unsigned Integer (64 Bit)**
Access policy	**Read-Only**
Storage policy	**Transient**
Allowed values	**Minimum 0** **Maximum 144115188075855872** **Stepsize 1**

Example 17.31. Usage of FG_SYSTEMMONITOR_FPGA_DNA_LOW

```
int result = 0;
uint64_t value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT64_T;

if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_FPGA_DNA_LOW, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.32. FG_SYSTEMMONITOR_FPGA_DNA_HIGH

The parameter *FG_SYSTEMMONITOR_FPGA_DNA_HIGH* provides the upper 32s bit unique FPGA DNA.

Table 17.33. Parameter properties of FG_SYSTEMMONITOR_FPGA_DNA_HIGH

Property	Value
Name	**FG_SYSTEMMONITOR_FPGA_DNA_HIGH**
Display Name	**FPGA DNA High**
Type	**Unsigned Integer**
Access policy	**Read-Only**
Storage policy	**Transient**
Allowed values	**Minimum 0** **Maximum 4294967295** **Stepsize 1**

Example 17.32. Usage of FG_SYSTEMMONITOR_FPGA_DNA_HIGH

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_FPGA_DNA_HIGH, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.33. FG_SYSTEMMONITOR_EXTERNAL_POWER

Board external power state.

Table 17.34. Parameter properties of FG_SYSTEMMONITOR_EXTERNAL_POWER

Property	Value
Name	**FG_SYSTEMMONITOR_EXTERNAL_POWER**
Display Name	**External Power**
Type	**Enumeration**
Access policy	**Read-Only**
Storage policy	**Transient**
Allowed values	**FG_GOOD** Good **FG_NO_POWER** No Power

Example 17.33. Usage of FG_SYSTEMMONITOR_EXTERNAL_POWER

```
int result = 0;
int value = NO_POWER;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_EXTERNAL_POWER, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.34. Legacy

This category includes the legacy parameter for user software compatibility. The parameter of this category shouldn't be used anymore.

17.34.1. FG_CXP_TRIGGER_PACKET_MODE

This parameter is for legacy use only and shouldn't be used anymore.

Setting the parameter to the CXP standard triggers re-writing the *FG_TRIGGERCAMERA_SOURCE* legacy parameter.

However, if you set the parameter to rising edge only, the legacy compatibility mode applies. In this case, *FG_TRIGGERCAMERA_SOURCE_CXP1*, *FG_TRIGGERCAMERA_SOURCE_CXP2* and *FG_TRIGGERCAMERA_SOURCE_CXP3* are set to **GND**.

Be careful with this parameter as it overwrites the *FG_TRIGGERCAMERA_SOURCE_CXP0* and *FG_TRIGGERCAMERA_SOURCE_CXP0* parameters.

This legacy parameter can't be used in configuration files anymore.

Table 17.35. Parameter properties of FG_CXP_TRIGGER_PACKET_MODE

Property	Value
Name	**FG_CXP_TRIGGER_PACKET_MODE**
Display Name	**CXP Trigger Packet Mode**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**FG_STANDARD** CXP Trigger Standard **FG_RISING_EDGE_ONLY** CXP Trigger Rising
Default value	**FG_STANDARD**

Example 17.34. Usage of FG_CXP_TRIGGER_PACKET_MODE

```
int result = 0;
int value = FG_STANDARD;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_CXP_TRIGGER_PACKET_MODE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_CXP_TRIGGER_PACKET_MODE, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.34.2. FG_TRIGGERCAMERA_SOURCE

This is a legacy parameter. It is replaced by the *FG_TRIGGERCAMERA_SOURCE_CXP0* and *FG_TRIGGERCAMERA_SOURCE_CXP1* parameters. Before, the legacy parameter controlled the generation for CXP LinkTrigger0 associated with the start of a pulse and CXP LinkTrigger1 associated with the end of a pulse depending on the polarity settings. To keep the compatibility, when writing to this parameter, the value is copied to *FG_TRIGGERCAMERA_SOURCE_CXP0* and sets the same value to *FG_TRIGGERCAMERA_SOURCE_CXP1*.

Furthermore, *FG_TRIGGERCAMERA_SOURCE_EDGE_CXP1* is set to the inverse value of *FG_TRIGGERCAMERA_SOURCE_EDGE_CXP0*.

Reading the value only represents the status of *FG_TRIGGERCAMERA_SOURCE_CXP0* and can be ambiguous as the new parameters offer more possibilities which can't be represented with the legacy parameter.

This legacy parameter can't be used in configuration files anymore.

Table 17.36. Parameter properties of FG_TRIGGERCAMERA_SOURCE

Property	Value	
Name	**FG_TRIGGERCAMERA_SOURCE**	
Display Name	**Legacy Trigger Camera Source**	
Type	**Enumeration**	
Access policy	**Read/Write/Change**	
Storage policy	**Transient**	
Allowed values	**GND**	GND
	VCC	VCC
	FG_SIGNAL_CAM0_EXSYNC	Signal Exsync
	FG_SIGNAL_CAM0_EXSYNC2	Signal Exsync2
	FG_SIGNAL_CAM0_FLASH	Signal Flash
	FG_SIGNAL_CAM0_LVAL	Signal Line Valid
	FG_SIGNAL_CAM0_FVAL	Signal Frame Valid
	FG_SIGNAL_FRONT_GPI_0	Signal Front GPI 0
	FG_SIGNAL_FRONT_GPI_1	Signal Front GPI 1
Default value	**FG_SIGNAL_CAM0_EXSYNC**	

Example 17.35. Usage of FG_TRIGGERCAMERA_SOURCE

```
int result = 0;
int value = FG_SIGNAL_CAM0_EXSYNC;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_TRIGGERCAMERA_SOURCE, &value, 0, type)) < 0) {
    /* error handling */
```

```
}

if ((result = Fg_getParameterWithType(fg, FG_TRIGGERCAMERA_SOURCE, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.34.3. FG_TRIGGERCAMERA_POLARITY

This is a legacy parameter. It is replaced by the parameters *FG_TRIGGERCAMERA_SOURCE_CXP0* and *FG_TRIGGERCAMERA_SOURCE_CXP1* as well as *FG_TRIGGERCAMERA_SOURCE_EDGE_CXP0* and *FG_TRIGGERCAMERA_SOURCE_EDGE_CXP1*. Before, the legacy parameter defined which edge of the trigger signal is used for CXP LinkTrigger and CXP LinkTrigger1. Now, this can be done individually for each CXP link trigger.

To keep the compatibility, when writing to this parameter, the value is copied to *FG_TRIGGERCAMERA_SOURCE_EDGE_CXP0* and sets the inverse value to *FG_TRIGGERCAMERA_SOURCE_EDGE_CXP1*.

Furthermore, the value of parameter *FG_TRIGGERCAMERA_SOURCE_CXP0* is copied to *FG_TRIGGERCAMERA_SOURCE_CXP1*.

Reading the value only represents the status of *FG_TRIGGERCAMERA_SOURCE_EDGE_CXP0* and can be ambiguous as the new parameters offer more possibilities which can't be represented with the legacy parameter.

This legacy parameter can't be used in configuration files anymore.

Table 17.37. Parameter properties of FG_TRIGGERCAMERA_POLARITY

Property	Value
Name	**FG_TRIGGERCAMERA_POLARITY**
Display Name	**Legacy Trigger Camera Polarity**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Transient**
Allowed values	**FG_LOW** Low Active **FG_HIGH** High Active
Default value	**FG_HIGH**

Example 17.36. Usage of FG_TRIGGERCAMERA_POLARITY

```
int result = 0;
int value = FG_HIGH;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_TRIGGERCAMERA_POLARITY, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_TRIGGERCAMERA_POLARITY, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.35. Debug

17.35.1. FG_DEBUGSOURCE

This parameter is for internal testing. Please DON'T use this parameter.

Table 17.38. Parameter properties of FG_DEBUGSOURCE

Property	Value
Name	**FG_DEBUGSOURCE**
Display Name	**Debug Source**
Type	**Unsigned Integer**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum 0** **Maximum 0** **Stepsize 1**
Default value	**0**

Example 17.37. Usage of FG_DEBUGSOURCE

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_setParameterWithType(fg, FG_DEBUGSOURCE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_DEBUGSOURCE, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.35.2. FG_DEBUGSOURCENAME

This parameter is for internal testing. Please DON'T use this parameter.

Table 17.39. Parameter properties of FG_DEBUGSOURCENAME

Property	Value
Name	**FG_DEBUGSOURCENAME**
Display Name	**Debug Source Name**
Type	**String**
Access policy	**Read-Only**
Storage policy	**Transient**

Example 17.38. Usage of FG_DEBUGSOURCENAME

```
int result = 0;
char* value = "";
const enum FgParamTypes type = FG_PARAM_TYPE_CHAR_PTR;

if ((result = Fg_getParameterWithType(fg, FG_DEBUGSOURCENAME, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.35.3. FG_DEBUGSAVECONFIG

This parameter is for internal testing. Please DON'T use this parameter.

Table 17.40. Parameter properties of FG_DEBUGSAVECONFIG

Property	Value
Name	**FG_DEBUGSAVECONFIG**
Display Name	**Debug Save Config**
Type	**String**
Access policy	**Read/Write/Change**
Storage policy	**Transient**
Default value	**""**

Example 17.39. Usage of FG_DEBUGSAVECONFIG

```
int result = 0;
char* value = "";
const enum FgParamTypes type = FG_PARAM_TYPE_CHAR_PTR;

if ((result = Fg_setParameterWithType(fg, FG_DEBUGSAVECONFIG, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_DEBUGSAVECONFIG, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.35.4. FG_DEBUG_SLOWMODE

This parameter is for internal testing. Please DON'T use this parameter.

Table 17.41. Parameter properties of FG_DEBUG_SLOWMODE

Property	Value	
Name	**FG_DEBUG_SLOWMODE**	
Display Name	**Debug Slow Output**	
Type	**Enumeration**	
Access policy	**Read/Write/Change**	
Storage policy	**Persistent**	
Allowed values	**FG_SLOW_OFF**	Off
	FG_SLOW_SOFTWARE	Software
	FG_SLOW_PWM	PWM
Default value	**FG_SLOW_OFF**	

Example 17.40. Usage of FG_DEBUG_SLOWMODE

```
int result = 0;
int value = FG_SLOW_OFF;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_DEBUG_SLOWMODE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_DEBUG_SLOWMODE, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.35.5. FG_DEBUG_SOFTWRAE_SLOWGATE

This parameter is for internal testing. Please DON'T use this parameter.

Table 17.42. Parameter properties of FG_DEBUG_SOFTWRAE_SLOWGATE

Property	Value
Name	**FG_DEBUG_SOFTWRAE_SLOWGATE**
Display Name	**Debug Slow Software**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**FG_ON** On **FG_OFF** Off **FG_PULSE** Pulse
Default value	**FG_OFF**

Example 17.41. Usage of FG_DEBUG_SOFTWRAE_SLOWGATE

```
int result = 0;
int value = FG_OFF;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_DEBUG_SOFTWRAE_SLOWGATE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_DEBUG_SOFTWRAE_SLOWGATE, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.35.6. FG_DEBUG_PWM_SLOWRATE

This parameter is for internal testing. Please DON'T use this parameter.

Table 17.43. Parameter properties of FG_DEBUG_PWM_SLOWRATE

Property	Value
Name	**FG_DEBUG_PWM_SLOWRATE**
Display Name	**Slowrate PWM**
Type	**Unsigned Integer (64 Bit)**
Access policy	**Read/Write/Change**
Storage policy	**Transient**
Allowed values	**Minimum** 0 **Maximum** 144115188075855872 **Stepsize** 1
Default value	**10000000**

Example 17.42. Usage of FG_DEBUG_PWM_SLOWRATE

```
int result = 0;
uint64_t value = 10000000;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT64_T;

if ((result = Fg_setParameterWithType(fg, FG_DEBUG_PWM_SLOWRATE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_DEBUG_PWM_SLOWRATE, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.35.7. FG_DEBUG_VERSION

This parameter is for internal testing. Please DON'T use this parameter.

Table 17.44. Parameter properties of FG_DEBUG_VERSION

Property	Value
Name	**FG_DEBUG_VERSION**
Display Name	**Version**
Type	**String**
Access policy	**Read-Only**
Storage policy	**Transient**

Example 17.43. Usage of FG_DEBUG_VERSION

```
int result = 0;
char* value = "";
const enum FgParamTypes type = FG_PARAM_TYPE_CHAR_PTR;

if ((result = Fg_getParameterWithType(fg, FG_DEBUG_VERSION, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.35.8. FG_DEBUG_FRAMEID_TO_FIRSTPIXEL

This parameter is for internal testing. Please DON'T use this parameter.

Table 17.45. Parameter properties of FG_DEBUG_FRAMEID_TO_FIRSTPIXEL

Property	Value
Name	**FG_DEBUG_FRAMEID_TO_FIRSTPIXEL**
Display Name	**FrameID mapped**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Transient**
Allowed values	**FG_ON** On **FG_OFF** Off
Default value	**FG_OFF**

Example 17.44. Usage of FG_DEBUG_FRAMEID_TO_FIRSTPIXEL

```
int result = 0;
int value = FG_OFF;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_DEBUG_FRAMEID_TO_FIRSTPIXEL, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_DEBUG_FRAMEID_TO_FIRSTPIXEL, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.35.9. Input

17.35.9.1. FG_DEBUGINENABLE

This parameter is for internal testing. Please DON'T use this parameter.

Table 17.46. Parameter properties of FG_DEBUGINENABLE

Property	Value
Name	**FG_DEBUGINENABLE**
Display Name	**Debug Input Mode**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**FG_ON** On **FG_OFF** Off
Default value	**FG_OFF**

Example 17.45. Usage of FG_DEBUGINENABLE

```
int result = 0;
int value = FG_OFF;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_DEBUGINENABLE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_DEBUGINENABLE, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.35.9.2. FG_DEBUGFILE

This parameter is for internal testing. Please DON'T use this parameter.

Table 17.47. Parameter properties of FG_DEBUGFILE

Property	Value
Name	**FG_DEBUGFILE**
Display Name	**Debug File**
Type	**String**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Default value	**""**

Example 17.46. Usage of FG_DEBUGFILE

```
int result = 0;
char* value = "";
const enum FgParamTypes type = FG_PARAM_TYPE_CHAR_PTR;

if ((result = Fg_setParameterWithType(fg, FG_DEBUGFILE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_DEBUGFILE, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.35.9.3. FG_DEBUGINSERT

This parameter is for internal testing. Please DON'T use this parameter.

Table 17.48. Parameter properties of FG_DEBUGINSERT

Property	Value
Name	**FG_DEBUGINSERT**
Display Name	**Debug Insert Image**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Transient**
Allowed values	**FG_APPLY** Apply
Default value	**FG_APPLY**

Example 17.47. Usage of FG_DEBUGINSERT

```
int result = 0;
int value = FG_APPLY;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_DEBUGINSERT, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_DEBUGINSERT, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.35.9.4. FG_DEBUGWRITEPIXEL

This parameter is for internal testing. Please DON'T use this parameter.

Table 17.49. Parameter properties of FG_DEBUGWRITEPIXEL

Property	Value
Name	**FG_DEBUGWRITEPIXEL**
Display Name	**Debug Write Pixel**
Type	**Unsigned Integer (64 Bit)**
Access policy	**Read/Write/Change**
Storage policy	**Transient**
Allowed values	**Minimum** 0 **Maximum** 144115188075855872 **Stepsize** 1
Default value	**0**

Example 17.48. Usage of FG_DEBUGWRITEPIXEL

```
int result = 0;
uint64_t value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT64_T;

if ((result = Fg_setParameterWithType(fg, FG_DEBUGWRITEPIXEL, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_DEBUGWRITEPIXEL, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.35.9.5. FG_DEBUGWRITEFLAG

This parameter is for internal testing. Please DON'T use this parameter.

Table 17.50. Parameter properties of FG_DEBUGWRITEFLAG

Property	Value
Name	**FG_DEBUGWRITEFLAG**
Display Name	**Debug Write Flag**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Transient**
Allowed values	**FG_ENDOFLINE** EndOfLine **FG_ENDOFFRAME** EndOfFrame
Default value	**FG_ENDOFLINE**

Example 17.49. Usage of FG_DEBUGWRITEFLAG

```
int result = 0;
int value = FG_ENDOFLINE;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_DEBUGWRITEFLAG, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_DEBUGWRITEFLAG, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.35.9.6. FG_DEBUGREADY

This parameter is for internal testing. Please DON'T use this parameter.

Table 17.51. Parameter properties of FG_DEBUGREADY

Property	Value
Name	**FG_DEBUGREADY**
Display Name	**Debug Write Ready**
Type	**Enumeration**
Access policy	**Read-Only**
Storage policy	**Transient**
Allowed values	**FG_YES** Yes **FG_NO** No

Example 17.50. Usage of FG_DEBUGREADY

```
int result = 0;
int value = FG_NOE;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_getParameterWithType(fg, FG_DEBUGREADY, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.35.9.7. FG_DEBUG_FORCE_FRAMEID

This parameter is for internal testing. Please DON'T use this parameter.

Table 17.52. Parameter properties of FG_DEBUG_FORCE_FRAMEID

Property	Value
Name	**FG_DEBUG_FORCE_FRAMEID**
Display Name	**Debug force FrameID**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**FG_ON** On **FG_OFF** Off
Default value	**FG_OFF**

Example 17.51. Usage of FG_DEBUG_FORCE_FRAMEID

```
int result = 0;
int value = FG_OFF;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_DEBUG_FORCE_FRAMEID, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_DEBUG_FORCE_FRAMEID, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.35.9.8. FG_DEBUG_FRAMEID

This parameter is for internal testing. Please DON'T use this parameter.

Table 17.53. Parameter properties of FG_DEBUG_FRAMEID

Property	Value
Name	**FG_DEBUG_FRAMEID**
Display Name	**Debug FrameID**
Type	**Unsigned Integer**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**Minimum** 0 **Maximum** 65535 **Stepsize** 1
Default value	**0**

Example 17.52. Usage of FG_DEBUG_FRAMEID

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_setParameterWithType(fg, FG_DEBUG_FRAMEID, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_DEBUG_FRAMEID, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.35.9.9. FG_DEBUG_ENABLE_SEQUENCEID

This parameter is for internal testing. Please DON'T use this parameter.

Table 17.54. Parameter properties of FG_DEBUG_ENABLE_SEQUENCEID

Property	Value
Name	**FG_DEBUG_ENABLE_SEQUENCEID**
Display Name	**Debug enable Sequence ID**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**FG_ON** On **FG_OFF** Off
Default value	**FG_OFF**

Example 17.53. Usage of FG_DEBUG_ENABLE_SEQUENCEID

```
int result = 0;
int value = FG_OFF;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_DEBUG_ENABLE_SEQUENCEID, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_DEBUG_ENABLE_SEQUENCEID, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.35.10. Output

17.35.10.1. FG_DEBUGOUTENABLE

This parameter is for internal testing. Please DON'T use this parameter.

Table 17.55. Parameter properties of FG_DEBUGOUTENABLE

Property	Value
Name	**FG_DEBUGOUTENABLE**
Display Name	**Debug Output Mode**
Type	**Enumeration**
Access policy	**Read/Write/Change**
Storage policy	**Persistent**
Allowed values	**FG_ON** On **FG_OFF** Off
Default value	**FG_OFF**

Example 17.54. Usage of FG_DEBUGOUTENABLE

```
int result = 0;
int value = FG_OFF;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_DEBUGOUTENABLE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_DEBUGOUTENABLE, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.35.10.2. FG_DEBUGOUTXPOS

This parameter is for internal testing. Please DON'T use this parameter.

Table 17.56. Parameter properties of FG_DEBUGOUTXPOS

Property	Value
Name	**FG_DEBUGOUTXPOS**
Display Name	**Debug Output XPosition**
Type	**Unsigned Integer**
Access policy	**Read-Only**
Storage policy	**Transient**
Allowed values	**Minimum 8** **Maximum 32768** **Stepsize 1**
Unit of measure	**pixel**

Example 17.55. Usage of FG_DEBUGOUTXPOS

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_getParameterWithType(fg, FG_DEBUGOUTXPOS, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.35.10.3. FG_DEBUGOUTYPOS

This parameter is for internal testing. Please DON'T use this parameter.

Table 17.57. Parameter properties of FG_DEBUGOUTYPOS

Property	Value
Name	**FG_DEBUGOUTYPOS**
Display Name	**Debug Output YPosition**
Type	**Unsigned Integer**
Access policy	**Read-Only**
Storage policy	**Transient**
Allowed values	**Minimum 1** **Maximum 8388607** **Stepsize 1**
Unit of measure	**pixel**

Example 17.56. Usage of FG_DEBUGOUTYPOS

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_getParameterWithType(fg, FG_DEBUGOUTYPOS, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.35.10.4. FG_DEBUGOUTPIXEL

This parameter is for internal testing. Please DON'T use this parameter.

Table 17.58. Parameter properties of FG_DEBUGOUTPIXEL

Property	Value
Name	**FG_DEBUGOUTPIXEL**
Display Name	**Debug Output Pixel**
Type	**Unsigned Integer**
Access policy	**Read-Only**
Storage policy	**Transient**
Allowed values	**Minimum 0** **Maximum -1** **Stepsize 1**
Unit of measure	**pixel**

Example 17.57. Usage of FG_DEBUGOUTPIXEL

```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

if ((result = Fg_getParameterWithType(fg, FG_DEBUGOUTPIXEL, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.36. GenTL

17.36.1. FG_GENTL_INFO_VERSION

This paramter gives the version of the GenTL description used by our GenTL producer. This parameter is of internal use only.

Table 17.59. Parameter properties of FG_GENTL_INFO_VERSION

Property	Value
Name	**FG_GENTL_INFO_VERSION**
Display Name	**Version**
Type	**String**
Access policy	**Read-Only**
Storage policy	**Transient**
Unit of measure	

Example 17.58. Usage of FG_GENTL_INFO_VERSION

```
int result = 0;
char* value = "";
const enum FgParamTypes type = FG_PARAM_TYPE_CHAR_PTR;

if ((result = Fg_getParameterWithType(fg, FG_GENTL_INFO_VERSION, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.36.2. FG_GENTL_INFO_IGNOREFGFORMAT

This paramter describs the handling of outputformats in the GenTL producer. If the parameter is set to 1 the handling of Output formats is done by the producer ignoring the Outputformatsettings of the Applet.

Table 17.60. Parameter properties of FG_GENTL_INFO_IGNOREFGFORMAT

Property	Value
Name	**FG_GENTL_INFO_IGNOREFGFORMAT**
Display Name	**IgnoreFGFormat**
Type	**String**
Access policy	**Read-Only**
Storage policy	**Transient**

Example 17.59. Usage of FG_GENTL_INFO_IGNOREFGFORMAT

```
int result = 0;
char* value = "";
const enum FgParamTypes type = FG_PARAM_TYPE_CHAR_PTR;

if ((result = Fg_getParameterWithType(fg, FG_GENTL_INFO_IGNOREFGFORMAT, &value, 0, type)) < 0) {
    /* error handling */
}
```

17.36.3. FG_GENTL_INFO_OVERFLOWCAPABLE

This parameter informs the producer that the Applet is capable of extended overflow management.

Table 17.61. Parameter properties of FG_GENTL_INFO_OVERFLOWCAPABLE

Property	Value
Name	**FG_GENTL_INFO_OVERFLOWCAPABLE**
Display Name	**OverflowCapable**
Type	**String**
Access policy	**Read-Only**
Storage policy	**Transient**

Example 17.60. Usage of FG_GENTL_INFO_OVERFLOWCAPABLE

```
int result = 0;
char* value = "";
const enum FgParamTypes type = FG_PARAM_TYPE_CHAR_PTR;

if ((result = Fg_getParameterWithType(fg, FG_GENTL_INFO_OVERFLOWCAPABLE, &value, 0, type)) < 0) {
    /* error handling */
}
```

Chapter 18. Revision History

Revision history of AcquisitionApplets releases.

Applet Version	Release Date	Change Log	Delivered with
1.0.1.0	30 June 2023	Initial version of this applet.	Framegrabber SDK 5.11
1.1.1.0	22 December 2023	• CoaXPress trigger packets updated to CoaXPress 2.1: The applet now supports sending the CXP **`LinkTrigger0`**, **`LinkTrigger1`**, **`LinkTrigger2`** and **`LinkTrigger3`**. For each link trigger, an individual source can be selected. By default, **`LinkTrigger0`** is the rising edge of the trigger source and **`LinkTrigger1`** is the falling edge of the trigger, which usually represents the exposure time. The default setting is equal to previous versions. Thus, the applet is fully compatible to CXP 1.1 and 2.0. **`LinkTrigger2`** and **`LinkTrigger3`** are not used by default. The parameters *FG_CXP_TRIGGER_PACKET_MODE* (CxpTriggerPacketMode), *FG_TRIGGERCAMERA_SOURCE* (TriggerCameraSource) and *FG_TRIGGERCAMERA_POLARITY* (TriggerCameraPolarity) became legacy. • Bugfixes.	Framegrabber SDK 5.11.2

18.1. Fixed Issues

18.1.1. Fixed in Version 1.1.1.0

- Before fixing this issue, when image mirroring was activated in line applets, (i.e. *FG_VANTAGEPOINT* = TopRight or BottomRight), the actual maximum image width could have been less than documented and depended on the applet and the used pixel format. This has been fixed. (Ticket-ID: 280289)
- In microDisplay X, the *FG_LINETRIGGERINSRC* and *FG_SHAFTENCODERINSRC* parameters didn't show the correct default value so that no changes could be made. This problem did only occur in microDisplay X and has been fixed. (Ticket-ID: 290601)

18.2. Known Issues

- In rare cases, loading an applet can fail with the error message “Fg_init(...) : -2050 (Design is invalid)”. In this case, re-load the applet and contact the Basler Support [https://www.baslerweb.com/en/sales-support/support-contact/]. (Ticket-ID: 259458)

Glossary

Area of Interest (AOI)	See Region of Interest.
Board	A Basler hardware. Usually, a board is represented by a interface card. Boards might comprise multiple devices.
Board ID Number	An identification number of a Basler board in a PC system. The number is not fixed to a specific hardware but has to be unique in a PC system.
Camera Index	The index of a camera connected to a interface card. The first camera will have index zero. Mind the difference between the camera index and the interface card camera port. See also Camera Port.
Camera Port	The Basler interface card connectors for cameras are called camera ports. They are numbered {0, 1, 2, ...} or enumerated {A, B, C, ... }. Depending on the interface one camera could be connected to multiple camera ports. Also, multiple cameras could be connected to one camera port.
Camera Tap	See Tap.
Device	A board can consist of multiple devices. Devices are numbered. The first device usually has number one.
Direct Memory Access (DMA)	A DMA transfer allows hardware subsystems within the computer to access the system memory independently of the central processing unit (CPU). Basler uses DMAs for data transfer such as image data between a board e.g. a interface card and a PC. Data transfers can be established in multiple directions i.e. from a interface card to the PC (download) and from the PC to a interface card (upload). Multiple DMA channels may exist for one board. Control and configuration data usually do not use DMA channels.
DMA Channel	See DMA Index.
DMA Index	The index of a DMA transfer channel. See also Direct Memory Access.
Event	In programming or runtime environments, a callback function is a piece of executable code that is passed as an argument, which is expected to call back (execute) exactly that time an event is triggered. These events are not related to a special camera functionality and based on interface card internal functionality. Basler uses hardware interrupts for the event transfer and processing is absolutely optimized for low latency. These interrupts are only produced by the interface card if an event is registered and activated by software. If an event is fired at a very high frequency this may influence the system performance. For example these events can be used to check the reliability between a frame trigger input and the resulting and expected camera frame. The Basler Framegrabber SDK enables an application to get these event notifications about certain state changes at the data flow from camera to RAM and the image and trigger processing as well. Please consult the Basler Framegrabber SDK documentation for more details concerning the implementation of this functionality. Some events are enabled to produce additional data, which is described for the event itself.

Frame Grabber	Usually a PC hardware using PCI express to interface the camera and grab camera images. The frame grabber will grab, buffer, pre-process and forward the images to the PC memory. Moreover, the frame grabber performs the trigger signal processing to trigger the camera, external lights and controllers. On V-series frame grabber custom processing can be implemented using VisualApplets. See also Direct Memory Access, Interface Card, VisualApplets.
GenICam	Generic Interface for Cameras is a generic programming interface for machine vision (industrial) cameras.
GenTL	GenICam Transport Layer. This is the transport layer interface for enumerating cameras, grabbing images from the camera, and moving them to the user application.
Interface Card	Usually a PC hardware using PCI express to interface the camera and grab camera images. The interface card will grab, buffer and forward the images to the PC memory. Moreover, the interface card performs the trigger signal processing to trigger the camera, external lights and controllers. See also Direct Memory Access, Frame Grabber.
Port	See Camera Port.
Process	An image or signal data processing block. A process can include one or more cameras, one or more DMA channels and modules.
Region of Interest (ROI)	Represents a part of a frame. Mostly rectangular and within the original image boundaries. Defined by source coordinates and its dimension. The interface card cuts the region of interest from the camera image. A region of interest might reduce or increase the required bandwidth and the corresponding image dimension.
Sensor Tap	See Tap.
Software Callback	See Event.
Tap	Some cameras have multiple taps. This means, they can acquire or transfer more than one pixel at a time which increses the camera's acquisition speed. The camera sensor tap readout order varies. Some cameras read the pixels interlaced using multiple taps, while some cameras read the pixel simultaneously from different locations on the sensor. The reconstruction of the frame is called sensor readout correction. The Camera Link interface is also using multiple taps for image transfer to increase the bandwidth. These taps are independent from the sensor taps.
Trigger	In machine vision and image processing, a trigger is an event which causes an action. This can be for example the initiation of a new line or frame acquisition, the control of external hardware such as flash lights or actions by a software applications. Trigger events can be initiated by external sources, an internal frequency generator (timer) or software applications. The event itself is mostly based on a rising or falling edge of a electrical signal.
Trigger Input	A logic input of a trigger IO. The first input has index 0. Check mapping of input pins to logic inputs in the hardware documentation.
Trigger Output	A logic output of a trigger IO. The first output has index 1. Please check the mapping of output pins to logic outputs in the hardware documentation. The electrical characteristics and specification can be found related to the selected or used trigger board/connector.
Trigger Reliability	See Event.

User Interrupt	See Event.
VisualApplets	Simple programming of FPGA-based image processing devices. VisualApplets enables access to the FPGA processors in the image processing hardware, such as frame grabbers, industrial cameras and image processing devices, to implement individual image processing applications.

Index

G

I

L

M

O

P

R

S

T

W