# microEnable 5 marathon/LightBridge ACL

AcquisitionApplets User Documentation for

## Acq_SingleBaseAreaGray

Functional Description
For Framegrabber SDK Usage

Document Number: AW001769
Part Number: 000 (English)
Document Version: 01
Release Date: 01 December 2022
Applet Version 2.2.4.0

# 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_SingleBaseAreaGray" for microEnable 5 marathon/LightBridge ACL frame grabber.


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_SingleBaseAreaGray

"Acq_SingleBaseAreaGray" is an applet for one camera (single-camera applet). You can configure the Camera Link camera interface for Camera Link cameras in Base-Configuration mode transferring grayscale (monochrome) pixels at bit depths between 8 and 16 bits. A multi-functional area trigger is included in the applet. This allows you to control the camera or external devices using frame grabber generated, external, or software generated trigger pulses. Area scan cameras transferring images with a resolution of up to 16384 by 65535 pixels are supported. 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 frame grabber and PCs. The applet supports horizontal and vertical tap geometry sorting for 1, 2 and 4 Taps. You can mirror the acquired sensor images in x- and y-direction (horizontal and vertical mirror). To enable mirroring, set the vantage point. For reverse operation, you can mirror the image in x-direction and y-direction before cutting the ROI. Acquired images are buffered in frame grabber 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. You can configure the 12 bit full resolution lookup table either by using a user defined table, or by using a processor. The processor gives you the opportunity to use pre-defined functions such as offset, gain, invert to enhance the image quality.

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 or 16 bits per pixel.

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

**Table 1.1. Feature Summary of Acq_SingleBaseAreaGray**

| Feature | Applet Property |
|---|---|
| Applet Name |  <br>Acq_SingleBaseAreaGray |
| Type of Applet | AcquisitionApplets |
| Board | microEnable 5 marathon/LightBridge ACL |
| Minimum Framegrabber SDK Version | 5.5.1 |
| No. of Cameras | 1 |
| Camera Type | Camera Link Base Configuration (1 Tap 8 Bit, 2 Tap 8 Bit, 3 Tap 8 Bit, 1 Tap 10 Bit, 2 Tap 10 Bit, 1 Tap 12 Bit, 2 Tap 12 Bit, 1 Tap 14 Bit, 1 Tap 16 Bit) |
| Sensor Type | Area Scan |
| Camera Format | Grayscale |
| Processing Bit Depth | 16 Bit |
| Maximum Images Dimensions | 16384 * 65535 |
| ROI Stepsize | x: 4, y: 1 |
| Tap Geometry | Yes, horizontal/vertical 1, 2 and 4 Taps<br>1X-1Y, 1X2-1Y, 1X3-1Y, 1X4-1Y, 1X-2Y, 1X-2YE, 1X2-2YE, 2X-1Y, 2XE-1Y, 2XM-1Y, 2X2-1Y, 2X2E-1Y, 2X2M-1Y, 2X-2Y, 2X-2YE, 2XE-2Y, 2XE-2YE, 2XM-2Y, 2XM-2Ye, 4X-1Y |
| Mirroring | Yes, horizontal and vertical (Use vantage point) Yes, horizontal and vertical (use vantage point) |
| Image Selector | Yes |
| Noise Filter | No |
| Shading Correction | No |
| Dead Pixel Interpolation | No |
| Bayer Filter | No |
| Color White Balancing | No |
| Lookup Table | Full Resolution<br>Input bits = 12, Output bits = 16<br>Lookup table can be disabled. |
| DMA | Full Speed |
| DMA Image Output Format | Grayscale 8 or 16 bit |
| 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_SingleBaseAreaGray are listed in the following table.

**Table 1.2. Bandwidth of Acq_SingleBaseAreaGray**

| Description | Bandwidth |
|---|---|
| Max. CamClk | 85 MHz |
| Peak Bandwidth per Camera | 255 MPixel/s |
| Mean Bandwidth per Camera | 255 MPixel/s |
| DMA Bandwidth | 1800 MByte/s (depends on PC mainboard)<br><br>Approx. 1350 MB/s on LightBridge depending on cable length and PC interface. |

Max. CamClk refers to the maximum pixel clock frequency used by the the Camera Link camera. You can use any lower pixel clock frequency. Lower pixel clock frequencies enable longer CameraLink cables.

The peak bandwidth defines the maximum allowed bandwidth for each camera at the camera interface. When transferring data from the camera to the frame grabber at this bandwidth, the buffer on the frame grabber will fill up as the data can be buffered, but not be processed at that speed. The peak bandwidth is related to the Camera Link interface mode and pixel clock frequency. The product of the used Camera Link taps and the pixel frequency must not exceed the peak bandwidth.

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 1800 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 frame grabber license are listed.

### 1.3.1. Software Requirements

To run this applet, a Silicon Software 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. The minimum Basler Framegrabber SDK version you need for using this applet is version 5.5.1.

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

### 1.3.2. Hardware Requirements

To run applet "Acq_SingleBaseAreaGray", a Silicon Software microEnable 5 marathon/LightBridge ACL frame grabber is required.

For PC system requirements, check the frame grabber 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 frame grabbers can run the applet using the Basler Framegrabber SDK.

## 1.4. Camera Interface

Applet "Acq_SingleBaseAreaGray" supports one Camera Link camera in base configuration mode. The frame grabber has two Camera Link connectors. Connect the camera to port 'A' only.

**Figure 1.1. Camera Interface and Camera Cable Setup**

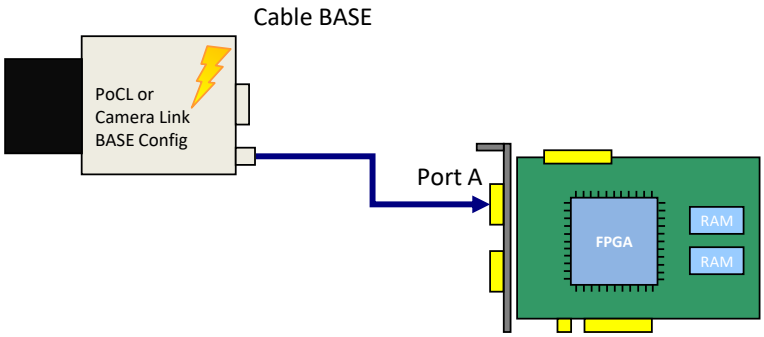
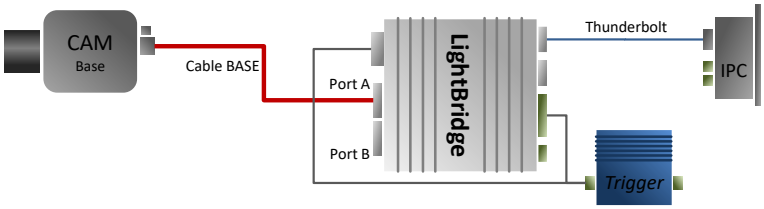



**Figure 1.2. Camera Interface and Camera Cable Setup**




## 1.5. Image Transfer to PC Memory

The image transfer between frame grabber 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 12, '*Output Format*'. All outputs are little-endian coded.

## DMA Image Tag

The applet does not generate a valid DMA image tag (**FG_IMAGE_TAG**). You may check for lost or corrupted frames using the overflow module described in Chapter 8, '*Overflow*'. The trigger system offers possibilities to detect lost trigger signals. See Chapter 7, '*Trigger*' for more information.

# 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. Camera Link

This AcquisitionApplets can be used with one area scan camera in Camera Link base configuration mode. To receive correct image data from your camera, it is crucial that the camera output format matches the selected frame grabber input format. The following parameters configure the grabber's camera interface to match with the individual camera properties. Most cameras support different operation modes. Please, consult the manual of your camera to obtain the necessary information, how to configure the camera to the desired data format.

Ensure that the images transferred by the camera do not exceed the maximum allowed image dimensions for this applet (16384 x 65535).

## 3.1. FG_CAMERA_LINK_CAMTYPE

This parameter specifies the data format of the connected camera.

This camera interface can be configured to support the different data formats specified by the Camera Link standard. In this AcquisitionApplets,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 Link bit depth is greater than the processing bit depth, bits will be right shifted to meet the internal bit depth. If the Camera Link 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_CAMERA_LINK_CAMTYPE**

| Property | Value | |
|---|---|---|
| Name | **FG_CAMERA_LINK_CAMTYPE** | |
| Display Name | **Input Format** | |
| Type | **Enumeration** | |
| Access policy | **Read/Write/Change** | |
| Storage policy | **Persistent** | |
| Allowed values | **FG_CL_SINGLETAP_8_BIT** | Single Tap 8bit |
| | **FG_CL_SINGLETAP_10_BIT** | Single Tap 10bit |
| | **FG_CL_SINGLETAP_12_BIT** | Single Tap 12bit |
| | **FG_CL_SINGLETAP_14_BIT** | Single Tap 14bit |
| | **FG_CL_SINGLETAP_16_BIT** | Single Tap 16bit |
| | **FG_CL_DUALTAP_8_BIT** | Dual Tap 8bit |
| | **FG_CL_DUALTAP_10_BIT** | Dual Tap 10bit |
| | **FG_CL_DUALTAP_12_BIT** | Dual Tap 12bit |
| | **FG_CL_TRIPLETAP_8_BIT** | Triple Tap 8bit |
| Default value | **FG_CL_DUALTAP_8_BIT** | |

**Example 3.1. Usage of FG_CAMERA_LINK_CAMTYPE**

```
int result = 0;
int value = FG_CL_DUALTAP_8_BIT;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

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

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

## 3.2. FG_USEDVAL

With this parameter it is possible to support cameras that do not fully comply with the Camera Link specification. If **FG_YES** is selected, DVAL, LVAL and FVAL is used to decode valid pixels. If the parameter is set to **FG_NO**, only LVAL and FVAL is used to decode valid pixels. If you are not sure about the required setting, keep the parameter's value at **FG_YES**.

**Table 3.2. Parameter properties of FG_USEDVAL**

| Property | Value |
|---|---|
| Name | **FG_USEDVAL** |
| Display Name | **Use DVAL** |
| Type | **Enumeration** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **FG_YES** Yes<br>**FG_NO** No |
| Default value | **FG_YES** |

**Example 3.2. Usage of FG_USEDVAL**

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

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

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

## 3.3. FG_CAMERA_LINK_CORE_RESET

This parameter allows to re-initialize the Camera Link core. Under normal circumstances, re-initializing the Camera Link core is not necessary. This parameter is exclusively for use by the Silicon Software Support department. Value range: [0;1]

1 = Enforces a reset of the Camera Link core.

0 = Resets the reset.

**Table 3.3. Parameter properties of FG_CAMERA_LINK_CORE_RESET**

| Property | Value |
|---|---|
| Name | **FG_CAMERA_LINK_CORE_RESET** |
| Display Name | **Camera Link Core Reset** |
| Type | **Unsigned Integer** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Transient** |
| Allowed values | **Minimum** 0<br>**Maximum** 1<br>**Stepsize** 1 |
| Default value | **0** |
| Unit of measure | |

**Example 3.3. Usage of FG_CAMERA_LINK_CORE_RESET**

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

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

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

## 3.4. FG_CAMERA_LINK_PIXEL_CLOCK

This parameter displays the value of the pixel clock (in MHz) as meassured on the Camera Link channel. This is a read only parameter. The value range is [0;85].

**Table 3.4. Parameter properties of FG_CAMERA_LINK_PIXEL_CLOCK**

| Property | Value |
|---|---|
| Name | **FG_CAMERA_LINK_PIXEL_CLOCK** |
| Display Name | **Pixel Clock** |
| Type | **Unsigned Integer** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **Minimum 0**<br>**Maximum 85**<br>**Stepsize 1** |
| Unit of measure | **MHz** |

**Example 3.4. Usage of FG_CAMERA_LINK_PIXEL_CLOCK**

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

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

# Chapter 4. Camera

This applet Acq_SingleBaseAreaGray for the microEnable 5 marathon/LightBridge ACL acquires the sensor data of an area 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

This event is generated when the first pixel of one camera frame arrives at the applet. Except for the timestamp, the event has no additional data included. Keep in mind that a high framerate can cause high interrupt rates which might slow down the overall PC system. This event can only occur if the acquisition is running.

### 4.1.2. FG_END_OF_FRAME_CAM_PORT_0

This event is generated right after the last pixel of one camera frame arrives at the applet. Except for the timestamp, the event has no additional data included. Keep in mind that a high framerate can cause high interrupt rates which might slow down the overall PC system. This event can only occur if the acquisition is running.

# Chapter 5. Tap Geometry

## 5.1. FG_TAPGEOMETRY

Define the tap geometry with this parameter. The parameter follows the naming defined in the GenICam SFNC (Standard Features Naming Convention).

**Table 5.1. Parameter properties of FG_TAPGEOMETRY**

| Property | Value |
|---|---|
| Name | **FG_TAPGEOMETRY** |
| Display Name | **Tap Geometry** |
| Type | **Enumeration** |
| Access policy | **Read/Write** |
| Storage policy | **Persistent** |
| Allowed values | **FG_GEOMETRY_1X_1Y** 1X-1Y<br>**FG_GEOMETRY_1X_2Y** 1X-2Y<br>**FG_GEOMETRY_1X_2YE** 1X-2YE<br>**FG_GEOMETRY_1X2_1Y** 1X2-1Y<br>**FG_GEOMETRY_1X2_2YE** 1X2-2YE<br>**FG_GEOMETRY_1X3_1Y** 1X3-1Y<br>**FG_GEOMETRY_1X4_1Y** 1X4-1Y<br>**FG_GEOMETRY_2X_1Y** 2X-1Y<br>**FG_GEOMETRY_2XE_1Y** 2XE-1Y<br>**FG_GEOMETRY_2XM_1Y** 2XM-1Y<br>**FG_GEOMETRY_2X_2Y** 2X-2Y<br>**FG_GEOMETRY_2X_2YE** 2X-2YE<br>**FG_GEOMETRY_2XE_2Y** 2XE-2Y<br>**FG_GEOMETRY_2XE_2YE** 2XE-2YE<br>**FG_GEOMETRY_2XM_2Y** 2XM-2Y<br>**FG_GEOMETRY_2XM_2YE** 2XM-2YE<br>**FG_GEOMETRY_2X2_1Y** 2X2-1Y<br>**FG_GEOMETRY_2X2E_1Y** 2X2E-1Y<br>**FG_GEOMETRY_2X2M_1Y** 2X2M-1Y<br>**FG_GEOMETRY_4X_1Y** 4X-1Y |
| Default value | **FG_GEOMETRY_1X_1Y** |

**Example 5.1. Usage of FG_TAPGEOMETRY**

```
int result = 0;
int value = FG_GEOMETRY_1X_1Y;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

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

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

## 5.2. FG_SENSORWIDTH

For tap geometry sorting, the applet needs to be parameterized with the exact sensor width that is transfered from the camera to the frame grabber. If you have set a region of interest (ROI) in the camera, the parameter *FG_SENSORWIDTH* needs to be set to the ROI size. Note that this value has to be a multiple of the taps. For example, if you are using a 10 tap camera, *FG_SENSORWIDTH* will be, for example, 1020 pixels. A value of 1024 is therefore not possible in this example.

### The Value of *FG_SENSORWIDTH* Is Not Used, If Only One *x-zone* Is Used and Mirroring Is Not Active

If the tap geometry is configured to one *x-zone* only and no mirroring is configured, 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**<br>**Maximum 16384**<br>**Stepsize 1** |
| 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

## 5.3. FG_SENSORHEIGHT

For tap geometry sorting, the applet needs to be parameterized with the exact sensor height transfered 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.

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

If the tap geometry is configured to one *y-zone* only and 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<br>**Maximum** 65535<br>**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 */
}
```

## 5.4. 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.

**Table 5.4. 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<br>**FG_VANTAGEPOINT_TOP_RIGHT** Top-Right<br>**FG_VANTAGEPOINT_BOTTOM_LEFT** Bottom-Left<br>**FG_VANTAGEPOINT_BOTTOM_RIGHT** Bottom-Right |
| Default value | **FG_VANTAGEPOINT_TOP_LEFT** |

**Example 5.4. Usage of FG_VANTAGEPOINT**

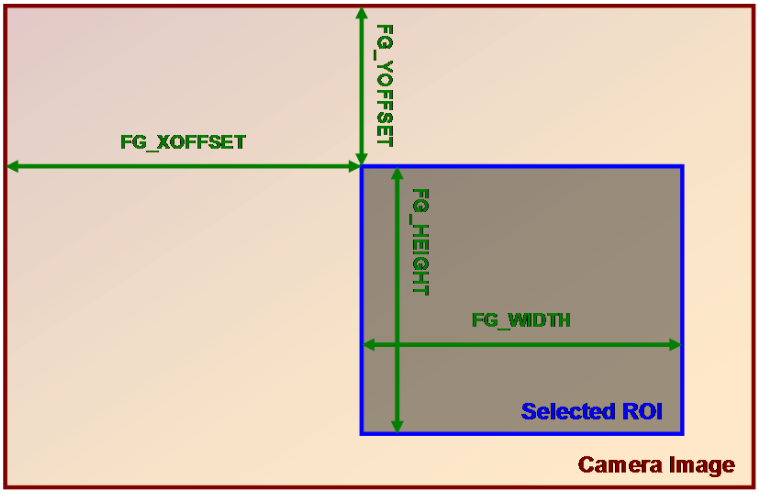
```
int result = 0;
int value = FG_VANTAGEPOINT_TOP_LEFT;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;
```

```
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 */
}
```

# Chapter 6. ROI

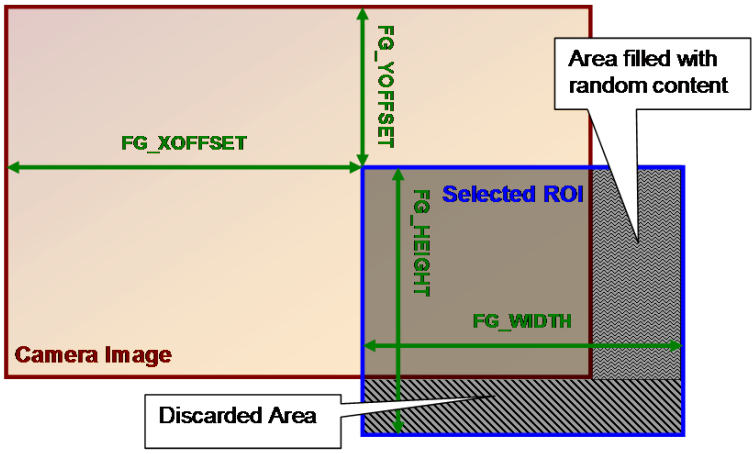
This module allows the definition of a region of interest (ROI), also called area of interest (AOI). A ROI allows the selection of a smaller subset pixel area from the input image. It is defined by using parameters *FG_XOFFSET*, *FG_WIDTH*, *FG_YOFFSET* and *FG_HEIGHT*. The following figure illustrates 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 16384 pixels and a maximum image height of 65535 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 frame grabber.

## 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 16384 pixels.

**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**<br>**Maximum 16384**<br>**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 65535 pixels.

**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**<br>**Maximum 65535**<br>**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.

**Table 6.3. Parameter properties of FG_XOFFSET**

| Property | Value |
|---|---|
| Name | **FG_XOFFSET** |
| Display Name | **XOffset** |
| Type | **Unsigned Integer** |
| Access policy | **Read/Write** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum 0**<br>**Maximum 16376**<br>**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.

**Table 6.4. Parameter properties of FG_YOFFSET**

| Property | Value |
|---|---|
| Name | **FG_YOFFSET** |
| Display Name | **YOffset** |
| Type | **Unsigned Integer** |
| Access policy | **Read/Write** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum 0**<br>**Maximum 65535**<br>**Stepsize 1** |
| Default value | **0** |
| Unit of measure | **pixel** |

**Example 6.4. Usage of FG_YOFFSET**

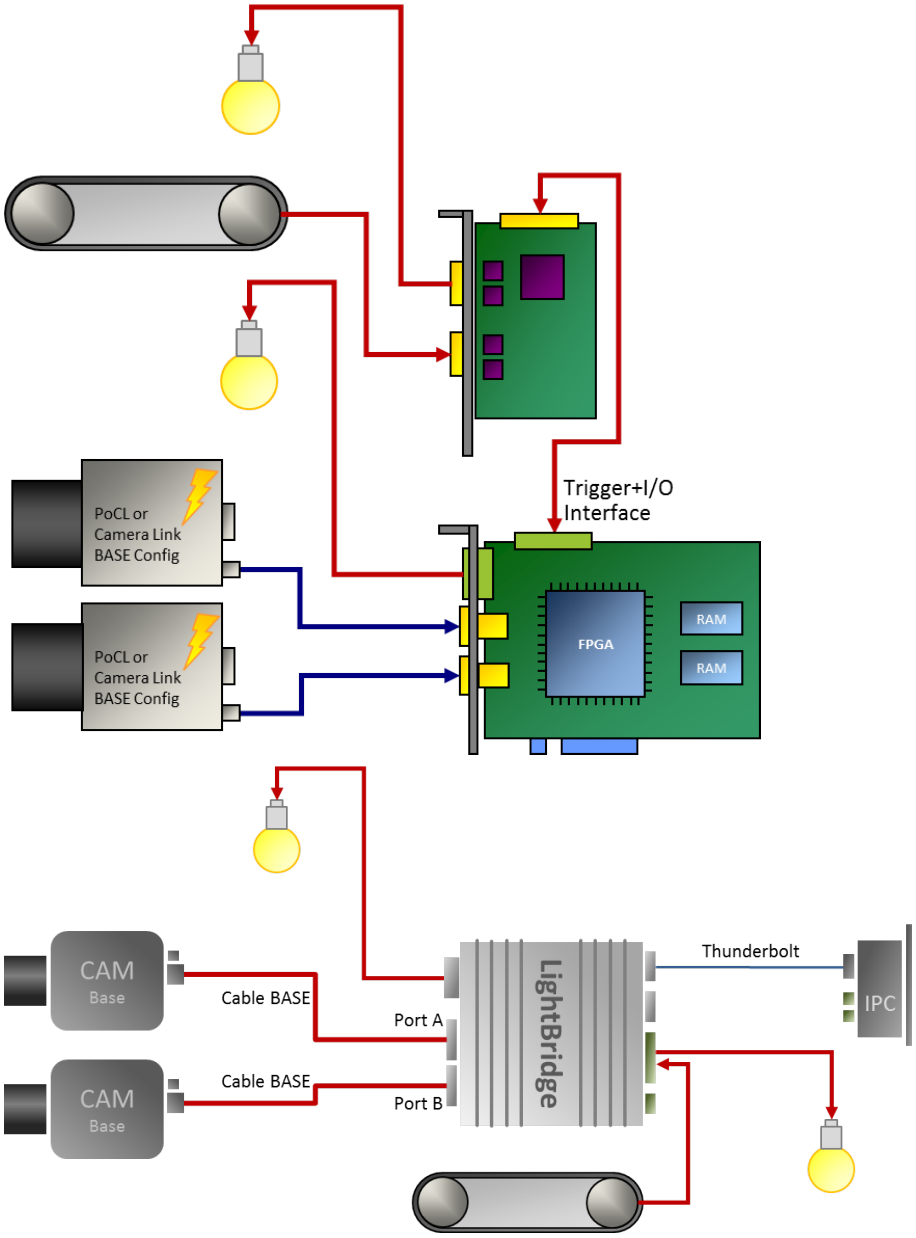
```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

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. Trigger

The area trigger system enables the control of the image acquisition process of the frame grabber and the connected camera. In detail it controls the exact exposure time of the camera and controls external devices. The trigger source can be external devices, internal frequency generators or the user's software application.

The microEnable 5 marathon/LightBridge ACL frame grabber has eight input IOs available on its extension port plus 4 inputs on the front IO connector. The extension port can be connected to external devices. Check the hardware documentation for more information. The microEnable 5 marathon/LightBridge ACL generates the desired trigger outputs and control signals from the input events according to the trigger system's parameterization. The trigger system outputs can be routed to the camera via the CameraLink's CC signals and additionally, outputs can be routed to the digital outputs for control of external devices such as flash lights, for synchronizing or for debugging.

**Figure 7.1. microEnable 5 marathon/LightBridge ACL Trigger System**




In the following an introduction into the Silicon Software microEnable 5 marathon/LightBridge ACL trigger system is presented. Several trigger scenarios will show the possibilities and functionalities and will help to

understand the trigger system. The documentation includes the parameter reference where all parameters of the trigger system are listed and their functionality is explained in detail.

## 7.1. Features and Functional Blocks of Area Trigger

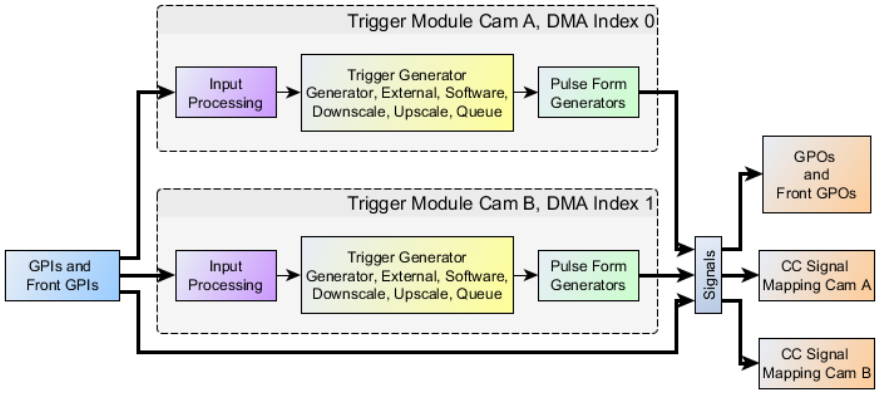
The Silicon Software trigger system was designed to fulfill the requirements of various applications. Powerful features for trigger generation, controlling and monitoring were included in the implementation. This includes:

- Trigger signal generation for cameras and external devices.
- **External devices** such as encoders and light barriers can be used to source the trigger system and control the trigger signal generation.
- In alternative an internal **frequency generator** can be used to generate trigger pulses.
- The trigger signal generation can be fully controlled by **software** . Single pulses or sequences of pulses can be generated. The trigger system will automatically control and limit the output frequency.
- Input **signal monitoring** .
- Input signal **frequency analysis** and **pulse counting** .
- Input signal **debouncing**
- Input signal **downscaling**
- **Pulse multiplication** using a sequencer and controllable maximum output frequency. Make up to 65,000 output pulses out of a single input pulse.
- **Trigger pulse queue** for buffering up to 2000 pulses and control the output using a maximum frequency valve.
- Four **pulse form generators** for individual controlling of pulse widths, delays and output downscaling.
- **Up to 10 outputs depending on the frame grabber type** plus the CameraLink Control outputs (CC-outputs).
- A **bypass** option to keep the pulse forms of the input signals and forward them to outputs and cameras.
- **Event generation** for input and output monitoring by application software.
- Trigger state events for fill level monitoring, trigger busy states and lost trigger signals give full control of the system.
- Camera **frame loss notification** .
- Full **trigger signal reliability** and easy error detections.

The trigger system is controlled and configured using parameters. Several read only parameters return status information on the current trigger state. Moreover, the trigger system is capable of generating events for efficient monitoring and controlling of the trigger system, the software, the frame grabber and external hardware.

The complex trigger system can be easily used and parameterized. The following block diagram figure shows an overview of the trigger system. As can be seen, the trigger system consists of four different main functional blocks.

**Figure 7.2. Trigger System**

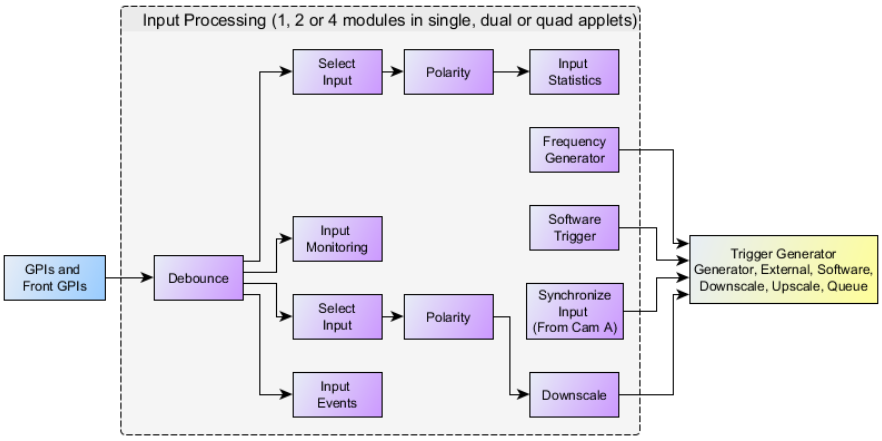



1. Trigger Input:

   Trigger inputs can be external signals, as well as software generated inputs and the frequency generator. An input monitoring and input statistics module allows analysis if the input signals.

   External input signals are debounced and split into several paths for monitoring, and further processing.

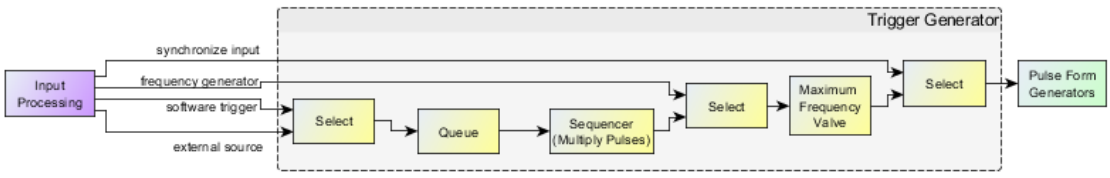
**Figure 7.3. Trigger Input Block Diagram**




2. Input Pulse Processing:

   The second main block of the trigger system is the Input Pulse Processing. External inputs as well as software trigger generated pulses can be queued and multiplied in a sequencer if desired. All external trigger pulses are processed in a maximum frequency valve. Pulses are only processed by this valve if their frequency is higher than the previously parameterized limit. If a higher frequency is present at the input, pulses will be rejected or the trigger pulse queue is filled if activated. The maximum frequency valve ensures that the output-pulses will not exceed the maximum possible frequency which can be processed by the camera.

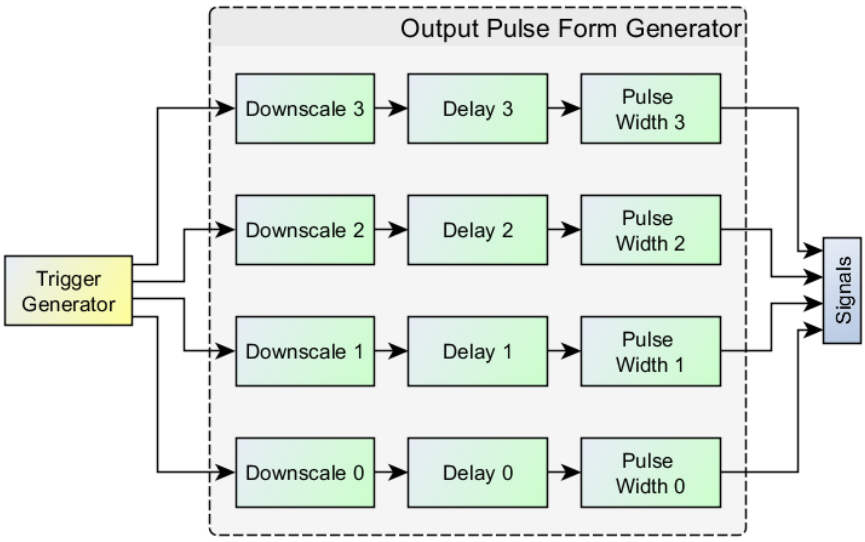
**Figure 7.4. Trigger Pulse Processing Block Diagram**




3. Output Pulse Form Generators:

After the input pulses have been processed, they are feed into four optional pulse form generators. These pulse form generators define the signal width, a delay and a possible downscale. The four pulse form generators can arbitrarily allocated to the outputs which makes the trigger system capable for numerous applications such as muliple flash light control, varying camera exposure times etc.

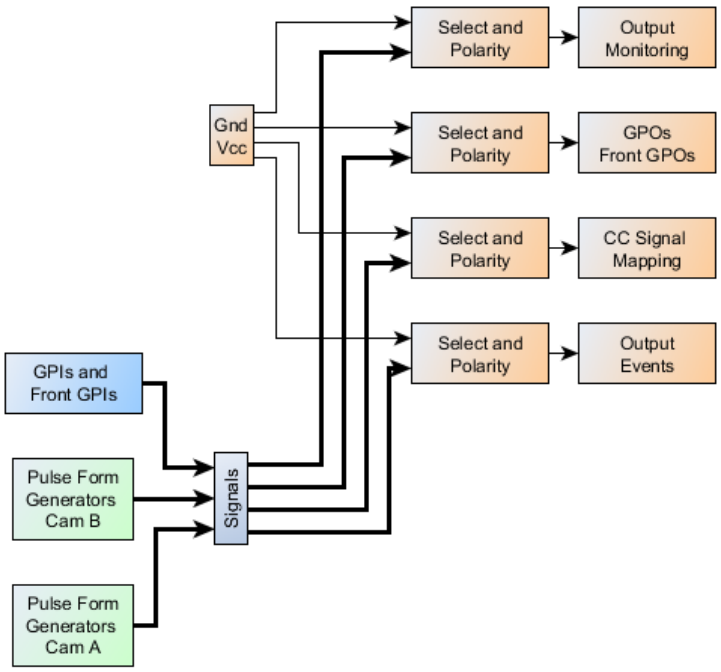
**Figure 7.5. Trigger Pulse Processing Block Diagram**




4. Trigger Output:

The last block is related to the trigger outputs. The pulse form generator signals can be output at the digital outputs and directly to the camera. Moreover, they can be monitored using registers and events.

**Figure 7.6. Trigger Output Block Diagram**




## 7.2. Digital Input/Output Mapping

The marathon frame grabbers support eight digital extension plus 4 front inputs. It has eigth extension and two front trigger outputs. In contrast the LightBridge frame grabbers support four digital extension plus 4 front inputs and has four extension and two front trigger outputs. The marathon extension port can be connected to a Silicon Software IO module using 34 pin flat cable.

The eight extension inputs have the indices 0 to 7 while the front inputs are number 0 to 3 using a different parameter name. In the documentation of the trigger IO boards and microEnable 5 marathon/LightBridge ACL frame grabber the allocation of these inputs to pins is described.

The available outputs can arbitrarily allocated to a trigger module or directly to a GPI.. See Section 7.5.12, 'Digital Output' for explanation.

## 7.3. Event Overview

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.

For a general explanation on events see Event.

In the following, a list of all events of the trigger system is presented. Detailed explanations can be found in the respective module descriptions.

- *FG_TRIGGER_INPUT0_RISING*, *FG_TRIGGER_INPUT0_FALLING* to *FG_TRIGGER_INPUT7_RISING*, *FG_TRIGGER_INPUT7_FALLING* and *FG_TRIGGER_FRONT_GPI0_RISING*, *FG_TRIGGER_FRONT_GPI0_FALLING* to *FG_TRIGGER_FRONT_GPI3_RISING*, *FG_TRIGGER_FRONT_GPI3_FALLING*

  Trigger input events. Events can be generated for all digital trigger inputs. The events are triggered by either rising or falling signal edges.

- *FG_TRIGGER_FRONT_GPI0_RISING*, *FG_TRIGGER_FRONT_GPI0_FALLING* to *FG_TRIGGER_FRONT_GPI3_RISING*, *FG_TRIGGER_FRONT_GPI3_FALLING*

  Trigger input events on the Front GPIs. Events can be generated for all digital trigger inputs. The events are triggered by either rising or falling signal edges.

- *FG_TRIGGER_EXCEEDED_PERIOD_LIMITS_CAM0*

  The event is generated for each lost input trigger pulse.

- *FG_TRIGGER_QUEUE_FILLLEVEL_THRESHOLD_CAM0_ON* and *FG_TRIGGER_QUEUE_FILLLEVEL_THRESHOLD_CAM0_OFF*

  The trigger queue exceeded the upper "on" threshold or got less than the "off" threshold.

- *FG_TRIGGER_OUTPUT_CAM0*

  Events for trigger output.

- *FG_MISSING_CAM0_FRAME_RESPONSE*

  The event is generated for a missing camera frame response.

## 7.4. Trigger Scenarios

In the following, trigger sample scenarios are presented. These scenarios will help you to use the trigger system and facilitate easy adaptation to own requirements.

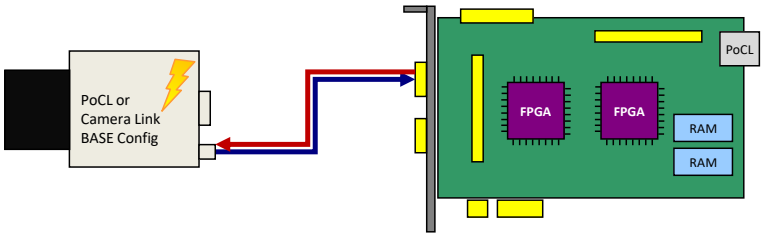
The scenarios show real life configurations. They explain the requirements, illustrate the inputs and outputs and list the required parameters and their values.

### 7.4.1. Internal Frequency Generator / frame grabber Controlled

Let's start the trigger system examples with a simple scenario. In this case we simply want to control the frequency of the camera's image output and the exposure time with the frame grabber. Assume that there is no additional external source for trigger events and we do not need to control any flash lights. Thus the frame grabber's trigger system has to control the frequency of the trigger pulses and the exposure time.

Figure 7.7 shows the hardware setup. Only the camera connected to the frame grabber is required.

**Figure 7.7. Generator Controlled Trigger Scenario**

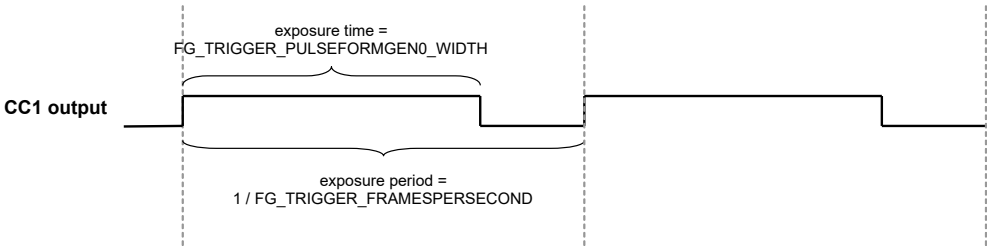



To put this scenario into practice, you will need to set your camera into an external trigger mode. Consult the vendor's user manual for more information. In general Camera Link cameras are configured, using a serial interface. The frame grabber provides this interface and will forward all commands to the camera via the CameraLink cable. Some camera manufacturers provide software tools for camera configuration. Other cameras are configured using command line input. Use the Silicon Software tool clshell in these cases.

After the camera is set to an external trigger mode, the exposure period and the exposure time can be controlled by one of the camera control inputs. For Camera Link cameras mostly this is input CC1. The names of the camera trigger modes vary. You will need to use an external trigger mode, where the exposure period is programmable. If you also want to define the exposure time using the frame grabber, the respective trigger mode needs to support this, too.

In the following, a waveform is shown which illustrates the frame grabber trigger output. Most cameras will start the acquisition on the rising or falling edge of the signal. The exposure time is defined by the length of the signal. Note that some cameras use inverted inputs. In this case, the signal has to be 'low active' instead of being 'high active'. Thus the frame grabber output has to be inverted which is explained later on.

**Figure 7.8. Waveform of Generator Controlled Trigger Scenario**




After hardware setup and camera configuration we can start parameterizing the frame grabber's trigger system. Parameters can be changed in your own application, in microDisplay or by editing a microEnable Configuration File (mcf). For more information on how to parameterize applets, consult the Basler Framegrabber SDK documentation.

In the following, all required parameters and their values are listed.

- *FG_AREATRIGGERMODE* = **ATM_GENERATOR**

First, we will need to configure the trigger system to use the internal frequency generator.

- *FG_TRIGGER_FRAMESPERSECOND* = 10

  Next, the output frequency is defined. In this example, we use a frequency of 10Hz.

- *FG_TRIGGER_PULSEFORMGEN0_WIDTH* = 200

  So far, we have set the trigger system to generate trigger pulses at a rate of 10Hz. However, we have not set the pulse form of these pulses i.e. the signal length or signal width. The frame grabber's trigger system includes four pulse form generators which allow to set the signal width, a delay and a downscaling. In our example, we only have one output and therefore, we will need only one pulse form generator, respectively pulse form generator 0. Moreover, only the signal length has to be defined, a delay and a downscaling is not required.

  Suppose, that we require an exposure time of 200µs. Thus, we will set the parameter to value 200 since the unit is µs.

- *FG_TRIGGERCC_SELECT0* = **CC_PULSEGEN0**

  The only thing left to do is to allocate the output of pulse form generator 0 to output CC1. If your camera requires low active signals, choose **CC_NOT_PULSEGEN0** instead.

Now, the trigger is fully configured. However the trigger signal generation is not started yet. Set parameter *FG_TRIGGERSTATE* to **TS_ACTIVE** to start the system. Of course, you will also need to start your image acquisition. It is up to you if you like to start the trigger generation prior or after the acquisition has been started. If the trigger system is started first, the camera will already send images to the frame grabber. These images are discarded as no acquisition is started.

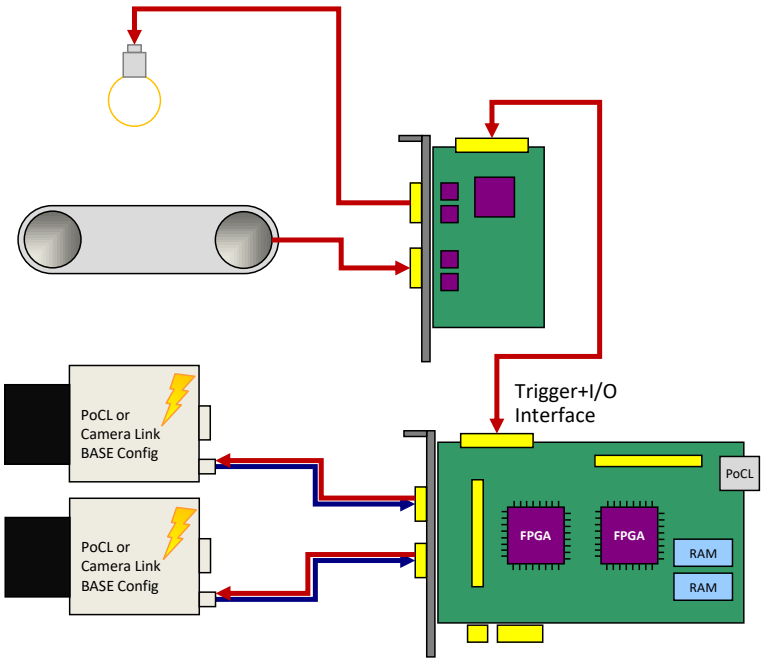
You will now receive images from your camera. Change the frequency and the signal width to see the influence of these parameters. A higher frequency will give you a higher frame rater. A shorter exposure time will make the images 'darker'. You will realize, that it is not possible to set an exposure time which is longer than the exposure period. In this case, writing to the parameter will result in an error. Therefore, the order of changing parameter values might be of importance. Also be careful to not select a frequency or exposure time which exceeds the camera's specifications. In this cases you will loose trigger pulses, as the camera cannot progress them. Get the maximum ranges from the camera's specification sheets.

To stop the trigger pulse generation, set parameter *FG_TRIGGERSTATE* to **TS_SYNC_STOP**. The trigger system will then finalize the current pulse and stop any further output until the system is activated again. The asynchronous stop mode is not required in this scenario.

### 7.4.2. External Trigger Signals / IO Triggered

In the previous example we used an internal frequency generator to control the camera's exposure. In this scenario, an external source will define the exact moment of exposure. This can be, for example, a light barrier as illustrated in the following figure. Objects move in front of the camera, a light barrier will define the moment, when an object is located directly under the camera. In practice, it might not be possible to locate the light barrier and the camera at the exact position. Therefore, a delay is required which delays the pulses from the light barrier before using them to trigger the camera. Moreover, in our scenario, we assume that a flash light has to be controlled by the trigger system, too.

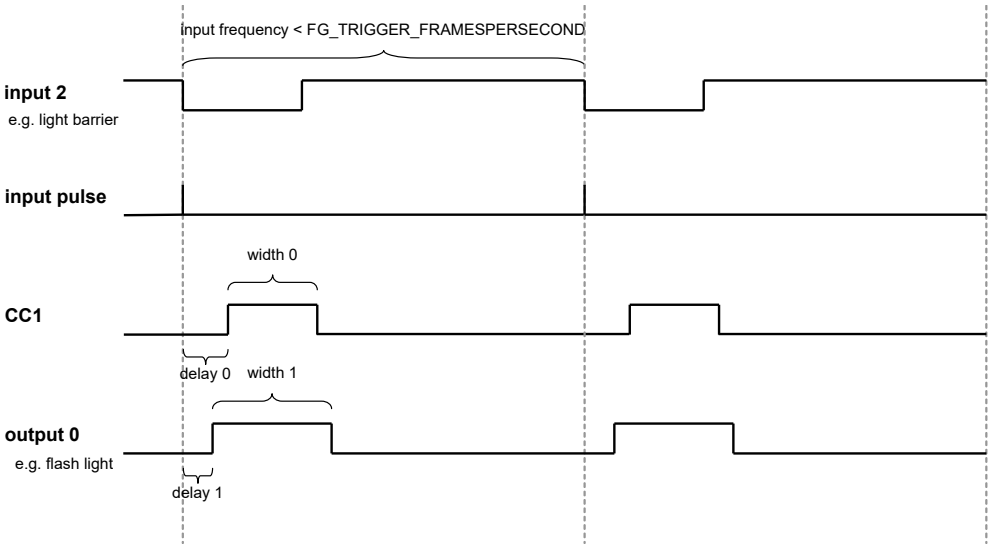
**Figure 7.9. External Controlled Trigger Scenario**




An exemplary waveform (Figure 7.10) provides information on the input signal and shows the desired output signals. The input is shown on top. As you can see, the falling edge of the signal defines the moment which is used for trigger generation. Thus, the signal is 'low active'. Mind that the pulse length of any external input is ignored (second row), only falling edges are considered.

The output to the camera is shown in the third row. Here we can see an inserted delay. This delay will compensate the positions of the light barrier and the camera. The signal width at output CC1 defines the exposure time, if the camera is configured to the respective trigger mode. Control of the flash light is done using trigger output 0. Again, a delay is added. Depending on the requirements of the flash light, this delay has to be shorter or longer than the CC1 output delay. Similarly, the required pulse length varies for different hardware.

**Figure 7.10. Waveform of External Trigger Scenario**




Before parameterizing the applet, ensure that your camera has been set to an external trigger mode. Check the previous trigger scenario for more explanations.

In this example, we have to parameterize the trigger mode, the input source and we have to configure two trigger outputs.

- *FG_AREATRIGGERMODE* = **ATM_EXTERNAL**

In external trigger mode, the trigger system will not use the internal frequency generator. External pulses control the output of trigger signals. This requires the selection of an input source and the configuration of the input polarity.

- *FG_TRIGGERIN_SRC* = **TRGINSRC_GPI_2**

  Select the trigger input by use of this parameter. You can choose any of the inputs. If you use a multi-camera applet, cameras can share same sources.

- *FG_TRIGGERIN_POLARITY* = **LOW_ACTIVE**

  For the given scenario, we assume that a trigger is required on a falling edge of the input signal.

- *FG_TRIGGER_FRAMESPERSECOND* = 500

  Do not forget to set this parameter. For any use of the trigger system, the correct parameterization of this parameter is required. If you do not use the internal frequency generator, this parameter defines the maximum allowed trigger pulse frequency. In other words, you can set a limit with this parameter. The limiting frequency could be the maximum exposure frequency of the camera.

  **The advantage of setting this limit is the information on lost trigger signals.** Let's suppose the frequency of the external trigger signals will get to high for the camera or the applet. In this case, you will loose images or obtain corrupted images. If you have set a correct frequency limit in the trigger system, the trigger system will provide you with information of these exceeding line periods. This information can be obtained by register polling or you can use the event system. Thus you always have the possibility to prevent your application of getting into a bad, probably undefined state and you will always get the information of when and how many pulses got lost. Check the explanations of parameters *FG_TRIGGER_FRAMESPERSECOND* and *FG_TRIGGER_EXCEEDED_PERIOD_LIMITS* as well as the event *FG_TRIGGER_EXCEEDED_PERIOD_LIMITS_CAM0* for more information.

  More information on error detection and analysis can be found in scenario Section 7.4.9, 'Hardware System Analysis and Error Detection / Trigger Debugging'

  The trigger system also allows the queuing of trigger pulses if you have a short period of excess pulses. We will have a look at this in a later scenario.

  In our example, we set the maximum frequency to 500 frames per second. If you do not want to use this feature, set *FG_TRIGGER_FRAMESPERSECOND* to a high value, such as 1MHz.

- *FG_TRIGGER_PULSEFORMGEN0_WIDTH* = 200

  So far, we have set the trigger system to accept external signals and generate the trigger pulses out of these signals. Next, we need to output these pulses. For realization, we need to define the pulse form of the output signals. Just as shown in the previous scenario, we use pulse form generator 0 for generating the pulse form of the trigger signals. We set a pulse width of 200µs.

- *FG_TRIGGER_PULSEFORMGEN0_DELAY* = 50

  In addition to the signal width, a delay will give us the possibility to delay the output as the light barrier might not be positioned at the exact location. For this fictitious scenario we use a delay of 50µs.

- *FG_TRIGGER_PULSEFORMGEN1_WIDTH* = 250

  In addition to the trigger output we want to control a flash light. We use pulse form generator 1 for this purpose and set the signal width to 250µs.

- *FG_TRIGGER_PULSEFORMGEN1_DELAY* = 25

  A delay for the flash output is set, too.

- *FG_TRIGGERCC_SELECT0* = **CC_PULSEGEN0**

Finally, we have to allocate output CC1 with the pulse form generator 0.

- *FG_TRIGGEROUT_SELECT_FRONT_GPO_0* = **PULSEGEN1**

  The flash light, connected to output 0 has to be allocated to pulse form generator 1.

- *FG_TRIGGEROUT_SELECT_FRONT_GPO_1* = **PULSEGEN0**

  Let's assume that it is necessary to measure the camera trigger output using a logic analyzer. Hence, we allocate output 1 to pulse form generator 0 as well.

The trigger is now fully configured. Just as described in the previous scenario, you can now start the acquisition and activate the trigger system using parameter *FG_TRIGGERSTATE*.

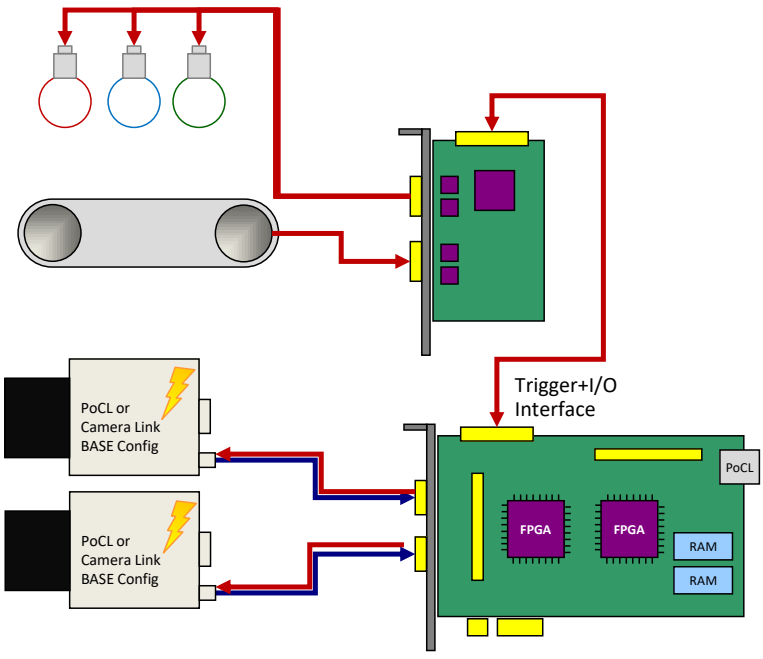
You will now receive images from the camera for each external trigger pulse. Compare the number of external pulses with the generated trigger signals and the received images for verification. Use parameter *FG_TRIGGERIN_STATS_PULSECOUNT* of category Trigger Input -> Input Statistics and parameter *FG_TRIGGEROUT_STATS_PULSECOUNT* of the output statistics parameters to get the number of input pulses and generated pulses. You can compare these values with the received image numbers.

### 7.4.3. Control of Three Flash Lights

This scenario is similar to the previous one. We use an external trigger to control the camera and a flash light. But in difference, we want to get three images from one external trigger pulse. Each image out of the sequence of three images has to use a different light source. Thus, in this scenario we will learn on how to use a trigger pulse multiplication and on how to control three lights connected to the frame grabber.

The application idea behind this scenario is that an object is acquired using different light sources. This could result in a HDR image or switching between normal and infrared illumination. The following figure illustrates the hardware setup. As you can see, we have three light sources this time. The objects move in front of the camera. The light barrier will provide the information on when to trigger the camera. Let's suppose that the objects stop in front of the camera or the movement is slow enough to generate three images with the different illuminations.

**Figure 7.11. External Controlled Trigger Scenario**



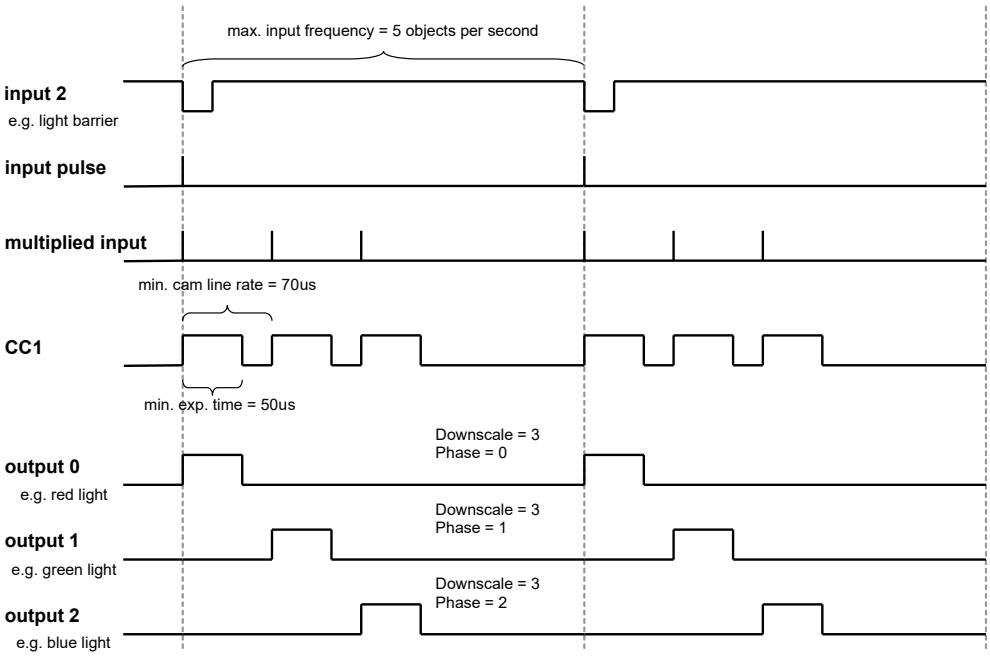

Before looking at the waveform, let's have a look at our fictitious hardware specifications.

**Table 7.1. Fictitious Hardware Specifications of Trigger Scenario Three Light Sources**

| Element | Limit |
|---|---|
| Object Speed | Max. 100 Objects per Second |
| Minimum Camera Exposure Time | 50µs |
| Minimum Camera Frame Period | 70µs |

The object speed is 100 objects per second. The minimum camera exposure time is 50µs at a minimum camera frame period of 70µs. Thus we only need 210µs to acquire the three images. The following waveform shows the input and output signals, as well as the multiplied input signals. The first row shows the input. Each falling edge represents the light barrier event as marked in the second row. The third row shows the multiplied input pulses with a gap of 70µs between the pulses. The trigger signal is generated for each of these pulses, however the trigger flash outputs 0, 1 and 2 are downscaled by three and a delay is added.

**Figure 7.12. Waveform of External Trigger Scenario Controlling Three Flash Lights**




Parameterization is similar to the previous example. In contrast, this time, we have to set the trigger pulse sequencer using a multiplication factor and we have to use the pulse form generators.

- *FG_AREATRIGGERMODE* = **ATM_EXTERNAL**
- *FG_TRIGGERIN_SRC* = 2
- *FG_TRIGGERIN_POLARITY* = **LOW_ACTIVE**
- *FG_TRIGGER_MULTIPLY_PULSES* = 3

  The parameter specifies the multiplication factor of the sequencer. For each input pulse, we have to generate three internal pulses. The period time of this multiplication is defined by parameter *FG_TRIGGER_FRAMESPERSECOND*

- *FG_TRIGGER_FRAMESPERSECOND* = 14285

  This time, the maximum frames per second correspond to the gap between the multiplied trigger pulses. We need a gap of 70µs which results in a frequency of 14285Hz.

- *FG_TRIGGER_PULSEFORMGEN0_WIDTH* = 50

  Again, we use pulse form generator 0 for trigger signal generation. The pulse width is 50µs. A delay or downscaling is not required.

- *FG_TRIGGER_PULSEFORMGEN1_WIDTH* = 50

  The pulse width for the flash lights depends on the hardware used. We assume a width of 50µs in this example.

- *FG_TRIGGER_PULSEFORMGEN2_WIDTH* = 50
- *FG_TRIGGER_PULSEFORMGEN3_WIDTH* = 50
- *FG_TRIGGER_PULSEFORMGEN1_DOWNSCALE* = 3

  The flash outputs need a downscale of three. This is the same for all flash pulse form generators.

- *FG_TRIGGER_PULSEFORMGEN2_DOWNSCALE* = 3
- *FG_TRIGGER_PULSEFORMGEN3_DOWNSCALE* = 3
- *FG_TRIGGER_PULSEFORMGEN1_DOWNSCALE_PHASE* = 0

  We use the phase shift for delaying the downscaled signals of the outputs. You could use the delay instead, but any frequency change will require a change of the delay as well. The phase shift of pulse form generator 1 i.e. the first flash light is 0.

- *FG_TRIGGER_PULSEFORMGEN2_DOWNSCALE_PHASE* = 1

  The phase shift of pulse form generator 2 i.e. the second flash light is 1.

- *FG_TRIGGER_PULSEFORMGEN3_DOWNSCALE_PHASE* = 2

  The phase shift of pulse form generator 3 i.e. the third flash light is 2.

- *FG_TRIGGERCC_SELECT0* = **CC_PULSEGEN0**

  The output allocation is as usual.

- *FG_TRIGGEROUT_SELECT_FRONT_GPO_0* = **PULSEGEN1**
- *FG_TRIGGEROUT_SELECT_FRONT_GPO_1* = **PULSEGEN2**

Start the trigger system using parameter *FG_TRIGGERSTATE* as usual. You will notice that you get thrice the number of images from the frame grabber than external trigger pulses have been generated by the light barrier. Equally to the previous example, check for exceeding line periods at the input when you run your application or ensure that your external hardware will not generate the input pulses with an exceeding frequency.

Keep in mind to start the acquisition before activating the trigger system. This is because you will receive three images for one external trigger pulse. If you start the acquisition after the trigger system, you cannot ensure that the first transfered image is the first image out of a sequence.

### 7.4.4. Software Trigger

The previous examples showed the use of the internal frequency generator and the use of external trigger pulses to trigger your camera and generate digital output signals. Another trigger mode is the software trigger. In this mode, you can control the generation of each trigger pulse using your software application. To use the software triggered mode, set parameter *FG_AREATRIGGERMODE* to **ATM_SOFTWARE**. Next, configure the pulse form generators and the outputs as usual and start the trigger system (set *FG_TRIGGERSTATE* to **TS_ACTIVE**) and the acquisition. Now, you can generate a trigger pulse by writing value '1' to parameter *FG_SENDSOFTWARETRIGGER* i.e. each time you write to this parameter, a trigger pulse is generated. The relevant blocks of the trigger system are illustrated in the following figure.

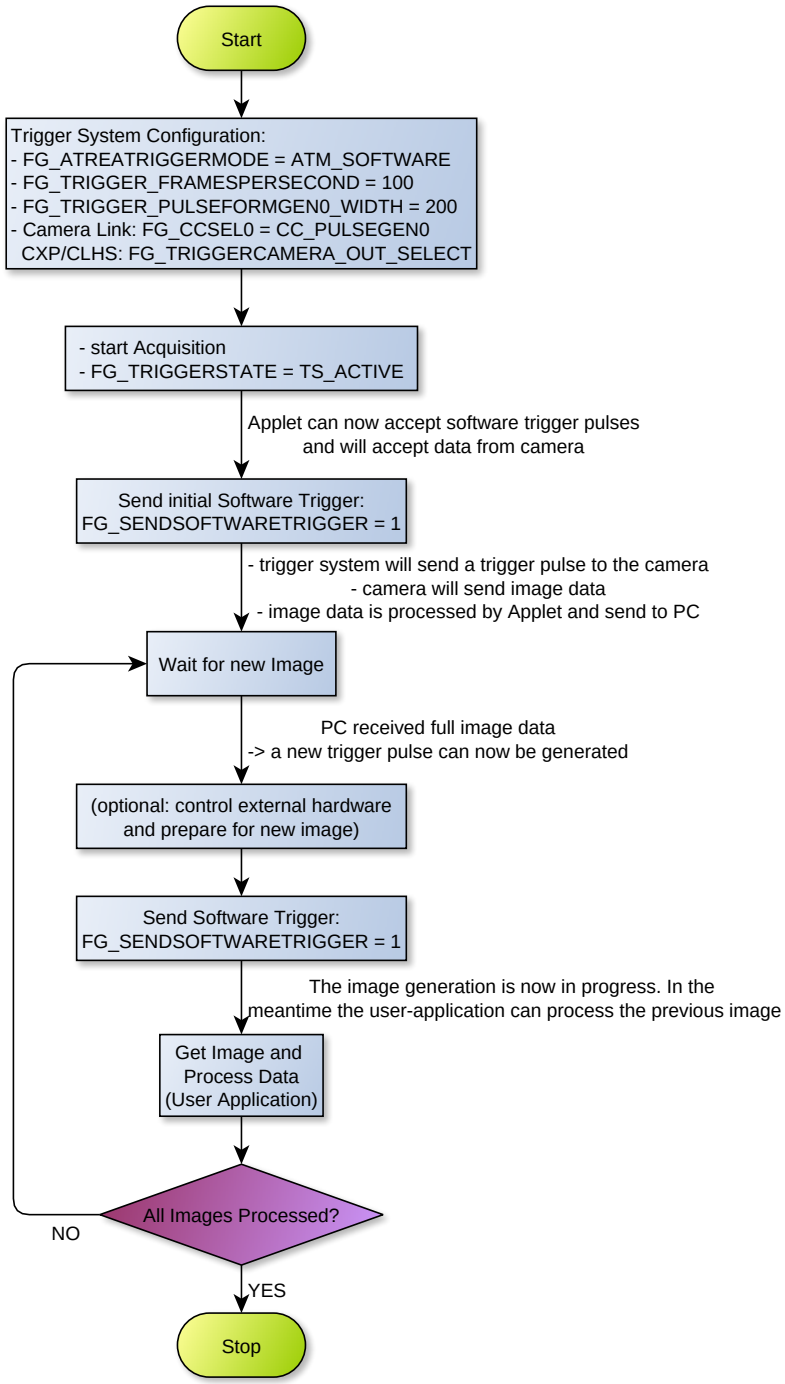
Keep in mind that the time between two pulses has to be larger than 1 / *FG_TRIGGER_FRAMESPERSECOND* as this will limit the maximum trigger frequency. The trigger system offers the possibility to check if a

new software trigger pulse can be accepted i.e. the trigger system is not busy anymore. Read parameter *FG_SOFTWARETRIGGER_IS_BUSY* to check it's state. While the parameter has value **IS_BUSY**, writing to parameter *FG_SENDSOFTWARETRIGGER* is not allowed and will be ignored. You should always check if the system is not busy before writing a pulse. To check if you lost a pulse, read parameter *FG_TRIGGER_EXCEEDED_PERIOD_LIMITS*.

In some cases, you might want to generate a sequence of pulses for each software trigger. To do this, simply set parameter *FG_TRIGGER_MULTIPLY_PULSES* to the desired sequence length. Now, for every software trigger pulse written to the trigger system, a sequence of the define length with a frequency defined by parameter *FG_TRIGGER_FRAMESPERSECOND* is generated. Again, the system cannot accept further inputs while a sequence is being processed.

Let's have a look at some flow chart examples on how to use the trigger system in software triggered mode. The flow charts visualize the steps of a fictitious user software implementation. In the first example, we simply generate single software trigger pulses using parameter *FG_SENDSOFTWARETRIGGER*. When the applet receives this pulse, it will trigger the camera. The camera will send an image to the frame grabber which will be processed there and will be output to the PC via DMA transfer. In the meantime, the users software application will wait for any DMA transfers. After the application got the notification that a new image has been fully tranferred to the PC it will send a new software trigger pulse and the frame grabber and camera will start again generating an image. Our software application will now have the time to process the previously received image until it is waiting for a new transfer. Thus, the software can process images while image generation is in progress. Of course, you can first process your images and afterwards generate a new trigger pulse, as well. So the steps for a repeating sequence are: Generate a SW trigger pulse, wait for image, generate a SW trigger pulse, wait for image. The flowchart of this example can be found in the following figure.

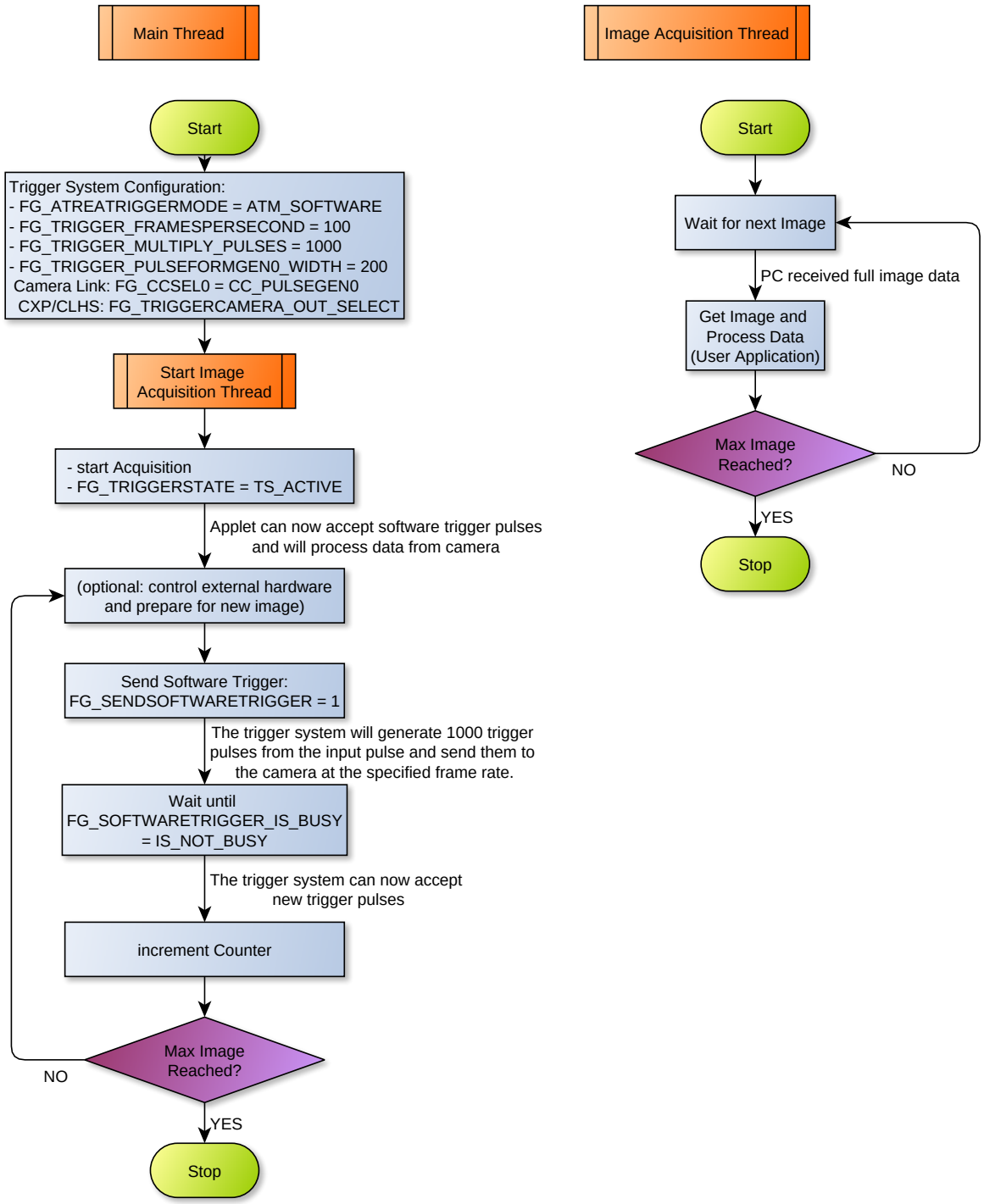
**Figure 7.13. Flowchart of Software Application Using the Software Trigger**




In the sample application shown above, it is ensured that the trigger system is not busy after you received the image. Therefore, we do not need to check for the software trigger busy flag in this example. One drawback of the example is that we might not acquire the frames at the maximum speed. This is because we have to wait for the full transfer of images before generating a new trigger pulse. Cameras can accept new trigger pulses while they transfer image data. The next example will therefore use the trigger sequencer.

The next example uses two threads. One thread for trigger generation and one thread for image acquisition and processing. In comparison to the previous example, we use the trigger sequencer for pulse multiplication and we will have to use the busy flag. This will allow an acquisition at a higher frame rate.

**Figure 7.14. Flowchart of Software Application Using the Software Trigger with a Sequencer**




The main thread will configure and start the trigger system and the acquisition. For each software trigger pulse we send to the frame grabber, 1000 pulses are generated and send to the camera at the framerate specified by *FG_TRIGGER_FRAMESPERSECOND*. After sending a software trigger pulse to the frame grabber we wait until the software is not busy anymore by polling on register *FG_SOFTWARETRIGGER_IS_BUSY*. To control the number of generated trigger pulses we count each successful sequence generation. If more images are required we can send another software trigger pulse to the frame grabber to start a new sequence.

The second thread is used for image acquisition and image data processing. Here, the software will wait for new incoming images (Use function Fg_getLastPicNumberBlockingEx() for example) and process the received images. The thread can exit if the desired number of images have been acquired and processed.

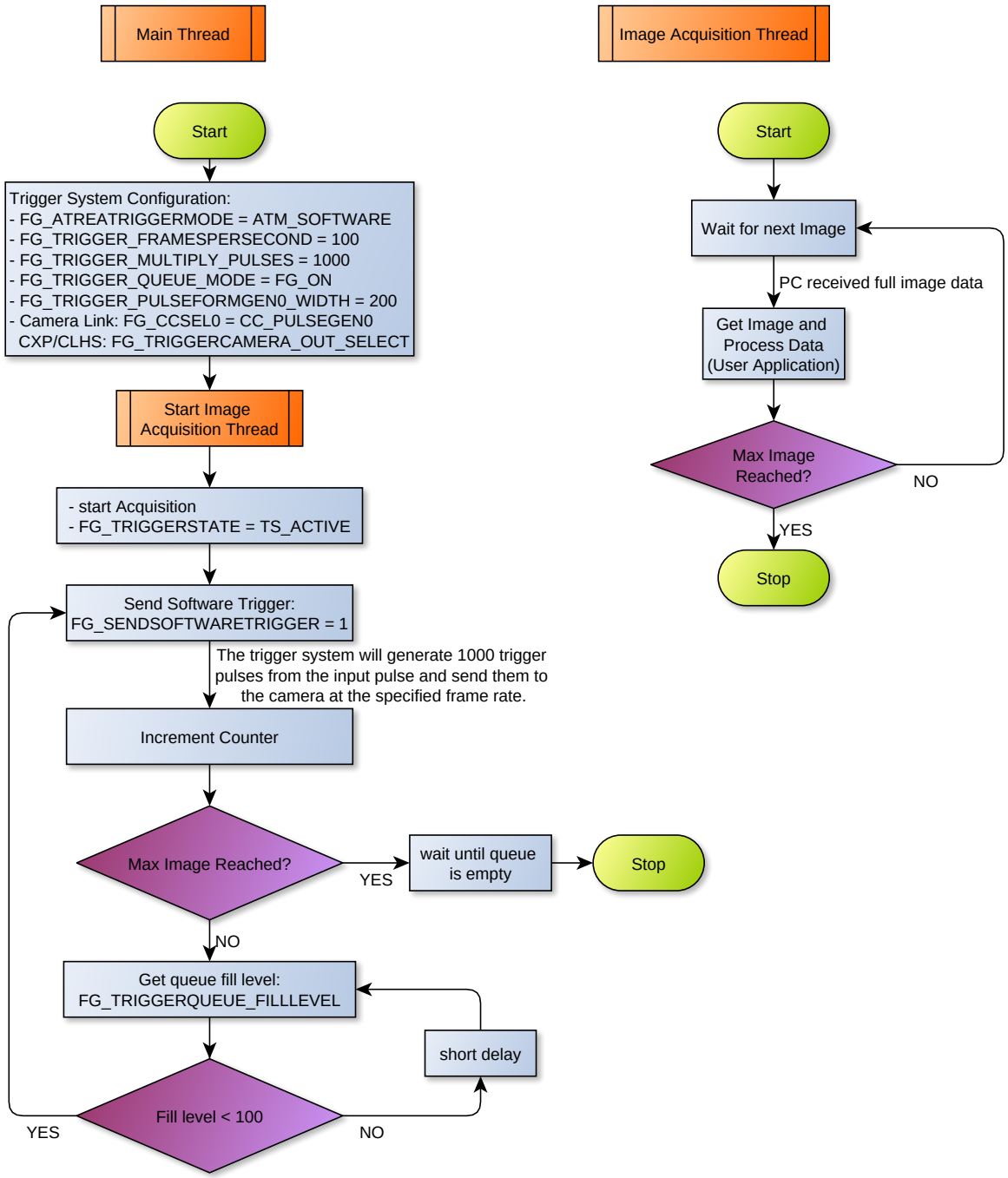
### 7.4.5. Software Trigger with Trigger Queue

To understand the following scenario you should have read the previous scenario first. In the following we will have a look at the software trigger once again. This time, we use the trigger queue. The trigger queue enables the buffering of trigger pulses from external sources or from the software trigger and will output these pulses at the maximum allowed frequency specified by *FG_TRIGGER_FRAMESPERSECOND*. Therefore, we can write to *FG_SENDSOFTWARETRIGGER* multiple times even if the trigger system is still busy. Parameter *FG_SOFTWARETRIGGER_IS_BUSY* will only have value **IS_BUSY** if the queue is full. Instead of writing multiple times to *FG_SENDSOFTWARETRIGGER* you can directly write the number of required pulses to the parameter.

The trigger queue can buffer 2040 sequence pulses. Thus if you have a certain sequence length of N pulses and currently 200 pulses in the queue, the trigger system can store additional 1840 remaining pulses. You can check the fill level by reading parameter *FG_TRIGGERQUEUE_FILLLEVEL*.

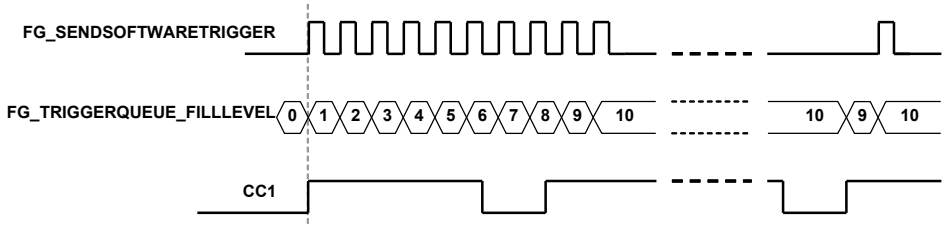
In the following flow chart you can see a queue fill level minimum limit of 10 pulses. In our supposed application we will check the queue fill level and compare it with our limit. If less pulses are in the queue, we generate a new software trigger pulse. Thus, on startup, the queue will fill-up until it contains 10 pulses. We count the software trigger pulses send to the trigger system. Multiplied with our sequence length, we can obtain the number of pulses which will be send to the camera. If enough pulses have been generated, we can stop the trigger pulse generation.

**Figure 7.15. Flowchart of Software Application Using the Software Trigger with Trigger Queue**




When having a look at the waveform (Figure 7.16) we can see the initialization phase where the queue is filled. After fill level value 10 has been reached, no more software trigger pulses are written to the applet. The system will now continue the output of trigger pulses. As our sequence length is 1000 pulses we have to wait for 1000 pulses to be generated until a change in the fill level will occur. After the 1000th pulse has been completely generated, the fill level will change to 9. This will cause the generation of another software trigger pulse by our sample application which will cause a fill level of 10 again.

**Figure 7.16. Waveform Illustrating Software Trigger with Queue Example"**




When using the trigger queue, the stopping of the trigger system is of interest. If you set parameter *FG_TRIGGERSTATE* to **TS_SYNC_STOP**, the trigger system will stop accepting inputs such as software trigger pulses, but it will complete the trigger pulse generation until the queue is empty and all pulses are fully output. You can immediately cancel the pulse generation by setting the *FG_TRIGGERSTATE* to **TS_ASYNC_STOP**.

## 7.4.6. External Trigger with Trigger Queue

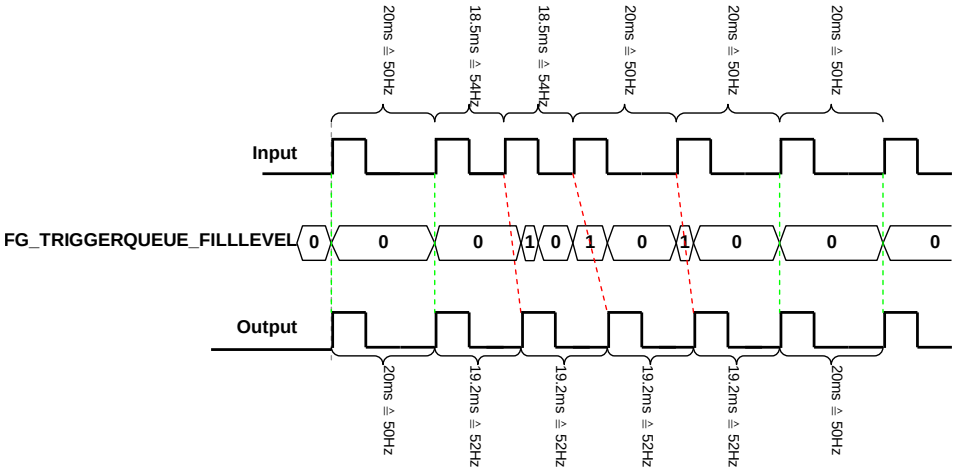
Of course, we can use the trigger queue with external triggers, too. This will give us a possibility to buffer 'jumpy' external encoders or any other external trigger signal generators. Let's suppose an external encoder which is configured to generate trigger pulses with a frequency of 50Hz and a camera which can be run at a maximum frequency of 52Hz. Thus, we set parameter *FG_TRIGGER_FRAMESPERSECOND* to 52Hz. Now assume that the external hardware is a little 'jumpy' and the 50Hz is just an average. So if we have inputs with a frequency higher than 52Hz we will loose at least one pulse. You can check this using the trigger lost events or by reading parameter *FG_TRIGGER_EXCEEDED_PERIOD_LIMITS*.

Now let's have a look at the same scenario if the queue is enabled. If it is enabled, we can buffer trigger pulses. Thus, we can buffer the exceeding input frequency and output the pulses at the maximum camera trigger frequency which is 52Hz in our example. After the input frequency is reduced, the queue will get empty and the pulse output is synchronous to the input again. Note that the delay might result in images with wrong content such as 'shifted' object positions.

To enable the queue, just write value **FG_ON** to *FG_TRIGGERQUEUE_MODE*.

The following waveform illustrates the input signal, the queue fill level and the output signal. At the beginning, the gap between the first two input signals is 20ms i.e. the frequency is less than 52Hz. Thus, the queue will not fill with pulses and the trigger system will directly output the second pulse. Now, the gap between the second and the third as well as the fourth pulse is less than 19.2ms and therefore, the trigger system will delay the output of these pulses to have a minimum gap of 19.2ms. During this period, the queue fill level will increment to value 1 for short periods. The gap between the fourth and the following input pulses is sufficiently long enough, however, the system will have to delay these pulses, too.

**Figure 7.17. Using External Trigger Signal Sources together with the Trigger Queue**




Note that the trigger lost event and *FG_TRIGGER_EXCEEDED_PERIOD_LIMITS* will only be set if the queue is full i.e. in overflow condition.

### 7.4.7. Bypass External Trigger Signals

When external trigger signals are used, the duty cycle i.e. signal width or signal length will always be ignored. Only the rising or falling edge depending on the polarity settings is considered. However, you can bypass an external source directly to an output. For example, you can bypass an external source to the camera which allows you to control the exposure time with the external source. Mind that you will bypass the trigger core system and therefore, no frequency checks or downscales can be performed.

It is also possible to mix the bypass with the generator or software trigger mode. Thus you can output the bypass signal to CC2, while CC1 includes pulses of the internal frequency generator or from another external source.

Use the output select parameters for camera control or digital outputs to select a bypass source. These are for example:

- *FG_TRIGGERCC_SELECT0* = BYPASS_GPI_0
- *FG_TRIGGEROUT_SELECT_FRONT_GPO_0* = BYPASS_FRONT_GPI_1

### 7.4.8. Multi Camera Applications / Synchronized Cameras

A basic application is that multiple cameras at one or more frame grabbers are connected to the same trigger source. If all cameras have to acquire images for every trigger pulse. Simply connect the trigger source to all frame grabbers and set the same trigger configuration for all cameras. Some applets support more than one camera. In this case, the same parameters for all cameras should be set. They may share the same trigger input.

If you do not have an external trigger source, but use the generator or the software trigger you can synchronize the triggers to ensure camera exposures at the same moment. Simply output the camera control signal on a digital trigger output and connect this output to a digital input of other frame grabbers which have to be synchronized with the master. In the slave applets bypass the input to the camera control outputs.

#### Arbitrary Output Allocation

In multiple camera applets you can also select another camera trigger module source. For example, camera B Camera Control trigger signal one (CC1) can use **CAM_A_PULSEGEN0**.

### 7.4.9. Hardware System Analysis and Error Detection / Trigger Debugging

The Silicon Software trigger system includes powerful monitoring possibilities. They allow a convenient and efficient system analysis and will help you to detect errors in your hardware setup and wrong parameterizations.

Let's have a look at the simple external trigger example once again. Assume that you have set up all devices and have fully configured the applet. You start the system and receive images. Unfortunately, the number of acquired images or the framerate is not as expected. This means, at some point trigger signals or frames got lost. To analyze the error, let's have a look at the monitoring applet registers.

- Trigger Input Statistics

  The parameters of the trigger input statistics category allow an analysis of the external trigger pulses. Parameter *FG_TRIGGERIN_STATS_FREQUENCY* performs a continuous frequency measurement of the input signals. Compare this value with the expected trigger input frequency. If the measured frequency is much higher or lower than the expected frequency, check your external hardware. Also check if the correct trigger input has been chosen by parameter *FG_TRIGGERIN_SRC* and if the pulse width of the input is long enough to be detected by the hardware.

  To validate a constant input frequency, the trigger system will also show the maximum and minimum detected frequencies using parameters *FG_TRIGGERIN_STATS_MAXFREQUENCY* and *FG_TRIGGERIN_STATS_MINFREQUENCY*. On startup, you will have a very low frequency as no external pulses might have been detected so far. Therefore, you have to clear the measurement using parameter

*FG_TRIGGERIN_STATS_MINMAXFREQUENCY_CLEAR* first. If you detect an unwanted deviation from the expected values or the difference between the minimum and maximum frequency is comparably high, your external trigger generating hardware might be 'jumpy', skips pulses or is 'bouncing' which causes pulse multiplication. In this case, you might be able to compensate the problem using a higher debouncing value, set a lower maximum allowed frequency (see Section 7.4.2, 'External Trigger Signals / IO Triggered') or use the trigger queue (see Section 7.4.6, 'External Trigger with Trigger Queue').

Another feature of the input statistics module is the pulse counting. This feature can be used to compare the number of input pulses with the output pulses and acquired images. Read the pulse count value from parameter *FG_TRIGGERIN_STATS_PULSECOUNT*. To ensure a synchronized counting of the input and any output pulses and images you should clear the pulse counter before generating external trigger inputs.

- Trigger Output Statistics

  A pulse counter is connected to the trigger output, too. Here you can select one of the pulse form generators using parameter *FG_TRIGGEROUT_STATS_SOURCE* and read the value with parameter *FG_TRIGGEROUT_STATS_PULSECOUNT*. Reset the pulse counter using *FG_TRIGGEROUT_STATS_PULSECOUNT_CLEAR*.

  Use the pulse count value to compare it with the input pulse counter. If the values vary, pulses in the frame grabber have been discarded. This can happen if the input frequency is higher than the maximum allowed frequency specified by parameter *FG_TRIGGER_FRAMESPERSECOND*. If this happens, flag *FG_TRIGGER_EXCEEDED_PERIOD_LIMITS* will be set. Moreover, if the pulse counter values dramatically differ, ensure that no trigger multiplication and/or downscaling has been set. Check parameters *FG_TRIGGERIN_DOWNSCALE*, *FG_TRIGGER_MULTIPLY_PULSES* and the downscale parameters of the pulse form generators.

  It is also possible to count the input and output pulses with the input events and the output event *FG_TRIGGER_OUTPUT_CAM0*.

- Camera Response Check

  Trigger pulses might get lost in the link to the camera or the trigger frequency is to high to be processed by the camera. In this case, the number of frames received by the frame grabber differs from the trigger pulses sent. For this error, the trigger system includes the missing camera frame response detection module. The module can detect missing frames and generate an event for each lost frame or set a register. Check Section 7.5.12.3, 'Statistics' for more information and usage.

- Acquired Image Compare

  Of course, it is also possible to count the number of acquired images i.e. the number of DMA transfers and compare them with the generated trigger pulses. If the values differ, you might have lost trigger pulses in the camera. In this case, check that the trigger frequency is not to high for the camera. Ensure that you do not run the applet in overflow state, where images can get lost in the applet. If the applet is run in overflow, check the maximum bandwidths of the applet. A smaller region of interest might solve the problems.

For every monitoring values, check the maximum and minimum ranges of the parameters. If pulse counters reached their maximum value, they will reset and start from zero.

## 7.5. Parameters

### 7.5.1. FG_AREATRIGGERMODE

The area trigger system of this applet can be run in three different operation modes.

- Generator

  An internal frequency generator at a specified frequency will be used as trigger source. All digital trigger inputs and software trigger pulses will be ignored.

- External

  In this mode, one of the digital inputs is used as trigger source i.e. you can use an external source for trigger generation.

- Software

  In software triggered mode, you will need to manually generate the trigger input signals. This has to be done by writing to an applet parameter.

- Synchronized

  The synchronized mode is not available in this applet. Multi-camera applets include this option. Check the respective applet documentations in this case.

## Free-Run Mode

If you like to use your camera in free run mode you can use any of the modes described above. The camera will ignore all trigger pulses or, if required, you can disable the output or deactivate the trigger using parameter *FG_TRIGGERSTATE*.

## Allowed Frequencies

Mind the influence of parameter *FG_TRIGGER_FRAMESPERSECOND* in external and software triggered mode. Always set this parameter for these modes.

**Table 7.2. Parameter properties of FG_AREATRIGGERMODE**

| Property | Value |
|---|---|
| Name | **FG_AREATRIGGERMODE** |
| Display Name | **Area Trigger Mode** |
| Type | **Enumeration** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **ATM_GENERATOR** Generator<br>**ATM_EXTERNAL** External<br>**ATM_SOFTWARE** Software |
| Default value | **ATM_GENERATOR** |

**Example 7.1. Usage of FG_AREATRIGGERMODE**

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

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

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

### 7.5.2. FG_TRIGGERSTATE

The area trigger system is operating in three trigger states. In the 'Active' state, the module is fully enabled. Trigger sources are used, pulses are queued, downscaled, multiplied and the output signals get their parameterized pulse forms. If the trigger is set into the 'Sync Stop' mode, the module will ignore further input pulses or stop the generation of pulses. However, the module will still process the pulses in the system. This means, a filled queue and the sequencer will continue processing the pulses and furthermore, the pulse form

generators will output the signals according to the parameterized parameters. Finally, the 'Async Stop' mode asynchronously and immediately stops the full trigger system . Note that this stop might result in output signals of undefined signal length as a current signal generation could be interrupted. Also note that a restart of a previously stopped trigger i.e. switching to the 'Active' state will clear the queue and the sequencer.

**Table 7.3. Parameter properties of FG_TRIGGERSTATE**

| Property | Value |
|---|---|
| Name | **FG_TRIGGERSTATE** |
| Display Name | **Trigger State** |
| Type | **Enumeration** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **TS_ACTIVE** Active<br>**TS_ASYNC_STOP** Async Stop<br>**TS_SYNC_STOP** Sync Stop |
| Default value | **TS_SYNC_STOP** |

**Example 7.2. Usage of FG_TRIGGERSTATE**

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

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

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

### 7.5.3. FG_TRIGGER_FRAMESPERSECOND

This is a very important parameter of the trigger system. It is used for multiple functionalities.

If you run the trigger system in 'Generator' mode, this parameter will define the frequency of the generator. If you run the trigger system in 'External' or 'Software Trigger' operation mode, this parameter will specify the maximum allowed input frequency. Input frequencies which exceed this limit will cause the loss of the input pulse. To notify the user of this error, a read register contains an error flag or an event is generated. However, if the trigger queue is enabled, the exceeding pulses will be buffered and output at the maximum frequency which is defined by *FG_TRIGGER_FRAMESPERSECOND*. Thus, the parameter also defines the maximum queue output frequency. Moreover, it defines the maximum sequencer frequency. The maximum valid value of *FG_TRIGGER_FRAMESPERSECOND* is limited by *FG_CAMERASIMULATOR_FRAMERATE* in camera simulator mode.

Note that the range of this parameter depends on the settings in the pulse form generators. If you want to increase the frequency you might need to decrease the width or delay of one of the pulse form generators.

**Equation 7.1. Dependency of Frequency and Pulse Form Generators**

$$\frac{1}{\text{fps}} > Max\left\{\frac{Max\{\text{WIDTH0},\text{DELAY0}\}}{\text{DOWNSCALE0}}, \frac{Max\{\text{WIDTH1},\text{DELAY1}\}}{\text{DOWNSCALE1}}, \frac{Max\{\text{WIDTH2},\text{DELAY2}\}}{\text{DOWNSCALE2}}, \frac{Max\{\text{WIDTH3},\text{DELAY3}\}}{\text{DOWNSCALE3}}\right\}$$

- fps = *FG_TRIGGER_FRAMESPERSECOND*
- WIDTH[0..3] = *FG_TRIGGER_PULSEFORMGEN[0..3]_WIDTH*
- DELAY[0..3] = *FG_TRIGGER_PULSEFORMGEN[0..3]_DELAY*
- DOWNSCALE[0..3] = *FG_TRIGGER_PULSEFORMGEN[0..3]_DOWNSCALE*

Read the general trigger system explanations and the respective parameter explanations for more information.

**Table 7.4. Parameter properties of FG_TRIGGER_FRAMESPERSECOND**

| Property | Value |
|---|---|
| Name | **FG_TRIGGER_FRAMESPERSECOND** |
| Display Name | **Frames/Sec** |
| Type | **Double** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum** 0.0291038304567337<br>**Maximum** 1.25E7<br>**Stepsize** 2.220446049250313E-16 |
| Default value | **8.0** |
| Unit of measure | **Hz** |

**Example 7.3. Usage of FG_TRIGGER_FRAMESPERSECOND**

```
int result = 0;
double value = 8.0;
const enum FgParamTypes type = FG_PARAM_TYPE_DOUBLE;

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

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

### 7.5.4. Trigger Input

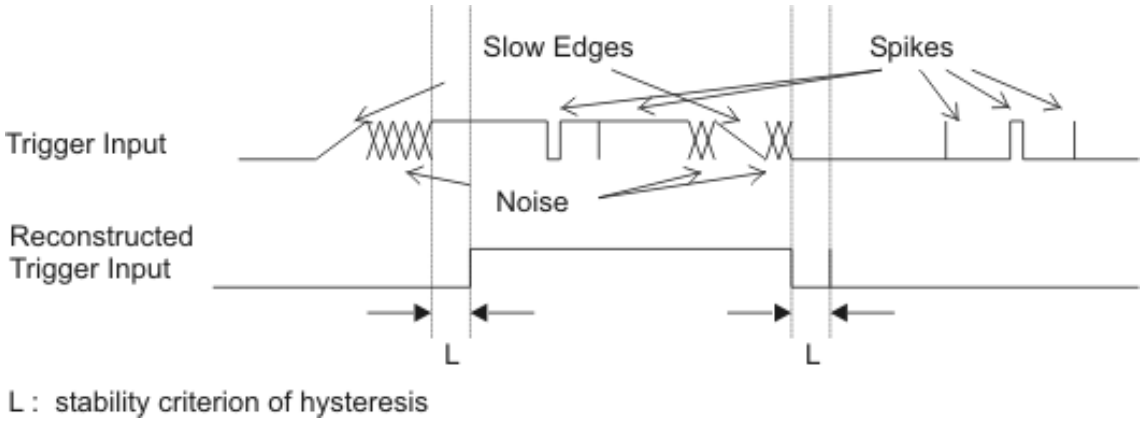
The parameters of category Trigger Input are used to configure the input source of the trigger system. The category is divided into sub categories. All external sources are configured in category external. Category software trigger allows the configuration, monitoring and controlling of software trigger pulses. In category statistics the parameters for input statistics are present.

#### 7.5.4.1. External

##### 7.5.4.1.1. FG_TRIGGERIN_DEBOUNCE

In general, a perfect and steady trigger input signal can not be guaranteed in practice. A transfer using long cable connections and the operation in bad shielded environments might have a distinct influence on the signal quality. Typical problems are strong flattening of the digital's signal edges, occurring interferences during toggling and inducing of short jamming pulses (spikes). In the following figure, some of the influences are illustrated.

**Figure 7.18. Faulty Signal and it's Reconstruction**




The trigger system has been designed to work highly reliable even under problematic signal conditions. An internal debouncing of the inputs will eliminate unwanted trigger pulses. It is comparable to a hysteresis. Only signal changes which are constant for a specified time (marked 'L' in the figure) are accepted which makes the input insensitive to jamming pulses. Also multiple triggering will be effectively disabled, which occurs by slow signal transfers and bouncing. Set the debounce time according to your requirements in µs. Note that the debounce time will also be the delay time before the trigger signal can be processed. The settings made for this parameter affect all digital inputs.

**Table 7.5. Parameter properties of FG_TRIGGERIN_DEBOUNCE**

| Property | Value |
|---|---|
| Name | **FG_TRIGGERIN_DEBOUNCE** |
| Display Name | **Input Debounce** |
| Type | **Double** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum 0.024**<br>**Maximum 1.512**<br>**Stepsize 0.024** |
| Default value | **1.0** |
| Unit of measure | **µs** |

**Example 7.4. Usage of FG_TRIGGERIN_DEBOUNCE**

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

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

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

### 7.5.4.1.2. FG_GPI

Parameter *FG_GPI* is used to monitor the digital inputs of the frame grabber.

You can read the current state of these inputs using parameter *FG_GPI*. 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 7.6. Parameter properties of FG_GPI**

| Property | Value |
|---|---|
| Name | **FG_GPI** |
| Display Name | **Digital Input at GPI** |
| Type | **Unsigned Integer** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **Minimum 0**<br>**Maximum 255**<br>**Stepsize 1** |

**Example 7.5. Usage of FG_GPI**

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

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

### 7.5.4.1.3. FG_FRONT_GPI

Parameter *FG_FRONT_GPI* is used to monitor the digital inputs of the frame grabber.

You can read the current state of these inputs using parameter *FG_FRONT_GPI*. Bit 0 of the read value represents input 0, bit 1 represents input 1 and so on. For example, if you obtain the value 10 or hexadecimal 0xA the frame grabber will have high level on it's digital inputs 1 and 3.

**Table 7.7. Parameter properties of FG_FRONT_GPI**

| Property | Value |
|---|---|
| Name | **FG_FRONT_GPI** |
| Display Name | **Digital Input at Front GPI** |
| Type | **Unsigned Integer** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **Minimum 0**<br>**Maximum 15**<br>**Stepsize 1** |

**Example 7.6. Usage of FG_FRONT_GPI**

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

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

### 7.5.4.1.4. FG_TRIGGERIN_SRC

To use the external trigger you have to select the input carrying the image trigger signal. Select one of the eight inputs. eight GPI and four Front GPI inputs on marathon frame grabbers and four GPI and two Front GPI inputs on LightBridge frame grabbers.

**Table 7.8. Parameter properties of FG_TRIGGERIN_SRC**

| Property | Value | |
|---|---|---|
| Name | **FG_TRIGGERIN_SRC** | |
| Display Name | **Source** | |
| Type | **Enumeration** | |
| Access policy | **Read/Write/Change** | |
| Storage policy | **Persistent** | |
| Allowed values | **TRGINSRC_GPI_0** | GPI Trigger Source 0 |
| | **TRGINSRC_GPI_1** | GPI Trigger Source 1 |
| | **TRGINSRC_GPI_2** | GPI Trigger Source 2 |
| | **TRGINSRC_GPI_3** | GPI Trigger Source 3 |
| | **TRGINSRC_GPI_4** | GPI Trigger Source 4 (not available on LightBridge) |
| | **TRGINSRC_GPI_5** | GPI Trigger Source 5 (not available on LightBridge) |
| | **TRGINSRC_GPI_6** | GPI Trigger Source 6 (not available on LightBridge) |
| | **TRGINSRC_GPI_7** | GPI Trigger Source 7 (not available on LightBridge) |
| | **TRGINSRC_FRONT_GPI_0** | Front GPI Trigger Source 0 |
| | **TRGINSRC_FRONT_GPI_1** | Front GPI Trigger Source 1 |
| | **TRGINSRC_FRONT_GPI_2** | Front GPI Trigger Source 2 |
| | **TRGINSRC_FRONT_GPI_3** | Front GPI Trigger Source 3 |
| Default value | **TRGINSRC_GPI_0** | |

**Example 7.7. Usage of FG_TRIGGERIN_SRC**

```
int result = 0;
int value = TRGINSRC_GPI_0;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

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

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

### 7.5.4.1.5. FG_TRIGGERIN_POLARITY

For the selected input using parameter *FG_TRIGGERIN_SRC* the polarity is set with this parameter.

**Table 7.9. Parameter properties of FG_TRIGGERIN_POLARITY**

| Property | Value | |
|---|---|---|
| Name | **FG_TRIGGERIN_POLARITY** | |
| Display Name | **Polarity** | |
| Type | **Enumeration** | |
| Access policy | **Read/Write/Change** | |
| Storage policy | **Persistent** | |
| Allowed values | **LOW_ACTIVE** | Low Active |
| | **HIGH_ACTIVE** | High Active |
| Default value | **HIGH_ACTIVE** | |

**Example 7.8. Usage of FG_TRIGGERIN_POLARITY**

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

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

```
    /* error handling */
}

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

### 7.5.4.1.6. FG_TRIGGERIN_DOWNSCALE

If you use the trigger system in external trigger mode, you can downscale the trigger inputs selected by *FG_TRIGGERIN_SRC*. See *FG_TRIGGERIN_DOWNSCALE_PHASE* for more information.

**Table 7.10. Parameter properties of FG_TRIGGERIN_DOWNSCALE**

| Property | Value |
|---|---|
| Name | **FG_TRIGGERIN_DOWNSCALE** |
| Display Name | **Input Downscale** |
| Type | **Unsigned Integer** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum 1**<br>**Maximum 2147483647**<br>**Stepsize 1** |
| Default value | **1** |

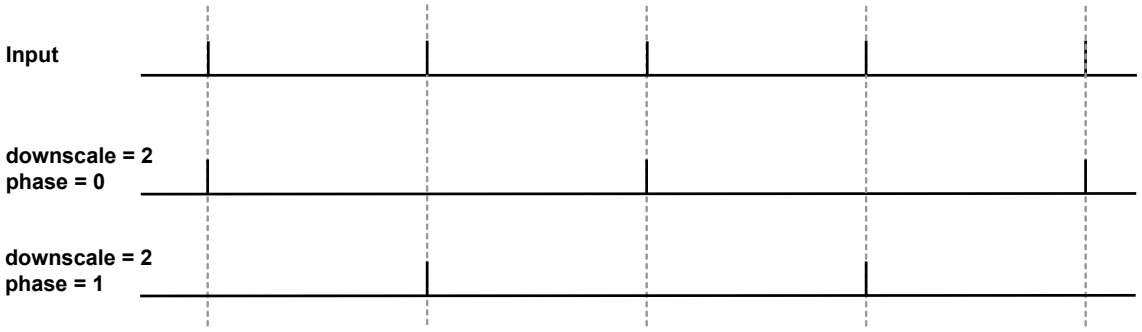
**Example 7.9. Usage of FG_TRIGGERIN_DOWNSCALE**

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

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

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

### 7.5.4.1.7. FG_TRIGGERIN_DOWNSCALE_PHASE

Parameters *FG_TRIGGERIN_DOWNSCALE* and *FG_TRIGGERIN_DOWNSCALE_PHASE* are used to downscale external trigger inputs. The downscale value represents the factor. For example value three will remove two out of three successive trigger pulses. The phase is used to make the selection of the pulse in the sequence. For the given example, a phase set to value zero will forward the first pulse and will remove pulses two and three of a sequence of three pulses. See the following figure for more explanations.

**Figure 7.19. Triggerin Dowscale**




Mind the dependency between the downscale factor and the phase. The value of the downscale factor has to be greater than the phase!

**Table 7.11. Parameter properties of FG_TRIGGERIN_DOWNSCALE_PHASE**

| Property | Value |
|---|---|
| Name | **FG_TRIGGERIN_DOWNSCALE_PHASE** |
| Display Name | **Downscale Phase** |
| Type | **Unsigned Integer** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum 0**<br>**Maximum 4294967294**<br>**Stepsize 1** |
| Default value | **0** |

**Example 7.10. Usage of FG_TRIGGERIN_DOWNSCALE_PHASE**

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

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

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

### 7.5.4.2. Software Trigger

#### 7.5.4.2.1. FG_SENDSOFTWARETRIGGER

If the trigger system is run in software triggered mode (see parameter *FG_AREATRIGGERMODE*), this parameter is activated. Write value '1' to this parameter to input a software trigger. If the trigger queue is activated multiple software trigger pulses can be written to the frame grabber. They will fill the queue and being processed with the maximum allowed frequency parameterized by *FG_TRIGGER_FRAMESPERSECOND*.

Note that software trigger pulses can only be written if the trigger system has been activated using parameter *FG_TRIGGERSTATE*. Moreover, if the queue has not been activated, new software trigger pulses can only be written if the trigger system is not busy. Therefore, writing to the parameter can cause an Software Trigger Busy error.

**Table 7.12. Parameter properties of FG_SENDSOFTWARETRIGGER**

| Property | Value |
|---|---|
| Name | **FG_SENDSOFTWARETRIGGER** |
| Display Name | **Send Pulses** |
| Type | **Unsigned Integer** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Transient** |
| Allowed values | **Minimum 1**<br>**Maximum 1**<br>**Stepsize 1** |
| Default value | **1** |
| Unit of measure | **pulses** |

**Example 7.11. 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 */
}
```

### 7.5.4.2.2. FG_SOFTWARETRIGGER_IS_BUSY

After writing one or multiple pulses to the trigger system using the software trigger, the system might be busy for a while. To check if there are no pulses left for processing use this parameter.

**Table 7.13. Parameter properties of FG_SOFTWARETRIGGER_IS_BUSY**

| Property | Value |
|---|---|
| Name | **FG_SOFTWARETRIGGER_IS_BUSY** |
| Display Name | **Software Trigger Busy** |
| Type | **Enumeration** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **IS_BUSY** Busy Flag is set<br>**IS_NOT_BUSY** Busy Flag is not set |

**Example 7.12. Usage of FG_SOFTWARETRIGGER_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_SOFTWARETRIGGER_IS_BUSY, &value, 0, type)) < 0) {
    /* error handling */
}
```

### 7.5.4.2.3. FG_SOFTWARETRIGGER_QUEUE_FILLLEVEL

The value of this parameter represents the number of pulses in the software trigger queue which have to be processed. The fill level depends on the number of pulses written to *FG_SENDSOFTWARETRIGGER*, the trigger pulse multiplication factor *FG_TRIGGER_MULTIPLY_PULSES* and the maximum output frequency defined by *FG_TRIGGER_FRAMESPERSECOND*. The value decrement is given in steps of *FG_TRIGGER_MULTIPLY_PULSES*.

**Table 7.14. Parameter properties of FG_SOFTWARETRIGGER_QUEUE_FILLLEVEL**

| Property | Value |
|---|---|
| Name | **FG_SOFTWARETRIGGER_QUEUE_FILLLEVEL** |
| Display Name | **Software Trigger Queue Filllevel** |
| Type | **Unsigned Integer** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **Minimum** 0<br>**Maximum** 2040<br>**Stepsize** 1 |
| Unit of measure | **pulses** |

**Example 7.13. Usage of FG_SOFTWARETRIGGER_QUEUE_FILLLEVEL**

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

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

### 7.5.4.3. Statistics

The trigger input statistics module will offer you frequency analysis and pulse counting of the selected input. The digital input for the statistics is selected by *FG_TRIGGERIN_POLARITY*. Measurements are performed after debouncing and polarity selection but before downscaling.

The statistics section also includes a list of digital input events.

#### 7.5.4.3.1. FG_TRIGGERIN_STATS_SOURCE

The trigger statistics module allows you to individually select one of the inputs as source. Select one of the eight GPI and two Front GPI inputs on marathon frame grabbers and four GPI and two Front GPI inputs on LightBridge frame grabbers.

**Table 7.15. Parameter properties of FG_TRIGGERIN_STATS_SOURCE**

| Property | Value | |
|---|---|---|
| Name | **FG_TRIGGERIN_STATS_SOURCE** | |
| Display Name | **Statistics Source** | |
| Type | **Enumeration** | |
| Access policy | **Read/Write/Change** | |
| Storage policy | **Persistent** | |
| Allowed values | **TRGINSRC_GPI_0** | GPI Trigger Source 0 |
| | **TRGINSRC_GPI_1** | GPI Trigger Source 1 |
| | **TRGINSRC_GPI_2** | GPI Trigger Source 2 |
| | **TRGINSRC_GPI_3** | GPI Trigger Source 3 |
| | **TRGINSRC_GPI_4** | GPI Trigger Source 4 (not available on LightBridge) |
| | **TRGINSRC_GPI_5** | GPI Trigger Source 5 (not available on LightBridge) |
| | **TRGINSRC_GPI_6** | GPI Trigger Source 6 (not available on LightBridge) |
| | **TRGINSRC_GPI_7** | GPI Trigger Source 7 (not available on LightBridge) |
| | **TRGINSRC_FRONT_GPI_0** | Front GPI Trigger Source 0 |
| | **TRGINSRC_FRONT_GPI_1** | Front GPI Trigger Source 1 |
| | **TRGINSRC_FRONT_GPI_2** | Front GPI Trigger Source 2 |
| | **TRGINSRC_FRONT_GPI_3** | Front GPI Trigger Source 3 |
| Default value | **TRGINSRC_GPI_0** | |

**Example 7.14. Usage of FG_TRIGGERIN_STATS_SOURCE**

```
int result = 0;
int value = TRGINSRC_GPI_0;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

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

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

### 7.5.4.3.2. FG_TRIGGERIN_STATS_POLARITY

For the selected input using parameter *FG_TRIGGERIN_STATS_SOURCE* the polarity is set using this parameter.

**Table 7.16. Parameter properties of FG_TRIGGERIN_STATS_POLARITY**

| Property | Value |
|---|---|
| Name | **FG_TRIGGERIN_STATS_POLARITY** |
| Display Name | **Statistics Polarity** |
| Type | **Enumeration** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **LOW_ACTIVE** Low Active<br>**HIGH_ACTIVE** High Active |
| Default value | **HIGH_ACTIVE** |

**Example 7.15. Usage of FG_TRIGGERIN_STATS_POLARITY**

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

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

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

### 7.5.4.3.3. FG_TRIGGERIN_STATS_PULSECOUNT

The input pulses are count and the current value can be read with this parameter. Use the counter for verification of your system. For example, compare the counter value with the received number of images to check for exceeding periods.

**Table 7.17. Parameter properties of FG_TRIGGERIN_STATS_PULSECOUNT**

| Property | Value |
|---|---|
| Name | **FG_TRIGGERIN_STATS_PULSECOUNT** |
| Display Name | **Input Pulses** |
| Type | **Unsigned Integer** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **Minimum** **0**<br>**Maximum** **65535**<br>**Stepsize** **1** |
| Unit of measure | **pulses** |

**Example 7.16. Usage of FG_TRIGGERIN_STATS_PULSECOUNT**

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

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

### 7.5.4.3.4. FG_TRIGGERIN_STATS_PULSECOUNT_CLEAR

Clear the input pulse counter by writing to this register.

**Table 7.18. Parameter properties of FG_TRIGGERIN_STATS_PULSECOUNT_CLEAR**

| Property | Value |
|---|---|
| Name | **FG_TRIGGERIN_STATS_PULSECOUNT_CLEAR** |
| Display Name | **Clear Pulse Count** |
| Type | **Enumeration** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Transient** |
| Allowed values | **FG_APPLY** Apply |
| Default value | **FG_APPLY** |

**Example 7.17. Usage of FG_TRIGGERIN_STATS_PULSECOUNT_CLEAR**

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

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

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

### 7.5.4.3.5. FG_TRIGGERIN_STATS_FREQUENCY

The current frequency can be read using this parameter. It shows the frequency of the last two received pulses at the frame grabber.

**Table 7.19. Parameter properties of FG_TRIGGERIN_STATS_FREQUENCY**

| Property | Value |
|---|---|
| Name | **FG_TRIGGERIN_STATS_FREQUENCY** |
| Display Name | **Current Frequency** |
| Type | **Double** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **Minimum** 0.0<br>**Maximum** 1.25E8<br>**Stepsize** 2.220446049250313E-16 |
| Unit of measure | **Hz** |

**Example 7.18. Usage of FG_TRIGGERIN_STATS_FREQUENCY**

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

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

### 7.5.4.3.6. FG_TRIGGERIN_STATS_MINFREQUENCY

The trigger system will memorize the minimum detected input frequency. This will give you information about frequency peaks.

**Table 7.20. Parameter properties of FG_TRIGGERIN_STATS_MINFREQUENCY**

| Property | Value |
|---|---|
| Name | **FG_TRIGGERIN_STATS_MINFREQUENCY** |
| Display Name | **Minimum Frequency** |
| Type | **Double** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **Minimum 0.0**<br>**Maximum 1.25E8**<br>**Stepsize 2.220446049250313E-16** |
| Unit of measure | **Hz** |

**Example 7.19. Usage of FG_TRIGGERIN_STATS_MINFREQUENCY**

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

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

### 7.5.4.3.7. FG_TRIGGERIN_STATS_MAXFREQUENCY

The trigger system will memorize the maximum detected input frequency. This will give you information about frequency peaks.

**Table 7.21. Parameter properties of FG_TRIGGERIN_STATS_MAXFREQUENCY**

| Property | Value |
|---|---|
| Name | **FG_TRIGGERIN_STATS_MAXFREQUENCY** |
| Display Name | **Maximum Frequency** |
| Type | **Double** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **Minimum 0.0**<br>**Maximum 1.25E8**<br>**Stepsize 2.220446049250313E-16** |
| Unit of measure | **Hz** |

**Example 7.20. Usage of FG_TRIGGERIN_STATS_MAXFREQUENCY**

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

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

### 7.5.4.3.8. FG_TRIGGERIN_STATS_MINMAXFREQUENCY_CLEAR

To clear the minimum and maximum frequency measurements, write to this register. The minimum and maxumum frequency will then be the current input frequency.

**Table 7.22. Parameter properties of FG_TRIGGERIN_STATS_MINMAXFREQUENCY_CLEAR**

| Property | Value |
|---|---|
| Name | **FG_TRIGGERIN_STATS_MINMAXFREQUENCY_CLEAR** |
| Display Name | **Clear MinMax Frequency** |
| Type | **Enumeration** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Transient** |
| Allowed values | **FG_APPLY** Apply |
| Default value | **FG_APPLY** |

**Example 7.21. Usage of FG_TRIGGERIN_STATS_MINMAXFREQUENCY_CLEAR**

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

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

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

### 7.5.4.3.9. 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.

For a general explanation on events see Event.

### 7.5.4.3.10. 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.

For a general explanation on events see Event.

## 7.5.5. Sequencer

The sequencer is a powerful feature to generate multiple pulses out of one input pulse. It is available in external and software trigger mode, but not in generator mode. The sequencer multiplies an input pulse using the factor set by *FG_TRIGGER_MULTIPLY_PULSES*. The inserted pulses will have a time delay to the original signal according to the setting made for parameter *FG_TRIGGER_FRAMESPERSECOND*. Thus, the inserted pulses are not evenly distributed between the input pulses, they will be inserted with a delay specified by *FG_TRIGGER_FRAMESPERSECOND*. Hence, it is very important, that the multiplicate pulses with a parameterized delay will not cause a loss of input signals.

Let's have a look at an example. Suppose you have an external trigger source generating a pulse once every second. Your input frequency will then be 1Hz. Assume that the sequencer is set to a multiplication factor of 2 and the maximum frequency defined by *FG_TRIGGER_FRAMESPERSECOND* is set to 2.1Hz.

The trigger system will forward each external pulse into the trigger system and will also generate a second pulse 0.48 seconds later. As you can see, the multiplication frequency is chosen to be slightly higher than the doubled input frequency. This will allow the compensation of varying input frequencies.

If the time between two pulses at the input will be less than 0.96 seconds, you will loose the second pulse. Silicon Software recommends the multiplication frequency to be fast enough to not loose pulses or recommends the activation of the trigger queue for compensation. You can check for lost pulses with parameter *FG_TRIGGER_EXCEEDED_PERIOD_LIMITS*.

#### 7.5.5.1. FG_TRIGGER_MULTIPLY_PULSES

Set the trigger input multiplication factor.

**Table 7.23. Parameter properties of FG_TRIGGER_MULTIPLY_PULSES**

| Property | Value |
|---|---|
| Name | **FG_TRIGGER_MULTIPLY_PULSES** |
| Display Name | **Upscale Trigger Pulses** |
| Type | **Unsigned Integer** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum 1**<br>**Maximum 65535**<br>**Stepsize 1** |
| Default value | **1** |

**Example 7.22. Usage of FG_TRIGGER_MULTIPLY_PULSES**

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

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

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

### 7.5.6. Queue

The maximum trigger output frequency is limited to the the setting of parameter *FG_TRIGGER_FRAMESPERSECOND*. This can avoid the loss of trigger pulses in the camera which is hard to detect. In some cases it is possible, that the frequency of your external trigger source varies. To prevent the loose of trigger pulses, you can activate the trigger queue to buffer these pulses. Furthermore, the queue can be used to buffer trigger input pulses if you use the sequencer and the software trigger.

Activate the trigger queue using parameter *FG_TRIGGERQUEUE_MODE*.

The queue fill level can be monitored by parameter *FG_TRIGGERQUEUE_FILLLEVEL*. Moreover, two events allow the monitoring of the fill level. Using parameters *FG_TRIGGER_QUEUE_FILLLEVEL_EVENT_ON_THRESHOLD* and *FG_TRIGGER_QUEUE_FILLLEVEL_EVENT_OFF_THRESHOLD* it is possible to set two threshold. If the fill level exceeds the ON-threshold the respective event *FG_TRIGGER_QUEUE_FILLLEVEL_THRESHOLD_CAM0_ON* is generated. If the fill level gets less or equal than the OFF-threshold the event *FG_TRIGGER_QUEUE_FILLLEVEL_THRESHOLD_CAM0_OFF* is generated.

Note that a fill level value *n* indicates that between *n* and *n* + 1 trigger pulses have to be processed by the system. Therefore, a fill level value zero means that no more values are in the queue, but there might be still a pulse (or multiple pulses if the sequencer is used) to be processed. There exists one exception for value

zero obtained with *FG_TRIGGERQUEUE_FILLLEVEL* i.e. the parameter and not the events. This value at this parameter truly indicates that no more pulses are in the queue and all pulses have been full processed.

### 7.5.6.1. FG_TRIGGERQUEUE_MODE

Activate the queue using this parameter. Note that a queue de-activation will erase all remaining values in the queue.

**Table 7.24. Parameter properties of FG_TRIGGERQUEUE_MODE**

| Property | Value |
|---|---|
| Name | **FG_TRIGGERQUEUE_MODE** |
| Display Name | **Queue Mode** |
| Type | **Enumeration** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **FG_ON** On<br>**FG_OFF** Off |
| Default value | **FG_OFF** |

**Example 7.23. Usage of FG_TRIGGERQUEUE_MODE**

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

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

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

### 7.5.6.2. FG_TRIGGERQUEUE_FILLLEVEL

Obtain the currently queued pulses with this parameter. At maximum 2040 pulses can be queued. The queue fill level includes the input pulses, i.e. the external trigger pulses in the queue or the software trigger pulses in the queue. The fill level does not include the pulses generated by the sequencer. The fill level is zero, if the trigger system is not busy anymore i.e. no more pulses are left to be processed.

**Table 7.25. Parameter properties of FG_TRIGGERQUEUE_FILLLEVEL**

| Property | Value |
|---|---|
| Name | **FG_TRIGGERQUEUE_FILLLEVEL** |
| Display Name | **Filllevel** |
| Type | **Unsigned Integer** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **Minimum** 0<br>**Maximum** 2040<br>**Stepsize** 1 |
| Unit of measure | **pulses** |

**Example 7.24. Usage of FG_TRIGGERQUEUE_FILLLEVEL**

```
int result = 0;
```

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

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

### 7.5.6.3. FG_TRIGGER_QUEUE_FILLLEVEL_EVENT_ON_THRESHOLD

Set the ON-threshold for fill level event generation with this parameter.

**Table 7.26. Parameter properties of FG_TRIGGER_QUEUE_FILLLEVEL_EVENT_ON_THRESHOLD**

| Property | Value |
|---|---|
| Name | **FG_TRIGGER_QUEUE_FILLLEVEL_EVENT_ON_THRESHOLD** |
| Display Name | **Queue Fill Level Event On Threshold** |
| Type | **Unsigned Integer** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum 2**<br>**Maximum 2047**<br>**Stepsize 1** |
| Default value | **2047** |
| Unit of measure | **pulses** |

**Example 7.25. Usage of FG_TRIGGER_QUEUE_FILLLEVEL_EVENT_ON_THRESHOLD**

```
int result = 0;
unsigned int value = 2047;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

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

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

### 7.5.6.4. FG_TRIGGER_QUEUE_FILLLEVEL_EVENT_OFF_THRESHOLD

Set the OFF-threshold for fill level event generation with this parameter.

**Table 7.27. Parameter properties of FG_TRIGGER_QUEUE_FILLLEVEL_EVENT_OFF_THRESHOLD**

| Property | Value |
|---|---|
| Name | **FG_TRIGGER_QUEUE_FILLLEVEL_EVENT_OFF_THRESHOLD** |
| Display Name | **Queue Fill Level Event Off Threshold** |
| Type | **Unsigned Integer** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum 2**<br>**Maximum 2047**<br>**Stepsize 1** |
| Default value | **2** |
| Unit of measure | **pulses** |

**Example 7.26. Usage of FG_TRIGGER_QUEUE_FILLLEVEL_EVENT_OFF_THRESHOLD**

```
int result = 0;
unsigned int value = 2;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

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

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

### 7.5.6.5. FG_TRIGGER_QUEUE_FILLLEVEL_THRESHOLD_CAM0_ON

The event is generated if the queue fill level exceeds the ON-threshold set by parameter *FG_TRIGGER_QUEUE_FILLLEVEL_EVENT_ON_THRESHOLD*. Except for the timestamp, the event has no additional data included.

For a general explanation on events see Event.

### 7.5.6.6. FG_TRIGGER_QUEUE_FILLLEVEL_THRESHOLD_CAM0_OFF

The event is generated if the queue fill level gets less or equal than the OFF-threshold set by parameter *FG_TRIGGER_QUEUE_FILLLEVEL_EVENT_OFF_THRESHOLD*. Except for the timestamp, the event has no additional data included.

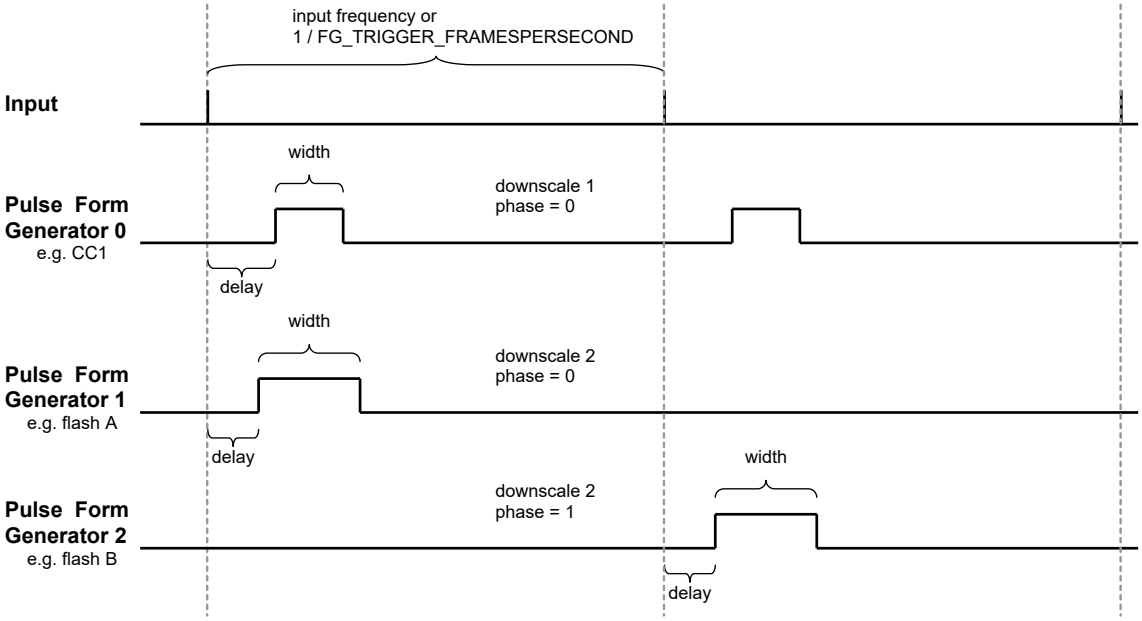
For a general explanation on events see Event.

## 7.5.7. Pulse Form Generator 0

The parameters explained previously were used to generate the trigger pulses. Next, we will need to prepare the pulses for the outputs. The area trigger system includes four individual pulse form generators. These generators define the width and delay of the output signals and also support downscaling of pulses which can be useful if different light sources are used successively. After parameterizing the pulse form generators you can arbitrarily allocate the pulse form generators to the outputs.

The following figure illustrates the output of the pulse form generators and the parameters.

**Figure 7.20. Pulse Form Generators**



Once again, note that the ranges of the parameters depend on the other settings in the pulse form generators and on paramerter *FG_TRIGGER_FRAMESPERSECOND*. If you want to increase the frequency you might need to decrease the width or delay of one of the pulse form generators.

**Equation 7.2. Dependency of Frequency and Pulse Form Generators**

$$\frac{1}{\text{fps}} > Max\left\{\begin{array}{l}\frac{Max\{\text{WIDTH0},\text{DELAY0}\}}{\text{DOWNSCALE0}},\\ \frac{Max\{\text{WIDTH1},\text{DELAY1}\}}{\text{DOWNSCALE1}},\\ \frac{Max\{\text{WIDTH2},\text{DELAY2}\}}{\text{DOWNSCALE2}},\\ \frac{Max\{\text{WIDTH3},\text{DELAY3}\}}{\text{DOWNSCALE3}}\end{array}\right\}$$

- fps = *FG_TRIGGER_FRAMESPERSECOND*
- WIDTH[0..3] = *FG_TRIGGER_PULSEFORMGEN[0..3]_WIDTH*
- DELAY[0..3] = *FG_TRIGGER_PULSEFORMGEN[0..3]_DELAY*
- DOWNSCALE[0..3] = *FG_TRIGGER_PULSEFORMGEN[0..3]_DOWNSCALE*

### 7.5.7.1. FG_TRIGGER_PULSEFORMGEN0_DOWNSCALE et al.

#### Note

This description applies also to the following parameters:
FG_TRIGGER_PULSEFORMGEN1_DOWNSCALE,
FG_TRIGGER_PULSEFORMGEN2_DOWNSCALE,
FG_TRIGGER_PULSEFORMGEN3_DOWNSCALE

The trigger pulses can be downscaled. Set the downscale factor by use of this parameter. Note the dependency between this parameter and the phase. See *FG_TRIGGER_PULSEFORMGEN[0..3]_DOWNSCALE_PHASE* for more information.

**Table 7.28. Parameter properties of FG_TRIGGER_PULSEFORMGEN0_DOWNSCALE**

| Property | Value |
|---|---|
| Name | **FG_TRIGGER_PULSEFORMGEN0_DOWNSCALE** |
| Display Name | **Downscale** |
| Type | **Unsigned Integer** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum 1**<br>**Maximum 7**<br>**Stepsize 1** |
| Default value | **1** |

**Example 7.27. Usage of FG_TRIGGER_PULSEFORMGEN0_DOWNSCALE**

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

if ((result = Fg_setParameterWithType(fg, FG_TRIGGER_PULSEFORMGEN0_DOWNSCALE, &value, 0, type)) < 0) {
    /* error handling */
}
```

```
if ((result = Fg_getParameterWithType(fg, FG_TRIGGER_PULSEFORMGEN0_DOWNSCALE, &value, 0, type)) < 0) {
    /* error handling */
}
```

### 7.5.7.2. FG_TRIGGER_PULSEFORMGEN0_DOWNSCALE_PHASE et al.

#### Note

This description applies also to the following parameters: FG_TRIGGER_PULSEFORMGEN1_DOWNSCALE_PHASE, FG_TRIGGER_PULSEFORMGEN2_DOWNSCALE_PHASE, FG_TRIGGER_PULSEFORMGEN3_DOWNSCALE_PHASE

The parameter *FG_TRIGGER_PULSEFORMGEN[0..3]_DOWNSCALE* defines the number of phases and parameter *FG_TRIGGER_PULSEFORMGEN[0..3]_DOWNSCALE_PHASE* selects the one being used. The downscale value represents the factor. For example value three will remove two out of three successive trigger pulses. The phase is used to make the selection of the pulse in the sequence. For the given example, a phase set to value zero will forward the first pulse and will remove pulses two and three of a sequence of three pulses. Check Section 7.5.7, 'Pulse Form Generator 0' for more information.

Take care of the dependency between the downscale factor and the phase. The factor has to be greater than the phase.

**Table 7.29. Parameter properties of FG_TRIGGER_PULSEFORMGEN0_DOWNSCALE_PHASE**

| Property | Value |
|---|---|
| Name | **FG_TRIGGER_PULSEFORMGEN0_DOWNSCALE_PHASE** |
| Display Name | **Phase** |
| Type | **Unsigned Integer** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum 0**<br>**Maximum 6**<br>**Stepsize 1** |
| Default value | **0** |

**Example 7.28. Usage of FG_TRIGGER_PULSEFORMGEN0_DOWNSCALE_PHASE**

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

if ((result = Fg_setParameterWithType(fg, FG_TRIGGER_PULSEFORMGEN0_DOWNSCALE_PHASE, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_TRIGGER_PULSEFORMGEN0_DOWNSCALE_PHASE, &value, 0, type)) < 0) {
    /* error handling */
}
```

### 7.5.7.3. FG_TRIGGER_PULSEFORMGEN0_DELAY et al.

#### Note

This description applies also to the following parameters: FG_TRIGGER_PULSEFORMGEN1_DELAY, FG_TRIGGER_PULSEFORMGEN2_DELAY, FG_TRIGGER_PULSEFORMGEN3_DELAY

Set a signal delay with this parameter. The unit of this parameter is µs.

**Table 7.30. Parameter properties of FG_TRIGGER_PULSEFORMGEN0_DELAY**

| Property | Value |
|---|---|
| Name | **FG_TRIGGER_PULSEFORMGEN0_DELAY** |
| Display Name | **Delay** |
| Type | **Double** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum 0.0**<br>**Maximum 3.4E7**<br>**Stepsize 0.008** |
| Default value | **0.0** |
| Unit of measure | **µs** |

**Example 7.29. Usage of FG_TRIGGER_PULSEFORMGEN0_DELAY**

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

if ((result = Fg_setParameterWithType(fg, FG_TRIGGER_PULSEFORMGEN0_DELAY, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_TRIGGER_PULSEFORMGEN0_DELAY, &value, 0, type)) < 0) {
    /* error handling */
}
```

### 7.5.7.4. FG_TRIGGER_PULSEFORMGEN0_WIDTH et al.

**Note**

This description applies also to the following parameters: FG_TRIGGER_PULSEFORMGEN1_WIDTH, FG_TRIGGER_PULSEFORMGEN2_WIDTH, FG_TRIGGER_PULSEFORMGEN3_WIDTH

Set the signal width, i.e. the active time of the output signal. The unit of this parameter is µs.

**Table 7.31. Parameter properties of FG_TRIGGER_PULSEFORMGEN0_WIDTH**

| Property | Value |
|---|---|
| Name | **FG_TRIGGER_PULSEFORMGEN0_WIDTH** |
| Display Name | **Signal Width** |
| Type | **Double** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum 1.0**<br>**Maximum 6.8E7**<br>**Stepsize 0.008** |
| Default value | **4.0** |
| Unit of measure | **µs** |

**Example 7.30. Usage of FG_TRIGGER_PULSEFORMGEN0_WIDTH**

```
int result = 0;
double value = 4.0;
```

```
const enum FgParamTypes type = FG_PARAM_TYPE_DOUBLE;

if ((result = Fg_setParameterWithType(fg, FG_TRIGGER_PULSEFORMGEN0_WIDTH, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_TRIGGER_PULSEFORMGEN0_WIDTH, &value, 0, type)) < 0) {
    /* error handling */
}
```

### 7.5.8. Pulse Form Generator 1

The settings for pulse form generator 1 are equal to those of pulse form generator 0. Please read Section 7.5.7, 'Pulse Form Generator 0' for a detailed description.

### 7.5.9. Pulse Form Generator 2

The settings for pulse form generator 2 are equal to those of pulse form generator 0. Please read Section 7.5.7, 'Pulse Form Generator 0' for a detailed description.

### 7.5.10. Pulse Form Generator 3

The settings for pulse form generator 3 are equal to those of pulse form generator 0. Please read Section 7.5.7, 'Pulse Form Generator 0' for a detailed description.

### 7.5.11. CC Signal Mapping

The CameraLink interface specifies four camera input signals, i.e. CC1, CC2, CC3 and CC4. Usually the camera will use one particular CC-input-signal to trigger the data acquisition and define the exposure time. Please, consult the vendor's manual of your camera to identify the required signals and their mapping to the CC1--CC4 lines.

The trigger system of this AcquisitionApplets provides several possibilities of mapping sources to the CC lines:

- Pulse form generators

  The pulse form generators are the main output sources of the trigger system. You can either directly one of the four sources to a CC line or invert the signal if you need low active signals.

- Pulse form generators of other camera processes

  If you are using a multiple camera applet, you can also select a pulse form generator source of another camera. This will allow you to use the same trigger configuration for both cameras.

- Ground or Vcc if a CC line is not used or you want to temporarily deactivate or activate the line.

- The input bypass

  The trigger system will ignore the signal length of the input signals. If you want to bypass an input directly to the output you can select the specific input or its inverted version.

#### 7.5.11.1. FG_TRIGGERCC_SELECT0 et al.

**Note**

This description applies also to the following parameters: FG_TRIGGERCC_SELECT1, FG_TRIGGERCC_SELECT2, FG_TRIGGERCC_SELECT3

**Table 7.32. Parameter properties of FG_TRIGGERCC_SELECT0**

| Property | Value | |
|---|---|---|
| Name | **FG_TRIGGERCC_SELECT0** | |
| Display Name | **CC0** | |
| Type | **Enumeration** | |
| Access policy | **Read/Write/Change** | |
| Storage policy | **Persistent** | |
| Allowed values | **VCC** | Vcc |
| | **GND** | Gnd |
| | **CAM_A_PULSEGEN0** | Cam A Pulse Generator 0 |
| | **CAM_A_PULSEGEN1** | Cam A Pulse Generator 1 |
| | **CAM_A_PULSEGEN2** | Cam A Pulse Generator 2 |
| | **CAM_A_PULSEGEN3** | Cam A Pulse Generator 3 |
| | **CAM_A_NOT_PULSEGEN0** | Not Cam A Pulse Generator 0 |
| | **CAM_A_NOT_PULSEGEN1** | Not Cam A Pulse Generator 1 |
| | **CAM_A_NOT_PULSEGEN2** | Not Cam A Pulse Generator 2 |
| | **CAM_A_NOT_PULSEGEN3** | Not Cam A Pulse Generator 3 |
| | **BYPASS_GPI_0** | Bypass GPI 0 |
| | **NOT_BYPASS_GPI_0** | Not Bypass GPI 0 |
| | **BYPASS_GPI_1** | Bypass GPI 1 |
| | **NOT_BYPASS_GPI_1** | Not Bypass GPI 1 |
| | **BYPASS_GPI_2** | Bypass GPI 2 |
| | **NOT_BYPASS_GPI_2** | Not Bypass GPI 2 |
| | **BYPASS_GPI_3** | Bypass GPI 3 |
| | **NOT_BYPASS_GPI_3** | Not Bypass GPI 3 |
| | **BYPASS_GPI_4** | Bypass GPI 4 (not available on LightBridge) |
| | **NOT_BYPASS_GPI_4** | Not Bypass GPI 4 (not available on LightBridge) |
| | **BYPASS_GPI_5** | Bypass GPI 5 (not available on LightBridge) |
| | **NOT_BYPASS_GPI_5** | Not Bypass GPI 5 (not available on LightBridge) |
| | **BYPASS_GPI_6** | Bypass GPI 6 (not available on LightBridge) |
| | **NOT_BYPASS_GPI_6** | Not Bypass GPI 6 (not available on LightBridge) |
| | **BYPASS_GPI_7** | Bypass GPI 7 (not available on LightBridge) |
| | **NOT_BYPASS_GPI_7** | Not Bypass GPI 7 (not available on LightBridge) |
| | **BYPASS_FRONT_GPI_0** | Bypass Front-GPI 0 |
| | **NOT_BYPASS_FRONT_GPI_0** | Not Bypass Front-GPI 0 |
| | **BYPASS_FRONT_GPI_1** | Bypass Front-GPI 1 |
| | **NOT_BYPASS_FRONT_GPI_1** | Not Bypass Front-GPI 1 |
| | **BYPASS_FRONT_GPI_2** | Bypass Front-GPI 2 |
| | **NOT_BYPASS_FRONT_GPI_2** | Not Bypass Front-GPI 2 |
| | **BYPASS_FRONT_GPI_3** | Bypass Front-GPI 3 |
| | **NOT_BYPASS_FRONT_GPI_3** | Not Bypass Front-GPI 3 |
| | **PULSEGEN0** | Pulse Generator 0 |
| | **PULSEGEN1** | Pulse Generator 1 |
| | **PULSEGEN2** | Pulse Generator 2 |
| | **PULSEGEN3** | Pulse Generator 3 |
| | **NOT_PULSEGEN0** | Not Pulse Generator 0 |
| | **NOT_PULSEGEN1** | Not Pulse Generator 1 |
| | **NOT_PULSEGEN2** | Not Pulse Generator 2 |
| | **NOT_PULSEGEN3** | Not Pulse Generator 3 |
| Default value | **CC_NOT_PULSEGEN0** | |

**Example 7.31. Usage of FG_TRIGGERCC_SELECT0**

```
int result = 0;
```

```
int value = CC_NOT_PULSEGEN0;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_TRIGGERCC_SELECT0, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_TRIGGERCC_SELECT0, &value, 0, type)) < 0) {
    /* error handling */
}
```

### 7.5.12. Digital Output

The microEnable 5 marathon/LightBridge ACL frame grabber have eight general purpose outputs (GPOs) and in addition 2 Front GPOs.

In contrast, the microEnable 5 LightBridge frame grabbers have four general purpose outputs (GPOs) and in addition 2 Front GPOs. On LightBridge grabbers, you can select a source for all eight + 2 outputs, even they are not mapped to a connector pin. Ensure to use the correct outputs.

The trigger system of this applet provides several possibilities of mapping sources to the digital output signals:

- Pulse form generators

  The pulse form generators are the main output sources of the trigger system. You can either directly bypass one of the four sources to a digital output or invert its signal.

- Ground or Vcc if a digital output is not used or you want to manually set the signal level.

- The input bypass

  The trigger system will ignore the signal length of the input signals. If you want to bypass an input directly to the output you can select the specific input or its inverted version.

#### 7.5.12.1. FG_TRIGGEROUT_SELECT_GPO_0 et al.

**Note**

This description applies also to the following parameters: FG_TRIGGEROUT_SELECT_GPO_1, FG_TRIGGEROUT_SELECT_GPO_2, FG_TRIGGEROUT_SELECT_GPO_3, FG_TRIGGEROUT_SELECT_GPO_4, FG_TRIGGEROUT_SELECT_GPO_5, FG_TRIGGEROUT_SELECT_GPO_6, FG_TRIGGEROUT_SELECT_GPO_7

Select the source for the output on the respective GPO.

**Using legacy values "PULSEGEN_0 to PULSEGEN_3" and "NOT_PULSEGEN_0 to NOT_PULSEGEN_3".**

Note that values "PULSEGEN_0 to PULSEGEN_3" and "NOT_PULSEGEN_0 to NOT_PULSEGEN_3" are legacy parameters. If you set the parameter to one of these values the respective trigger module will be used. Reading the parameter will always include the camera index.

For example if you set FG_TRIGGEROUT_SELECT_GPO_0 for DMA index 1 to PULSEGEN_0 on a multi-cam applet you will use the trigger module of camera B. Reading the parameter results in returning CAM_B_PULSEGEN0.

**Table 7.33. Parameter properties of FG_TRIGGEROUT_SELECT_GPO_0**

| Property | Value | |
|---|---|---|
| Name | **FG_TRIGGEROUT_SELECT_GPO_0** | |
| Display Name | **Output GPO 0** | |
| Type | **Enumeration** | |
| Access policy | **Read/Write/Change** | |
| Storage policy | **Persistent** | |
| Allowed values | **VCC** | Vcc |
| | **GND** | Gnd |
| | **CAM_A_PULSEGEN0** | Cam A Pulse Generator 0 |
| | **CAM_A_PULSEGEN1** | Cam A Pulse Generator 1 |
| | **CAM_A_PULSEGEN2** | Cam A Pulse Generator 2 |
| | **CAM_A_PULSEGEN3** | Cam A Pulse Generator 3 |
| | **CAM_A_NOT_PULSEGEN0** | Not Cam A Pulse Generator 0 |
| | **CAM_A_NOT_PULSEGEN1** | Not Cam A Pulse Generator 1 |
| | **CAM_A_NOT_PULSEGEN2** | Not Cam A Pulse Generator 2 |
| | **CAM_A_NOT_PULSEGEN3** | Not Cam A Pulse Generator 3 |
| | **BYPASS_GPI_0** | Bypass GPI 0 |
| | **NOT_BYPASS_GPI_0** | Not Bypass GPI 0 |
| | **BYPASS_GPI_1** | Bypass GPI 1 |
| | **NOT_BYPASS_GPI_1** | Not Bypass GPI 1 |
| | **BYPASS_GPI_2** | Bypass GPI 2 |
| | **NOT_BYPASS_GPI_2** | Not Bypass GPI 2 |
| | **BYPASS_GPI_3** | Bypass GPI 3 |
| | **NOT_BYPASS_GPI_3** | Not Bypass GPI 3 |
| | **BYPASS_GPI_4** | Bypass GPI 4 (not available on LightBridge) |
| | **NOT_BYPASS_GPI_4** | Not Bypass GPI 4 (not available on LightBridge) |
| | **BYPASS_GPI_5** | Bypass GPI 5 (not available on LightBridge) |
| | **NOT_BYPASS_GPI_5** | Not Bypass GPI 5 (not available on LightBridge) |
| | **BYPASS_GPI_6** | Bypass GPI 6 (not available on LightBridge) |
| | **NOT_BYPASS_GPI_6** | Not Bypass GPI 6 (not available on LightBridge) |
| | **BYPASS_GPI_7** | Bypass GPI 7 (not available on LightBridge) |
| | **NOT_BYPASS_GPI_7** | Not Bypass GPI 7 (not available on LightBridge) |
| | **BYPASS_FRONT_GPI_0** | Bypass Front-GPI 0 |
| | **NOT_BYPASS_FRONT_GPI_0** | Not Bypass Front-GPI 0 |
| | **BYPASS_FRONT_GPI_1** | Bypass Front-GPI 1 |
| | **NOT_BYPASS_FRONT_GPI_1** | Not Bypass Front-GPI 1 |
| | **BYPASS_FRONT_GPI_2** | Bypass Front-GPI 2 |
| | **NOT_BYPASS_FRONT_GPI_2** | Not Bypass Front-GPI 2 |
| | **BYPASS_FRONT_GPI_3** | Bypass Front-GPI 3 |
| | **NOT_BYPASS_FRONT_GPI_3** | Not Bypass Front-GPI 3 |
| | **PULSEGEN0** | Pulse Generator 0 |
| | **PULSEGEN1** | Pulse Generator 1 |
| | **PULSEGEN2** | Pulse Generator 2 |
| | **PULSEGEN3** | Pulse Generator 3 |
| | **NOT_PULSEGEN0** | Not Pulse Generator 0 |
| | **NOT_PULSEGEN1** | Not Pulse Generator 1 |
| | **NOT_PULSEGEN2** | Not Pulse Generator 2 |
| | **NOT_PULSEGEN3** | Not Pulse Generator 3 |
| Default value | **CAM_A_NOT_PULSEGEN0** | |

**Example 7.32. Usage of FG_TRIGGEROUT_SELECT_GPO_0**

```
int result = 0;
```

```
int value = CAM_A_NOT_PULSEGEN0;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_TRIGGEROUT_SELECT_GPO_0, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_TRIGGEROUT_SELECT_GPO_0, &value, 0, type)) < 0) {
    /* error handling */
}
```

### 7.5.12.2. FG_TRIGGEROUT_SELECT_FRONT_GPO_0 et al.

#### Note

This description applies also to the following parameters: FG_TRIGGEROUT_SELECT_FRONT_GPO_1

Select the source for the output on the repsective Front GPO.

#### Using legacy values "PULSEGEN_0 to PULSEGEN_3" and "NOT_PULSEGEN_0 to NOT_PULSEGEN_3".

Note that values "PULSEGEN_0 to PULSEGEN_3" and "NOT_PULSEGEN_0 to NOT_PULSEGEN_3" are legacy parameters. If you set the parameter to one of these values the respective trigger module will be used. Reading the parameter will always include the camera index.

For example if you set FG_TRIGGEROUT_SELECT_GPO_0 for DMA index 1 to PULSEGEN_0 on a multi-cam applet you will use the trigger module of camera B. Reading the parameter results in returning CAM_B_PULSEGEN0.

**Table 7.34. Parameter properties of FG_TRIGGEROUT_SELECT_FRONT_GPO_0**

| Property | Value | |
|---|---|---|
| Name | **FG_TRIGGEROUT_SELECT_FRONT_GPO_0** | |
| Display Name | **Output Front GPO 0** | |
| Type | **Enumeration** | |
| Access policy | **Read/Write/Change** | |
| Storage policy | **Persistent** | |
| Allowed values | **VCC** | Vcc |
| | **GND** | Gnd |
| | **CAM_A_PULSEGEN0** | Cam A Pulse Generator 0 |
| | **CAM_A_PULSEGEN1** | Cam A Pulse Generator 1 |
| | **CAM_A_PULSEGEN2** | Cam A Pulse Generator 2 |
| | **CAM_A_PULSEGEN3** | Cam A Pulse Generator 3 |
| | **CAM_A_NOT_PULSEGEN0** | Not Cam A Pulse Generator 0 |
| | **CAM_A_NOT_PULSEGEN1** | Not Cam A Pulse Generator 1 |
| | **CAM_A_NOT_PULSEGEN2** | Not Cam A Pulse Generator 2 |
| | **CAM_A_NOT_PULSEGEN3** | Not Cam A Pulse Generator 3 |
| | **BYPASS_GPI_0** | Bypass GPI 0 |
| | **NOT_BYPASS_GPI_0** | Not Bypass GPI 0 |
| | **BYPASS_GPI_1** | Bypass GPI 1 |
| | **NOT_BYPASS_GPI_1** | Not Bypass GPI 1 |
| | **BYPASS_GPI_2** | Bypass GPI 2 |
| | **NOT_BYPASS_GPI_2** | Not Bypass GPI 2 |
| | **BYPASS_GPI_3** | Bypass GPI 3 |
| | **NOT_BYPASS_GPI_3** | Not Bypass GPI 3 |
| | **BYPASS_GPI_4** | Bypass GPI 4 (not available on LightBridge) |
| | **NOT_BYPASS_GPI_4** | Not Bypass GPI 4 (not available on LightBridge) |
| | **BYPASS_GPI_5** | Bypass GPI 5 (not available on LightBridge) |
| | **NOT_BYPASS_GPI_5** | Not Bypass GPI 5 (not available on LightBridge) |
| | **BYPASS_GPI_6** | Bypass GPI 6 (not available on LightBridge) |
| | **NOT_BYPASS_GPI_6** | Not Bypass GPI 6 (not available on LightBridge) |
| | **BYPASS_GPI_7** | Bypass GPI 7 (not available on LightBridge) |
| | **NOT_BYPASS_GPI_7** | Not Bypass GPI 7 (not available on LightBridge) |
| | **BYPASS_FRONT_GPI_0** | Bypass Front-GPI 0 |
| | **NOT_BYPASS_FRONT_GPI_0** | Not Bypass Front-GPI 0 |
| | **BYPASS_FRONT_GPI_1** | Bypass Front-GPI 1 |
| | **NOT_BYPASS_FRONT_GPI_1** | Not Bypass Front-GPI 1 |
| | **BYPASS_FRONT_GPI_2** | Bypass Front-GPI 2 |
| | **NOT_BYPASS_FRONT_GPI_2** | Not Bypass Front-GPI 2 |
| | **BYPASS_FRONT_GPI_3** | Bypass Front-GPI 3 |
| | **NOT_BYPASS_FRONT_GPI_3** | Not Bypass Front-GPI 3 |
| | **PULSEGEN0** | Pulse Generator 0 |
| | **PULSEGEN1** | Pulse Generator 1 |
| | **PULSEGEN2** | Pulse Generator 2 |
| | **PULSEGEN3** | Pulse Generator 3 |
| | **NOT_PULSEGEN0** | Not Pulse Generator 0 |
| | **NOT_PULSEGEN1** | Not Pulse Generator 1 |
| | **NOT_PULSEGEN2** | Not Pulse Generator 2 |
| | **NOT_PULSEGEN3** | Not Pulse Generator 3 |
| Default value | **CAM_A_NOT_PULSEGEN0** | |

**Example 7.33. Usage of FG_TRIGGEROUT_SELECT_FRONT_GPO_0**

```
int result = 0;
```

```
int value = CAM_A_NOT_PULSEGEN0;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_TRIGGEROUT_SELECT_FRONT_GPO_0, &value, 0, type)) < 0) {
    /* error handling */
}

if ((result = Fg_getParameterWithType(fg, FG_TRIGGEROUT_SELECT_FRONT_GPO_0, &value, 0, type)) < 0) {
    /* error handling */
}
```

### 7.5.12.3. Statistics

The output statistics module counts the number of output pulses. The source can be selected by parameter *FG_TRIGGEROUT_STATS_SOURCE*. The count value can be read from parameter *FG_TRIGGEROUT_STATS_PULSECOUNT*. Parameter *FG_TRIGGEROUT_STATS_SOURCE* also selects the source for the missing frame detection functionality.

#### 7.5.12.3.1. FG_TRIGGER_EXCEEDED_PERIOD_LIMITS

This read-only register has value **FG_YES** if the input signal frequency exceeded the maximum allowed frequency defined by parameter *FG_TRIGGER_FRAMESPERSECOND*. If the queue is enabled, the register is only set if the queue is full and cannot store a new input pulse. Reading the register will not reset it. It is required to reset the register by writing to *FG_TRIGGER_EXCEEDED_PERIOD_LIMITS_CLEAR*.

**Table 7.35. Parameter properties of FG_TRIGGER_EXCEEDED_PERIOD_LIMITS**

| Property | Value |
|---|---|
| Name | **FG_TRIGGER_EXCEEDED_PERIOD_LIMITS** |
| Display Name | **Trigger Exceeded Period Limits** |
| Type | **Enumeration** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **FG_YES** Yes<br>**FG_NO** No |

**Example 7.34. Usage of FG_TRIGGER_EXCEEDED_PERIOD_LIMITS**

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

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

#### 7.5.12.3.2. FG_TRIGGER_EXCEEDED_PERIOD_LIMITS_CLEAR

Reset *FG_TRIGGER_EXCEEDED_PERIOD_LIMITS* with this parameter.

**Table 7.36. Parameter properties of FG_TRIGGER_EXCEEDED_PERIOD_LIMITS_CLEAR**

| Property | Value |
|---|---|
| Name | **FG_TRIGGER_EXCEEDED_PERIOD_LIMITS_CLEAR** |
| Display Name | **Clear Exceeded Period Limits Register** |
| Type | **Enumeration** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Transient** |
| Allowed values | **FG_APPLY** Apply |
| Default value | **FG_APPLY** |

**Example 7.35. Usage of FG_TRIGGER_EXCEEDED_PERIOD_LIMITS_CLEAR**

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

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

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

### 7.5.12.3.3. FG_TRIGGEROUT_STATS_SOURCE

**Table 7.37. Parameter properties of FG_TRIGGEROUT_STATS_SOURCE**

| Property | Value |
|---|---|
| Name | **FG_TRIGGEROUT_STATS_SOURCE** |
| Display Name | **Source** |
| Type | **Enumeration** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **PULSEGEN0** Pulse Generator 0<br>**PULSEGEN1** Pulse Generator 1<br>**PULSEGEN2** Pulse Generator 2<br>**PULSEGEN3** Pulse Generator 3 |
| Default value | **PULSEGEN0** |

**Example 7.36. Usage of FG_TRIGGEROUT_STATS_SOURCE**

```
int result = 0;
int value = PULSEGEN0;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

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

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

### 7.5.12.3.4. FG_TRIGGEROUT_STATS_PULSECOUNT

Output pulse count read register. Select the source for the pulse counter by parameter *FG_TRIGGEROUT_STATS_SOURCE*.

**Table 7.38. Parameter properties of FG_TRIGGEROUT_STATS_PULSECOUNT**

| Property | Value |
|---|---|
| Name | **FG_TRIGGEROUT_STATS_PULSECOUNT** |
| Display Name | **Pulse Count** |
| Type | **Unsigned Integer** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **Minimum** 0<br>**Maximum** 65535<br>**Stepsize** 1 |
| Unit of measure | **pulses** |

**Example 7.37. Usage of FG_TRIGGEROUT_STATS_PULSECOUNT**

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

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

### 7.5.12.3.5. FG_TRIGGEROUT_STATS_PULSECOUNT_CLEAR

Output pulse count register clear.

**Table 7.39. Parameter properties of FG_TRIGGEROUT_STATS_PULSECOUNT_CLEAR**

| Property | Value |
|---|---|
| Name | **FG_TRIGGEROUT_STATS_PULSECOUNT_CLEAR** |
| Display Name | **Clear pulse counter** |
| Type | **Enumeration** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Transient** |
| Allowed values | **FG_APPLY** Apply |
| Default value | **FG_APPLY** |

**Example 7.38. Usage of FG_TRIGGEROUT_STATS_PULSECOUNT_CLEAR**

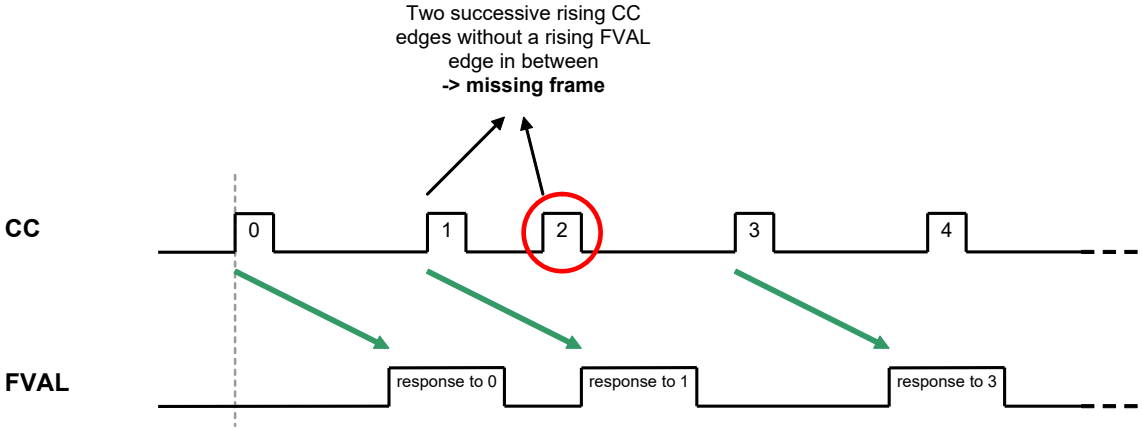
```
int result = 0;
int value = FG_APPLY;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

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

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

### 7.5.12.3.6. FG_MISSING_CAMERA_FRAME_RESPONSE

This applet is equipped with a detection of missing camera frame responses to trigger pulses. If the camera will not send a frame for each output trigger pulse, the register is set to **FG_YES** until cleared by writing to parameter *FG_MISSING_CAMERA_FRAME_RESPONSE_CLEAR*.

The idea of the frame loss detection is that for every trigger pulse generated by the trigger system, the camera will send a frame to the frame grabber. If a trigger pulse gets lost, or the camera cannot send a frame, this register will be set to **FG_YES**. Technically, between two output signal edges, a incoming image has to exist. Or in other words: There must not be two or more successive trigger start edges without a valid frame in between. The following figure illustrates the behavior.

**Figure 7.21. Missing Camera Frame Response**




The pulse form generator allocated to the camera trigger signal line carrying the image trigger pulses has to be selected by *FG_TRIGGEROUT_STATS_SOURCE*. The missing frame response system might not work correct for all camera models due to different timings.

## Select Camera Control/Trigger Signal Line

Take care to select the pulse form generator feeding the camera trigger signal line which carries the image trigger pulses by setting parameter *FG_TRIGGEROUT_STATS_SOURCE* to the respective source.

## Acquisition Start Before Trigger Activation

Keep in mind to start the acquisition before activating the trigger. Otherwise, the trigger pulses sent will get lost. Also keep in mind, that any changes of the camera configuration might result in invalid data transfers.

## Events for Missing Frame Response

If you want to monitor the exact moment of a missing frame responses, and the exact number of missing frames use event *FG_MISSING_CAM0_FRAME_RESPONSE*.

**Table 7.40. Parameter properties of FG_MISSING_CAMERA_FRAME_RESPONSE**

| Property | Value |
|---|---|
| Name | **FG_MISSING_CAMERA_FRAME_RESPONSE** |
| Display Name | **Missing Camera Frame Response** |
| Type | **Enumeration** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **FG_YES** Yes<br>**FG_NO** No |

**Example 7.39. Usage of FG_MISSING_CAMERA_FRAME_RESPONSE**

```
int result = 0;
int value = FG_NO;
```

```
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

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

### 7.5.12.3.7. FG_MISSING_CAMERA_FRAME_RESPONSE_CLEAR

Clear the *FG_MISSING_CAMERA_FRAME_RESPONSE* flag by writing to this parameter.

**Table 7.41. Parameter properties of FG_MISSING_CAMERA_FRAME_RESPONSE_CLEAR**

| Property | Value |
|---|---|
| Name | **FG_MISSING_CAMERA_FRAME_RESPONSE_CLEAR** |
| Display Name | **Clear Frame Response Register** |
| Type | **Enumeration** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Transient** |
| Allowed values | **FG_APPLY** Apply |
| Default value | **FG_APPLY** |

**Example 7.40. Usage of FG_MISSING_CAMERA_FRAME_RESPONSE_CLEAR**

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

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

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

### 7.5.12.3.8. FG_TRIGGER_EXCEEDED_PERIOD_LIMITS_CAM0

The event is generated for each lost input trigger pulse. A trigger loss can occur if the input frequency is higher than the maximum allowed frequency set by parameter *FG_TRIGGER_FRAMESPERSECOND* . If the trigger queue is enabled, the events will only be generated if the queue is full i.e. for overflows. In generator trigger modes, the events will not be generated. Except for the timestamp, the event has no additional data included.

For a general explanation on events see Event.

### 7.5.12.3.9. FG_MISSING_CAM0_FRAME_RESPONSE

The missing camera frame response event is generated for each camera output trigger pulse with no frame response. If the mechanism is compatible with the used camera and set up correct, the number of events is equal to the number of lost frames. Except for the timestamp, the event has no additional data included.

For a general explanation on events see Event.

## 7.5.13. Output Event

You can select one of the trigger outputs to generate a software event. i.e. a software callback function. Use parameter *FG_TRIGGER_OUTPUT_EVENT_SELECT* to specified the source pulse form generator for the event. The events name itself is *FG_TRIGGER_OUTPUT_CAM0*.

For a general explanation on events check Event.

### 7.5.13.1. FG_TRIGGER_OUTPUT_EVENT_SELECT

Select the source for the output event *FG_TRIGGER_OUTPUT_CAM0* with this register. One of the pulse form generators can be selected.

**Table 7.42. Parameter properties of FG_TRIGGER_OUTPUT_EVENT_SELECT**

| Property | Value |
|---|---|
| Name | **FG_TRIGGER_OUTPUT_EVENT_SELECT** |
| Display Name | **Output Event Select** |
| Type | **Enumeration** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **PULSEGEN0** Pulse Generator 0<br>**PULSEGEN1** Pulse Generator 1<br>**PULSEGEN2** Pulse Generator 2<br>**PULSEGEN3** Pulse Generator 3 |
| Default value | **PULSEGEN0** |

**Example 7.41. Usage of FG_TRIGGER_OUTPUT_EVENT_SELECT**

```
int result = 0;
int value = PULSEGEN0;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

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

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

### 7.5.13.2. FG_TRIGGER_OUTPUT_CAM0

This event is generated for each start of an output trigger pulse. The respective pulse form generator has to be selected by parameter *FG_TRIGGER_OUTPUT_EVENT_SELECT*. Except for the timestamp, the event has no additional data included. Keep in mind that a high output frequency can cause high interrupt rates which might slow down the system.

# Chapter 8. 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 frame grabber. 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.

- 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.

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.

## 8.1. FG_FILLLEVEL

The fill-level of the frame grabber 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 8.1. Parameter properties of FG_FILLLEVEL**

| Property | Value |
|---|---|
| Name | **FG_FILLLEVEL** |
| Display Name | **Buffer fill level** |
| Type | **Unsigned Integer** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **Minimum 0**<br>**Maximum 100**<br>**Stepsize 1** |
| Unit of measure | **%** |

**Example 8.1. 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 */
}
```

## 8.2. 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 8.2. 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**<br>**Maximum 1**<br>**Stepsize 1** |

**Example 8.2. 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 */
}
```

## 8.3. 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.

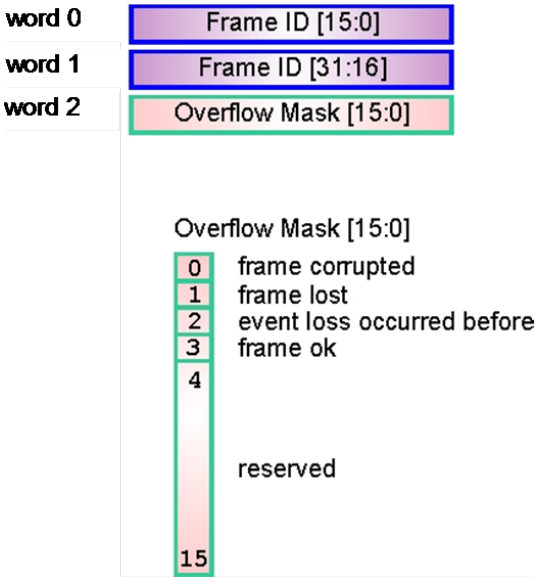
The Basler Framegrabber SDK and pylon SDK via GenTL enables an application to get these event notifications about certain state changes at the data flow from camera to RAM and the image and trigger processing as well. Please consult the Basler Framegrabber SDK, pylon SDK or GenTL documentation for more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

### 8.3.1. FG_OVERFLOW_CAM0

Overflow events are generated for each corrupted or lost frame. In contrast to the other events presented in this document, the overflow event transports data, namely the type of overflow, the image number and the

timestamp. The following figure illustrates the event data. Data is included in a 64 Bit data packet. The first 32 Bit include the frame number. Bits 32 to 47 include an overflow mask.

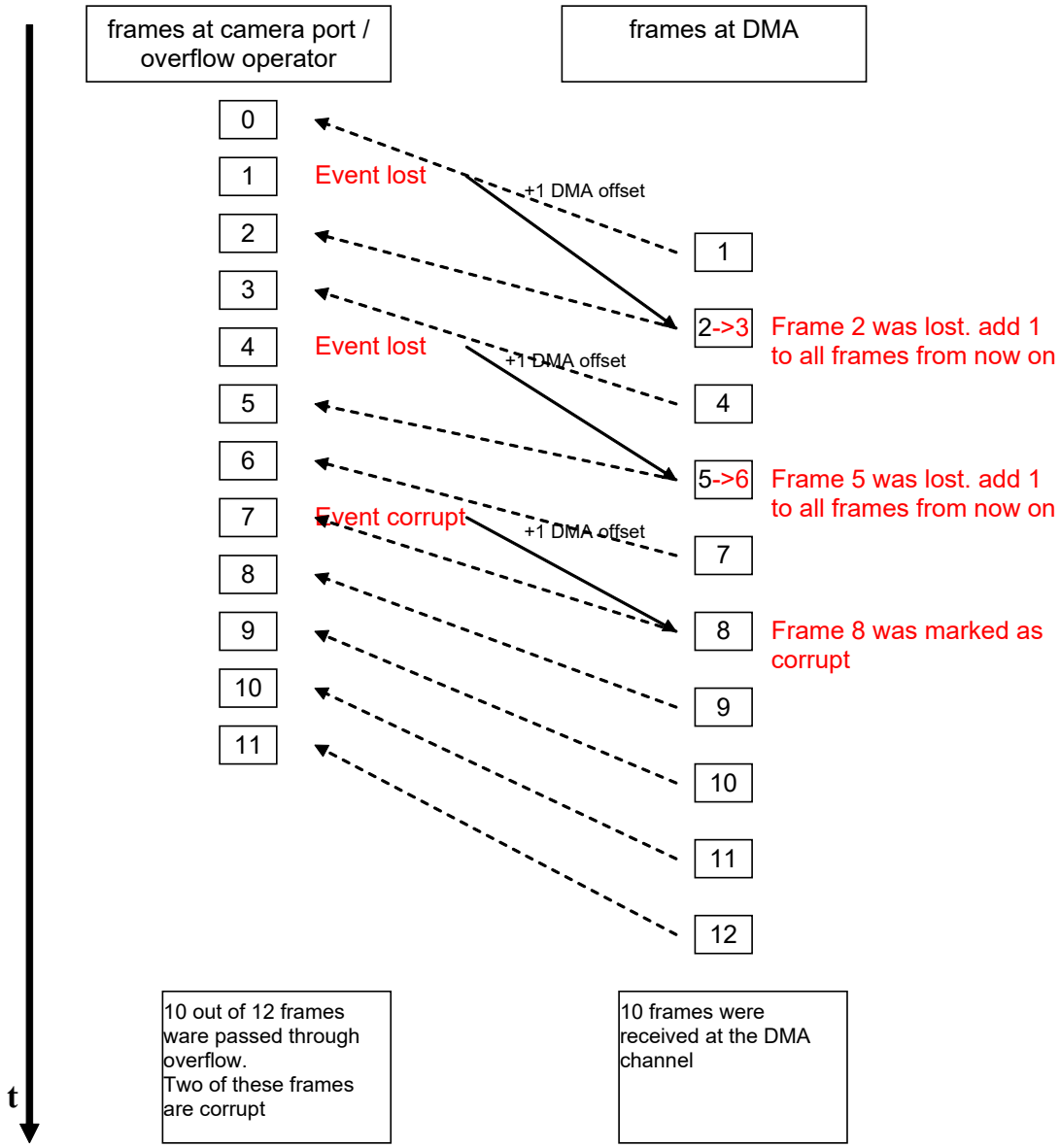
**Figure 8.1. Illustration of Overflow Data Packet**




Note that the frame number is reset on acquisition start. Also note that the first frame will have frame number zero, while a DMA transfer starts with frame number one. The frame number is a 32 Bit value. If it's maximum is reached, it will start from zero again. Keep in mind that on a 64 Bit runtime, the DMA transfer number will be a 64 Bit value. If the frame corrupted flag is set, the frame with the frame number in the event is corrupted i.e. it will not have it's full length but is still transfered via DMA channel. If the frame lost flag is set, the frame with the frame number in the event was fully discarded. No DMA transfer will exist for this frame. The corrupted frame flag and the frame lost flag will never occur for the same event. The flag "event loss occured before" is an additional security mechanism. It means that an event has been lost. This can only happen at very high event rates and should not happen under normal conditions.

The analysis of the overflow events depends on the user requirements. In the following, an example is shown on how to ensure the integrity if the DMA data by analyzing the events and DMA transfers.

**Figure 8.2. Analysis of Overflow Data**




In the example, two frames got lost and one is marked as corrupted. As the events are not synchronous with the DMA transfers, for analysis a software queue (push and pull) is required to allocate the events to the DMA transfers.

# Chapter 9. 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. Next, the acquisitions of both grabbers have to be started and finally, the trigger generation has to be enabled, generally by setting *FG_TRIGGERSTATE* to active. Now the camera will start sending image data and the grabbers acquire those synchronously. More information can be found in the trigger chapter Chapter 7, '*Trigger*'.

## 9.1. 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 9.1. 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**<br>**Maximum 256**<br>**Stepsize 1** |
| Default value | **1** |
| Unit of measure | **image** |

**Example 9.1. 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 */
}
```

## 9.2. 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 9.2. 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**<br>**Maximum 255**<br>**Stepsize 1** |
| Default value | **0** |
| Unit of measure | **image** |

**Example 9.2. 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 10. Lookup Table

This Acquisition Applet includes a full resolution lookup table (LUT). Settings are applied to the acquired images just before transferring them to the host PC. Thus, it is the last pre-processing step on the frame grabber.

A lookup table includes one entry for every allowed input pixel value. The pixel value will be replaced by the value of the lookup table element. In other words, a new value is assigned to each pixel value. This can be used for image quality enhancements such as an added offset, a gain factor or gamma correction which can be performed by use of the processing module of this applet in a convenient way (see Module Chapter 11, '*Processing*'). The lookup table can also be loaded with custom values. Application areas are custom image enhancements or correct pixel classifications.

This applet is processing data with an internal resolution of 16 bits. But the lookup table has 12 input bits i.e. pixel values can be in the range [0, 4095]. For each of these 4096 elements, a table entry exists containing a new output value. The new values are in the range from 0 to 65536. Since this applet uses 16 bit internally, consider that all values need to represent this value range. This LUT is applied to all pixel values before *FG_FORMAT* is applied. The input values for the LUT are aligned to the most significant bit (MSB).

In the following the parameters to use the lookup table are explained. Parameter *FG_LUT_TYPE* is important to be set correctly as it defines the lookup table operation mode.

## 10.1. FG_LUT_ENABLE

It is possible to disable the functionality of this lookup table. The intenal processor enables a convenient way to improve the image quality using parameters such as offset, gain and gamma. By disabling the lookup table the processing functions are not available anymore. See category Chapter 11, '*Processing*' for a more detailed documentation concerning this. Set this parameter to **FG_ON** to use the look up table. By default it is set to **FG_OFF** disabling the lookup table functionality itself and the related processing functions.

**Table 10.1. Parameter properties of FG_LUT_ENABLE**

| Property | Value |
|---|---|
| Name | **FG_LUT_ENABLE** |
| Display Name | **Enabled** |
| Type | **Enumeration** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **FG_ON** On<br>**FG_OFF** Off |
| Default value | **FG_OFF** |

**Example 10.1. Usage of FG_LUT_ENABLE**

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

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

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

## 10.2. FG_LUT_TYPE

There exist two basic possibilities to use and configure the lookup table. One possibility is to use the internal processor which allows a convenient way to improve the image quality using parameters such as offset, gain and gamma. Check category Chapter 11, '*Processing*' for more detailed documentation. Set this parameter to **LUT_TYPE_PROCESSING** to use the processor.

The second possibility to use the lookup table is to load a file containing custom values to the lookup table. Set the parameter to **LUT_TYPE_CUSTOM** to enable the possibility to load a custom file with lookup table entries.

Beside these two possibilities it is always possible to directly write to the lookup table entries using the field parameter *FG_LUT_VALUE*. The use of these parameters will overwrite the settings made with the processor or the custom input file. Vice versa, changing a processing parameter or loading a custom lookup table file, will overwrite the settings made by the field parameters.

**Table 10.2. Parameter properties of FG_LUT_TYPE**

| Property | Value | |
|---|---|---|
| Name | **FG_LUT_TYPE** | |
| Display Name | **Type** | |
| Type | **Enumeration** | |
| Access policy | **Read/Write/Change** | |
| Storage policy | **Persistent** | |
| Allowed values | **LUT_TYPE_PROCESSING**<br>**LUT_TYPE_CUSTOM** | Processor<br>User file |
| Default value | **LUT_TYPE_PROCESSING** | |

**Example 10.2. Usage of FG_LUT_TYPE**

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

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

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

## 10.3. FG_LUT_VALUE

**Table 10.3. Parameter properties of FG_LUT_VALUE**

| Property | Value |
|---|---|
| Name | **FG_LUT_VALUE** |
| Display Name | **LUT Values** |
| Type | **Unsigned Integer Field** |
| Field Size | **4096** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Transient** |
| Default value | **0** |

**Example 10.3. Usage of FG_LUT_VALUE**

```
int result = 0;

FieldParameterInt access;
```

```
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

for (unsigned int i = 0; i < 4096; ++i)
{
    access.index = i;
    access.value = 0;

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

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

## 10.4. FG_LUT_CUSTOM_FILE

If parameter *FG_LUT_TYPE* is set to **LUT_TYPE_CUSTOM**, the according path and filename to the file containing the custom lookup table entries can be set here. If the file is valid, the file values will be loaded to the lookup table. If the file is invalid, the call to this parameter will return an error.

A convenient way of getting a draft file, is to save the current lookup table settings to file using parameter *FG_LUT_SAVE_FILE*.

Please make sure to activate the Type of LUT *FG_LUT_TYPE* to "UserFile"/**LUT_TYPE_CUSTOM** in order to make the changes and file names taking effect.

This section describes the file formats which are in use to fill the so called look-up tables (LUT). The purpose of a LUT is a transformation of pixel values from a input (source) image to the pixel values of an output image. This transformation is done by a kind of table, which contains the assignment between these pixel values (input pixel values - output pixel values). Basically the LUT is defined for gray format and color formats as well. When defining a LUT for color formats, the definition of tables has to be done for each color component. The LUT file format consists of 2 parts:

- Header section containing control and description information.
- Main section containing the assignment table for transforming pixel values form a source (input) image to a destination (output) image.

The following example shows how a grey scale lookup table description could look like:

```
# Lut data file v1.1
id=3;
nrOfElements=4096;
format=0;
number=0;
0,0;
1,1;
2,2;
3,3;
4,4;
5,5;
6,6;
…
4095,4095;
```

General Properties:

- File format extension should be ".lut"
- LUT file format is an ASCII file format consisting of multiple lines of data.
- Lines are defined by a line separator a <CR> <LF> line feed (0x3D 0x0D 0x0A).
- Lines consist of key / value pairs. Key and value are separated by "=". The value has to be followed by a semicolon ; (0x3B)

- Formats consist of header data, containing control information and the assignment table for a specific color component (gray / red, green, blue).
- Basically the LUT file color format follows the same rules as the gray image format. In addition, due to the fact, that each color component can has its own transformation, the definitions are repeated for each color component.

The following example shows how a color scale lookup table description could look like:

```
# Lut data file v1.1
[red]
id=0;
nrOfElements=256;
format=0;
number=0;
0,0;
1,1;
..
255,255;
[green]
id=1;
nrOfElements=256;
format=0;
number=0;
0,0;
1,1;
..
255,255;
[blue]
id=2;
nrOfElements=256;
format=0;
number=0;
0,0;
1,1;
..
255,255;
```

A more detailed explanation of the lookup table file format can be found in the Basler Framegrabber API manual.

**Table 10.4. Parameter properties of FG_LUT_CUSTOM_FILE**

| Property | Value |
|---|---|
| Name | **FG_LUT_CUSTOM_FILE** |
| Display Name | **Load File** |
| Type | **String** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Default value | **""** |

**Example 10.4. Usage of FG_LUT_CUSTOM_FILE**

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

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

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

## 10.5. FG_LUT_SAVE_FILE

To save the current lookup table configuration to a file, write the according output filename to this parameter. Keep in mind that you need to have full write access to the specified path.

Writing the current lookup table settings to a file is also a convenient way to exploit the settings made by the processor. Moreover, you will get a draft version of the lookup table file format. The values in the output file can directly be used to be loaded to the lookup table again using parameter *FG_LUT_CUSTOM_FILE*.

**Table 10.5. Parameter properties of FG_LUT_SAVE_FILE**

| Property | Value |
|---|---|
| Name | **FG_LUT_SAVE_FILE** |
| Display Name | **Save File** |
| Type | **String** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Transient** |
| Default value | **""** |

**Example 10.5. Usage of FG_LUT_SAVE_FILE**

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

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

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

## 10.6. Applet Properties

In the following, some properties of the lookup table implementation are listed.

### 10.6.1. FG_LUT_IMPLEMENTATION_TYPE

In this applet, a full lookup table is implemented and can be setup in a custom way. By default a linear representation is performed.

**Table 10.6. Parameter properties of FG_LUT_IMPLEMENTATION_TYPE**

| Property | Value |
|---|---|
| Name | **FG_LUT_IMPLEMENTATION_TYPE** |
| Display Name | **LUT Implementation Type** |
| Type | **Enumeration** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **LUT_IMPLEMENTATION_FULL_LUT** Full LUT<br>**LUT_IMPLEMENTATION_KNEELUT** KneeLUT |

**Example 10.6. Usage of FG_LUT_IMPLEMENTATION_TYPE**

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

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

### 10.6.2. FG_LUT_IN_BITS

This applet is using 12 lookup table input bits.

**Table 10.7. Parameter properties of FG_LUT_IN_BITS**

| Property | Value |
|---|---|
| Name | **FG_LUT_IN_BITS** |
| Display Name | **LUT In Bits** |
| Type | **Unsigned Integer** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Unit of measure | **bit** |

**Example 10.7. Usage of FG_LUT_IN_BITS**

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

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

### 10.6.3. FG_LUT_OUT_BITS

This applet is using 16 lookup table output bits.

**Table 10.8. Parameter properties of FG_LUT_OUT_BITS**

| Property | Value |
|---|---|
| Name | **FG_LUT_OUT_BITS** |
| Display Name | **LUT Out Bits** |
| Type | **Unsigned Integer** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Unit of measure | **bit** |

**Example 10.8. Usage of FG_LUT_OUT_BITS**

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

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

# Chapter 11. Processing

A convenient way to improve the image quality are the processing parameters. Using these parameters an offset, gain and gamma correction can be performed. Moreover, the image can be inverted.

## Processor Activation

The processing parameters use the lookup table for determination of the correction values. For activation of the processing parameters, set *FG_LUT_TYPE* of category lookup table to **LUT_TYPE_PROCESSING** . Otherwise, parameter changes will have no effect.

All transformations apply in the following order:

1. Offset Correction, range [-1.0, +1.0], identity = 0
2. Gain Correction, range [0, 2^12[, identity = 1.0
3. Gamma Correction, range ]0, inf], identity = 1.0
4. Invert, identity = 'off'

In this applet, a full lookup table with m = 12 input bits and n = 16 outputs bits is used to perform the corrections. Values are determined by

**Equation 11.1. LUT Processor without Inversion**

$$Output(x) = \left[\left[gain * \left(\frac{x}{2^{12}-1} + offset\right)\right]^{\frac{1}{gamma}}\right] * \left(2^{16}-1\right).$$

If the inversion is used, output values are determined by

**Equation 11.2. LUT Processor with Inversion**

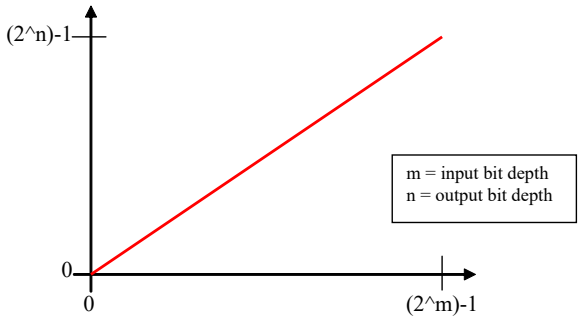
$$Output(x) = 2^{16} - 1 - \left[\left[gain * \left(\frac{x}{2^{12}-1} + offset\right)\right]^{\frac{1}{gamma}}\right] * \left(2^{16}-1\right),$$

where x represents the input pixel value i.e. is in the range from 0 to 2^12 - 1. If the determined output value is less than 0, it will be set to 0. If the determined output value is greater than 2^16 - 1 it is set to 2^16 - 1.

Color applets process each color component separately using the same processing parameters for each component.

If no parameters are changed, i.e. they are set to identity, the output values will be equal to the input values as shown in the figure below. In the following, you will find detailed explanations for all processing parameters.

**Figure 11.1. Lookup Table Processing: Identity**



## 11.1. FG_PROCESSING_OFFSET

The offset is a relative value added to each pixel, which leads to a behavior similar to a brightness controller. A relative offset means, that e. g. 0.5 adds half of the total brightness to each pixel. In absolute numbers when using 8 bit/pixel, 128 is added to each pixel (0.5 x 255 = 127.5). If you rather want to add an absolute value to each pixel do the following calculation: e. g. add -51 to an 8 bit/pixel offset = -51 / 255 = -0.2.Figure 11.2 shows an example of an offset.

**Figure 11.2. Lookup Table Processing: Offset**

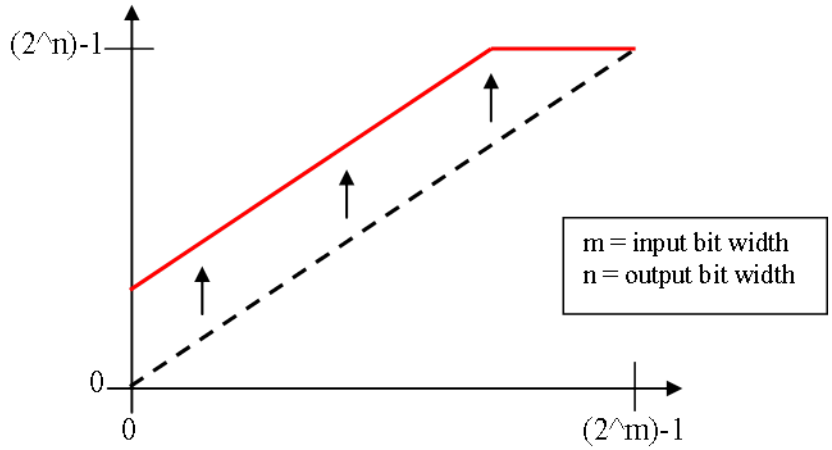



**Table 11.1. Parameter properties of FG_PROCESSING_OFFSET**

| Property | Value |
|---|---|
| Name | **FG_PROCESSING_OFFSET** |
| Display Name | **Offset** |
| Type | **Double** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum -1.0**<br>**Maximum 1.0**<br>**Stepsize 2.220446049250313E-16** |
| Default value | **0.0** |

**Example 11.1. Usage of FG_PROCESSING_OFFSET**

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

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

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

## 11.2. FG_PROCESSING_GAIN

The gain is a multiplicative coefficient applied to each pixel, which leads to a behavior similar to a contrast controller. Each pixel value will be multiplied with the given value. For identity select value 1.0.

**Figure 11.3. Lookup Table Processing: Gain**

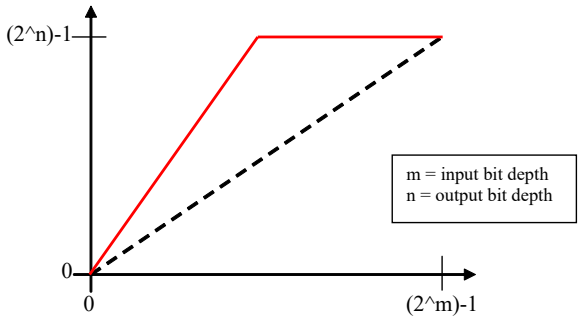



**Table 11.2. Parameter properties of FG_PROCESSING_GAIN**

| Property | Value |
|---|---|
| Name | **FG_PROCESSING_GAIN** |
| Display Name | **Gain** |
| Type | **Double** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum** 0.0<br>**Maximum** 4096.0<br>**Stepsize** 2.220446049250313E-16 |
| Default value | **1.0** |

**Example 11.2. Usage of FG_PROCESSING_GAIN**

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

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

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

## 11.3. FG_PROCESSING_GAMMA

The gamma correction is a power-law transformation applied to each pixel. Normalized pixel values $p$ ranging [0, 1.0] transform like $p' = p^{1/gamma}$ .

**Figure 11.4. Lookup Table Processing: Gamma**

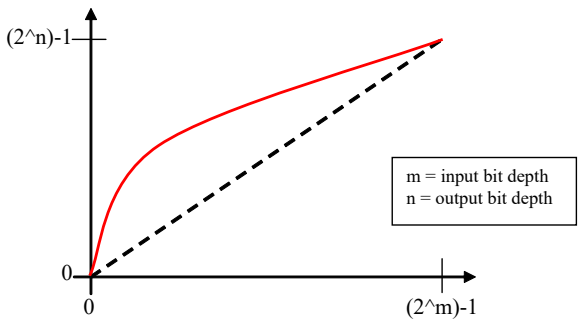


**Table 11.3. Parameter properties of FG_PROCESSING_GAMMA**

| Property | Value |
|---|---|
| Name | **FG_PROCESSING_GAMMA** |
| Display Name | **Gamma** |
| Type | **Double** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum** -1000.0<br>**Maximum** 1000.0<br>**Stepsize** 2.220446049250313E-16 |
| Default value | **1.0** |

**Example 11.3. Usage of FG_PROCESSING_GAMMA**

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

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

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

## 11.4. FG_PROCESSING_INVERT

When *FG_PROCESSING_INVERT* is set to **FG_ON**, the output is the negative of the input. Normalized pixel values $p$ ranging [0, 1.0] transform to $p' = 1 - p$.

**Figure 11.5. Lookup Table Processing: Invert**

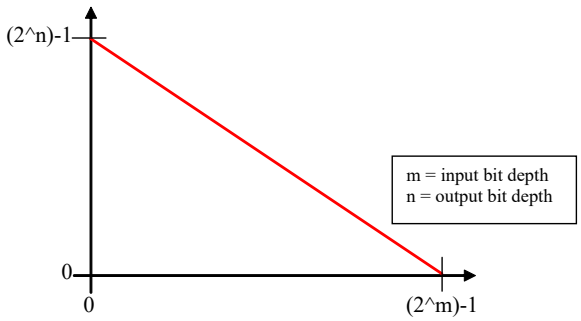



**Table 11.4. Parameter properties of FG_PROCESSING_INVERT**

| Property | Value |
|---|---|
| Name | **FG_PROCESSING_INVERT** |
| Display Name | **Invert** |
| Type | **Enumeration** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **FG_ON** On<br>**FG_OFF** Off |
| Default value | **FG_OFF** |

**Example 11.4. Usage of FG_PROCESSING_INVERT**

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

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

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

# Chapter 12. Output Format

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

## 12.1. 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*.

This applet has no integrated color converter. If you select a different color pixel format between the input and output no valid output data can be generated.

This applet supports the following output formats:

- **FG_GRAY**: 8 bit grayscale format
- **FG_GRAY16**: 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.

**Table 12.1. 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_GRAY** Gray 8bit<br>**FG_GRAY16** Gray 16bit |
| Default value | **FG_GRAY** |

**Example 12.1. Usage of FG_FORMAT**

```
int result = 0;
int value = FG_GRAY;
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 */
}
```

## 12.2. 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 12.2. Parameter properties of FG_BITALIGNMENT**

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

**Example 12.2. 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 */
}
```

## 12.3. 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 12.3. 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**<br>**Maximum 128**<br>**Stepsize 1** |
| Unit of measure | **bit** |

**Example 12.3. 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 */
}
```

## 12.4. 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 frame grabber. 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 12.4. 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**<br>**Maximum 15**<br>**Stepsize 1** |
| Default value | **0** |
| Unit of measure | **bit** |

**Example 12.4. 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 13. 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. Thus, for this aplet parameters *FG_CAMERA_LINK_CAMTYPE* and *FG_USEDVAL* have no influence on the images, if the camera simulator is activated.

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

**Figure 13.1. 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.

## 13.1. 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*.

The camera simulator will use 8bit values for all supported pixel formats.

**Table 13.1. 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<br>**FG_CAMERASIMULATOR** Simulator |
| Default value | **FG_CAMPORT** |

**Example 13.1. 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 */
}
```

## 13.2. 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 13.2. 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<br>**Maximum** 65536<br>**Stepsize** 1 |
| Default value | **1024** |
| Unit of measure | **pixel** |

**Example 13.2. 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 */
}
```

## 13.3. 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 13.3. 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**<br>**Maximum 65536**<br>**Stepsize 1** |
| Default value | **0** |
| Unit of measure | **pixel** |

**Example 13.3. 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 */
}
```

## 13.4. 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 13.4. 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**<br>**Maximum 65536**<br>**Stepsize 1** |
| Default value | **1024** |
| Unit of measure | **pixel** |

**Example 13.4. 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 */
}
```

## 13.5. 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 13.5. 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**<br>**Maximum 65536**<br>**Stepsize 1** |
| Default value | **0** |
| Unit of measure | **pixel** |

**Example 13.5. 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 */
}
```

## 13.6. FG_CAMERASIMULATOR_PATTERN

The simulator will generate pixel value ramps from 0 to 255.

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 13.6. 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<br>**FG_VERTICAL** Vertical<br>**FG_DIAGONAL** Diagonal |
| Default value | **FG_DIAGONAL** |

**Example 13.6. 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 */
}
```

## 13.7. 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 13.7. 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<br>**Maximum** 255<br>**Stepsize** 1 |
| Default value | **0** |
| Unit of measure | **pixel value** |

**Example 13.7. 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 */
}
```

## 13.8. 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 13.8. 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<br>**FG_OFF** Off |
| Default value | **FG_ON** |

**Example 13.8. 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 */
}
```

## 13.9. 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 13.9. 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**<br>**FG_LINERATE**<br>**FG_FRAMERATE** | Pixel Frequency<br>Line Rate<br>Frame Rate |
| Default value | **FG_FRAMERATE** | |

**Example 13.9. Usage of FG_CAMERASIMULATOR_SELECT_MODE**

```
int result = 0;
int value = FG_FRAMERATE;
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 */
}
```

## 13.10. 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 13.10. 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**<br>**Maximum**<br>**Stepsize** | **0.12499999999999997**<br>**-1.0**<br>**0.25** |
| Default value | **39.375** | |
| Unit of measure | **MHz** | |

**Example 13.10. 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 */
}
```

## 13.11. 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 13.11. 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**<br>**Maximum -35714.28571428572**<br>**Stepsize 7.0E-11** |
| Default value | **10240.0** |
| Unit of measure | **Hz** |

**Example 13.11. 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 */
}
```

## 13.12. FG_CAMERASIMULATOR_FRAMERATE

This parameter sets the frame rate of the generated images. For parameter *FG_TRIGGER_FRAMESPERSECOND* only frame values up to the upper value limit of *FG_CAMERASIMULATOR_FRAMERATE* are valid.

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 13.12. 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**<br>**Maximum -35714.28571428572**<br>**Stepsize 7.0E-11** |
| Default value | **10.0** |
| Unit of measure | **Hz** |

**Example 13.12. 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 */
}
```

## 13.13. 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 is a Camera Link applet, output CC1 of the respective camera port is used as camera simulator trigger input. The rising edge of the trigger will be used. Thus you should set the output polarity of the trigger module to high active.

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 13.13. 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<br>**RISING_EDGE_TRIGGERS_LINE** Rising Edge Triggers Line<br>**RISING_EDGE_TRIGGERS_FRAME** Rising Edge Triggers Frame |
| Default value | **SIMULATION_FREE_RUN** |

**Example 13.13. 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 */
}
```

## 13.14. FG_CAMERASIMULATOR_ACTIVE

**Table 13.14. 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<br>**Maximum** 2000<br>**Stepsize** 1 |
| Unit of measure | **pixel** |

**Example 13.14. 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 */
}
```

## 13.15. FG_CAMERASIMULATOR_PASSIVE

**Table 13.15. 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**<br>**Maximum 2000**<br>**Stepsize 1** |
| Unit of measure | **pixel** |

**Example 13.15. 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 14. 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.

## 14.1. 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 14.1. 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**<br>**Maximum 2147483646**<br>**Stepsize 1** |
| Default value | **1000000** |
| Unit of measure | **seconds** |

**Example 14.1. 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 */
}
```

## 14.2. FG_APPLET_VERSION

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

**Table 14.2. 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** |

**Example 14.2. 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 */
}
```

## 14.3. FG_APPLET_REVISION

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

**Table 14.3. 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** |

**Example 14.3. 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 */
}
```

## 14.4. FG_APPLET_ID

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

**Table 14.4. 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 14.4. 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 */
}
```

## 14.5. 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 14.5. 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 14.5. 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 */
}
```

## 14.6. 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 14.6. 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 14.6. 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 */
}
```

## 14.7. 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 14.7. Parameter properties of FG_DMASTATUS**

| Property | Value |
|---|---|
| Name | **FG_DMASTATUS** |
| Display Name | **DMA Status** |
| Type | **Unsigned Integer** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 14.7. 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 */
}
```

## 14.8. FG_CAMSTATUS

The camera status shows whether the camera clock signal can be recognized by frame grabber or not. If value "1" is determined from this read parameter, the grabber recognized a camera clock signal.

**Table 14.8. 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**<br>**Maximum 1**<br>**Stepsize 1** |

**Example 14.8. 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 */
}
```

## 14.9. 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, provided by CameraLink interface. Shows if CL PixelClock is available.
- 1 = CameraLval, provided by CameraLink interface. Shows if CameraLink LVAL is available, representing a line being transferred into frame grabber.
- 2 = CameraFval, provided by CameraLink interface. Shows if CameraLink FVAL is available, representing frames being transferred into frame grabber. Not relevant for standard line scan applets.
- 3 = Camera CC1 Signal, NOT provided by this frame grabber.
- 4 = ExTrg / external trigger, NOT provided by this frame grabber.
- 5 = BufferOverflow
- 6 = BufStatus, LSB

- 7 = BufStatus, MSB

**Table 14.9. 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**<br>**Maximum 255**<br>**Stepsize 1** |

**Example 14.9. 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 */
}
```

## 14.10. FG_SYSTEMMONITOR_FPGA_TEMPERATURE

Returns the current FGPA die temperature.

**Table 14.10. 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**<br>**Maximum 1000.0**<br>**Stepsize 0.0** |
| Unit of measure | **Celsius** |

**Example 14.10. 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 */
}
```

## 14.11. FG_SYSTEMMONITOR_FPGA_VCC_INT

Returns the current FPGA internal voltage.

**Table 14.11. 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**<br>**Maximum 1000.0**<br>**Stepsize 0.0** |
| Unit of measure | **V** |

**Example 14.11. 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 */
}
```

## 14.12. FG_SYSTEMMONITOR_FPGA_VCC_AUX

Returns the current FPGA Vcc auxiliary voltage.

**Table 14.12. 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**<br>**Maximum 1000.0**<br>**Stepsize 0.0** |
| Unit of measure | **V** |

**Example 14.12. 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 */
}
```

## 14.13. FG_SYSTEMMONITOR_FPGA_VCC_BRAM

Returns the current FPGA Vcc of the BlockRAM voltage.

**Table 14.13. 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**<br>**Maximum 1000.0**<br>**Stepsize 0.0** |
| Unit of measure | **V** |

**Example 14.13. 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 */
}
```

## 14.14. FG_SYSTEMMONITOR_CURRENT_LINK_WIDTH

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 14.14. Parameter properties of FG_SYSTEMMONITOR_CURRENT_LINK_WIDTH**

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_CURRENT_LINK_WIDTH** |
| Display Name | **Current Link Width** |
| Type | **Unsigned Integer** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **Minimum 0**<br>**Maximum 15**<br>**Stepsize 0** |
| Unit of measure | **lanes** |

**Example 14.14. Usage of FG_SYSTEMMONITOR_CURRENT_LINK_WIDTH**

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

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

## 14.15. 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 14.15. 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<br>**Maximum** 1000.0<br>**Stepsize** 0.0 |
| Unit of measure | **GB/s** |

**Example 14.15. 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 */
}
```

## 14.16. FG_SYSTEMMONITOR_PCIE_LINK_GEN2_CAPABLE

Returns if PCIe generation 2 is supported by current applet of the frame grabber.

**Table 14.16. Parameter properties of FG_SYSTEMMONITOR_PCIE_LINK_GEN2_CAPABLE**

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_PCIE_LINK_GEN2_CAPABLE** |
| Display Name | **PCIe Link Gen 2 Capable** |
| Type | **Enumeration** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **FG_YES** Yes<br>**FG_NO** No |

**Example 14.16. Usage of FG_SYSTEMMONITOR_PCIE_LINK_GEN2_CAPABLE**

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

if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_PCIE_LINK_GEN2_CAPABLE, &value, 0, type)) < 0) {
    /* error handling */
}
```

## 14.17. FG_SYSTEMMONITOR_PCIE_LINK_PARTNER_GEN2_CAPABLE

Returns if the expected PCIe generation 2 is supported by the partner. The partner would be the mainboard or in detail the corresponding PCIe interface on the host side.

**Table 14.17. Parameter properties of FG_SYSTEMMONITOR_PCIE_LINK_PARTNER_GEN2_CAPABLE**

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_PCIE_LINK_PARTNER_GEN2_CAPABLE** |
| Display Name | **PCIe Link Partner Gen 2 Capable** |
| Type | **Enumeration** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **FG_YES** Yes<br>**FG_NO** No |

**Example 14.17. Usage of FG_SYSTEMMONITOR_PCIE_LINK_PARTNER_GEN2_CAPABLE**

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

if ((result = Fg_getParameterWithType(fg, FG_SYSTEMMONITOR_PCIE_LINK_PARTNER_GEN2_CAPABLE, &value, 0, type)) < 0) {
    /* error handling */
}
```

## 14.18. FG_SYSTEMMONITOR_PCIE_TRAINED_PAYLOAD_SIZE

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

**Table 14.18. 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<br>**Maximum** 1024<br>**Stepsize** 0 |
| Unit of measure | **byte** |

**Example 14.18. 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 */
}
```

## 14.19. FG_SYSTEMMONITOR_EXTENSION_CONNECTOR_PRESENT

Returns if a extension connector is present on the frame grabber board.

**Table 14.19. Parameter properties of FG_SYSTEMMONITOR_EXTENSION_CONNECTOR_PRESENT**

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_EXTENSION_CONNECTOR_PRESENT** |
| Display Name | **Extension Connector Present** |
| Type | **Enumeration** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **FG_YES** Yes<br>**FG_NO** No |

**Example 14.19. Usage of FG_SYSTEMMONITOR_EXTENSION_CONNECTOR_PRESENT**

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

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

## 14.20. FG_ALTERNATIVE_BOARD_DETECTION

Returns the current state of the alternative frame grabber PCIe board detection algorithm. If value = FG_OFF, the Silicon Software default algorithm is used. If value = FG_ON, an alternative board detection algorithm is used.

This parameter is used for support purposes only.

**Table 14.20. Parameter properties of FG_ALTERNATIVE_BOARD_DETECTION**

| Property | Value |
|---|---|
| Name | **FG_ALTERNATIVE_BOARD_DETECTION** |
| Display Name | **Alternative Board Detection** |
| Type | **Enumeration** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **FG_ON** On<br>**FG_OFF** Off |

**Example 14.20. Usage of FG_ALTERNATIVE_BOARD_DETECTION**

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

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

## 14.21. FG_SYSTEMMONITOR_POCL_STATE_PORT_A

The parameter allows to read the current state of the Power over Camera Link (PoCL) state machine on the Port A connector.

The individual states indicate the following:

**FG_INITIALIZE** : This state has a duration of 100 ms. During this period, PoCL detection as well as PoCL operation is off. This way, the board establishes a defined initial state with no voltage applied.

**FG_POCL_CONNECTION_SENSE** : This state has a duration of 600 ms. It follows directly of state Initialize. During this state, the controller finds out if the connected camera is PoCL-capable or not.

- If a PoCL camera is detected, the PoCL state machine switches to state PoCL_Wait_for_Connection.
- If a CL camera without PoCL support is detected, the PoCL state machine switches to state CL_Wait_for_Connection.
- If a disconnect or disturbances are detected, the state machine switches back to state Initialize and starts again.

(The controller applies a test current and waits for 600 ms. Then, the voltage drop is measured. There are two thresholds: Is the measured value between both thresholds, the PoCL state machine switches to state PoCL_Wait_for_Connection. Is the measured value lower than both thresholds, the PoCL state machine switches to state CL_Wait_for_Connection. Is the measured value higher than both thresholds, the connection is either broken or disturbed. In this case, the PoCL state machine switches to state Initialize and starts again.)

**FG_POCL_WAIT_FOR_CONNECTION** : This state has a duration of 1.8 seconds. It follows directly of state **FG_POCL_CONNECTION_SENSE** in case a a power-over capable camera is detected. During this time, the controller waits and checks if the information about the availability of a power-over capable camera remains stable:

- If it is stable, the state machine switches to state **FG_POCL_CAMERA_DETECTED** and powers the camera.
- If it is not stable, the state machine switches back to state Initialize and starts again.

To ensure that a PoCL camera was not detected erroneously (due to disturbances), during state **FG_POCL_WAIT_FOR_CONNECTION** the controller checks if the measured voltage drop remains stable between the two thresholds values during the 1.8 seconds. If the measured voltage remains stable, a PoCL-capable camera is assumed, the state machine switches to state **FG_POCL_CAMERA_DETECTED** , and the camera is powered. Rises the measured voltage higher the upper threshold value, or falls it below the lower threshold value, there is a disturbance. The state machine switches to state Initialize and starts again.

**FG_POCL_CAMERA_DETECTED** : This state has a duration of up to 4 seconds. The camera is powered. The controller waits for the camera to get ready and for receiving a clock signal from the camera.

- If a clock is detected (within maximally 4s), the camera is ready for operation. The state machine switches to state **FG_POCL_CAMERA_CLOCK_DETECTED** .
- If no clock is detected (during maximally 4s), the state machine switches to state Initialize and starts again.

**FG_POCL_CAMERA_CLOCK_DETECTED** : The camera is ready for operation.

- As long as the state machine receives the clock signal from the camera, the state machine remains in this state.
- If there is no clock signal for more than 400 ms, the state machine switches to state Initialize. (It is assumed that either the camera has been disconnected, or an error has occurred.)

**FG_CL_WAIT_FOR_CONNECTION** : This state has a duration of 100ms. It follows directly of state FG_POCL_CONNECTION_SENSE in case a CL camera without PoCL support is detected. The test current is switched off. The system waits for 100ms to allow the charges to drain slowly. After this timespan, the state machine switches to state **FG_CL_CAMERA_DETECTED** , and ground (GND) is connected.

**FG_CL_CAMERA_DETECTED** : This state has a duration of up to 4 seconds. The connected camera has been identified as not PoCL-capable. The controller waits for the camera to get ready and for receiving a clock signal from the camera.

- If a clock is detected (within maximally 4s), the camera is ready for operation. The state machine switches to state **FG_CL_CAMERA_CLOCK_DETECTED** .

- If during 4s no clock is detected, the camera is not ready for operation. The state machine switches to state Initialize and starts again.

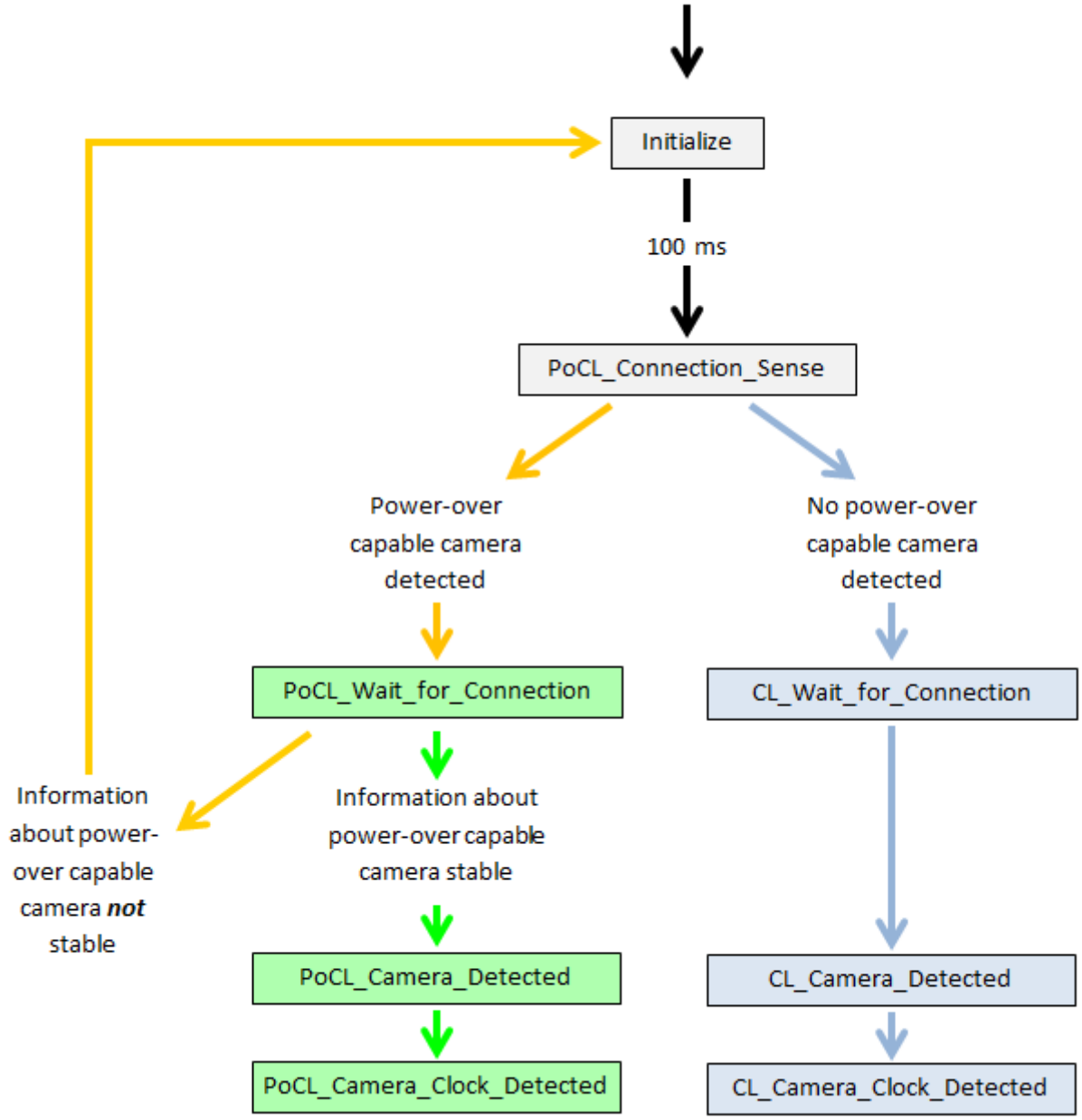
**FG_CL_CAMERA_CLOCK_DETECTED** : The camera is ready for operation.

- As long as the state machine receives the clock signal from the camera, the state machine remains in this state.
- If there is no clock signal for more than 1 s, the state machine switches to state Initialize. (It is assumed that either the camera has been disconnected, or an error has occurred.)

**FG_POCL_DISABLE** : PoCL is completely disabled for this the frame grabber. A PoCL camera needs to be powered from external power supply.

PoCL state machine decision flow:

**Figure 14.1. PoCL States**



## Power Watchdog

Additional security mechanism: The voltage level is permanently monitored.

- Is the voltage for 2 ms higher than the the lower threshold value while **no** PoCL voltage is applied: A short circuit is assumed. The state machine switches to state Initialize.
- Is the voltage for 2 ms lower than the the upper threshold value while PoCL voltage **is** applied: The occurrence of an error is assumed. The state machine switches to state Initialize.

**Table 14.21. Parameter properties of FG_SYSTEMMONITOR_POCL_STATE_PORT_A**

| Property | Value | |
|---|---|---|
| Name | **FG_SYSTEMMONITOR_POCL_STATE_PORT_A** | |
| Display Name | **PoCL State Port A** | |
| Type | **Enumeration** | |
| Access policy | **Read-Only** | |
| Storage policy | **Transient** | |
| Allowed values | **FG_INITIALIZE** | Initialize |
| | **FG_POCL_CONNECTION_SENSE** | PoCL Connection Sense |
| | **FG_POCL_WAIT_FOR_CONNECTION** | PoCL wait for Connection |
| | **FG_POCL_CAMERA_DETECTED** | PoCL Camera and Cable Detected |
| | **FG_POCL_CAMERA_CLOCK_DETECTED** | PoCL Camera Clock Detected |
| | **FG_CL_WAIT_FOR_CONNECTION** | CL wait for Connection |
| | **FG_CL_CAMERA_DETECTED** | CL Camera Detected |
| | **FG_CL_CAMERA_CLOCK_DETECTED** | CL Camera Clock Detected |
| | **FG_POCL_DISABLED** | PoCL Disabled |

**Example 14.21. Usage of FG_SYSTEMMONITOR_POCL_STATE_PORT_A**

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

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

## 14.22. FG_SYSTEMMONITOR_POCL_STATE_PORT_B

Returns the Power over CameraLink state of CameraLink Port B. Please see documentation of Section 14.21, 'FG_SYSTEMMONITOR_POCL_STATE_PORT_A'.

**Table 14.22. Parameter properties of FG_SYSTEMMONITOR_POCL_STATE_PORT_B**

| Property | Value | |
|---|---|---|
| Name | **FG_SYSTEMMONITOR_POCL_STATE_PORT_B** | |
| Display Name | **PoCL State Port B** | |
| Type | **Enumeration** | |
| Access policy | **Read-Only** | |
| Storage policy | **Transient** | |
| Allowed values | **FG_INITIALIZE** | Initialize |
| | **FG_POCL_CONNECTION_SENSE** | PoCL Connection Sense |
| | **FG_POCL_WAIT_FOR_CONNECTION** | PoCL wait for Connection |
| | **FG_POCL_CAMERA_DETECTED** | PoCL Camera and Cable Detected |
| | **FG_POCL_CAMERA_CLOCK_DETECTED** | PoCL Camera Clock Detected |
| | **FG_CL_WAIT_FOR_CONNECTION** | CL wait for Connection |
| | **FG_CL_CAMERA_DETECTED** | CL Camera Detected |
| | **FG_CL_CAMERA_CLOCK_DETECTED** | CL Camera Clock Detected |
| | **FG_POCL_DISABLED** | PoCL Disabled |

**Example 14.22. Usage of FG_SYSTEMMONITOR_POCL_STATE_PORT_B**

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

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

## 14.23. FG_SYSTEMMONITOR_FPGA_DNA

The parameter *FG_SYSTEMMONITOR_FPGA_DNA* provides the 57 bit unique FPGA DNA as an integer value.

**Table 14.23. Parameter properties of FG_SYSTEMMONITOR_FPGA_DNA**

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_FPGA_DNA** |
| Display Name | **FPGA DNA** |
| Type | **Unsigned Integer (64 Bit)** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **Minimum 0**<br>**Maximum 144115188075855872**<br>**Stepsize 0** |

**Example 14.23. Usage of FG_SYSTEMMONITOR_FPGA_DNA**

```
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, &value, 0, type)) < 0) {
    /* error handling */
}
```

## 14.24. Debug

### 14.24.1. FG_DEBUGSOURCE

This parameter is for internal testing. Please DON'T use this parameter.

**Table 14.24. 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**<br>**Maximum 10**<br>**Stepsize 1** |
| Default value | **0** |

**Example 14.24. 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 */
}
```

### 14.24.2. FG_DEBUGSOURCENAME

This parameter is for internal testing. Please DON'T use this parameter.

**Table 14.25. 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 14.25. 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 */
}
```

### 14.24.3. FG_DEBUGSAVECONFIG

This parameter is for internal testing. Please DON'T use this parameter.

**Table 14.26. 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 14.26. 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 */
}
```

### 14.24.4. FG_DEBUG_VERSION

This parameter is for internal testing. Please DON'T use this parameter.

**Table 14.27. 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 14.27. 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 */
}
```

### 14.24.5. Input

#### 14.24.5.1. FG_DEBUGINENABLE

This parameter is for internal testing. Please DON'T use this parameter.

**Table 14.28. 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<br>**FG_OFF** Off |
| Default value | **FG_OFF** |

**Example 14.28. 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 */
}
```

### 14.24.5.2. FG_DEBUGFILE

This parameter is for internal testing. Please DON'T use this parameter.

**Table 14.29. 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 14.29. 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 */
}
```

### 14.24.5.3. FG_DEBUGINSERT

This parameter is for internal testing. Please DON'T use this parameter.

**Table 14.30. 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 14.30. 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 */
}
```

### 14.24.5.4. FG_DEBUGWRITEPIXEL

This parameter is for internal testing. Please DON'T use this parameter.

**Table 14.31. 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<br>**Maximum** 144115188075855872<br>**Stepsize** 1 |
| Default value | **0** |

**Example 14.31. 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 */
}
```

### 14.24.5.5. FG_DEBUGWRITEFLAG

This parameter is for internal testing. Please DON'T use this parameter.

**Table 14.32. 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<br>**FG_ENDOFFRAME** EndOfFrame |
| Default value | **FG_ENDOFLINE** |

**Example 14.32. 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 */
}
```

### 14.24.5.6. FG_DEBUGREADY

This parameter is for internal testing. Please DON'T use this parameter.

**Table 14.33. 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<br>**FG_NO** No |

**Example 14.33. 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 */
}
```

## 14.24.6. Output

### 14.24.6.1. FG_DEBUGOUTENABLE

This parameter is for internal testing. Please DON'T use this parameter.

**Table 14.34. 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<br>**FG_OFF** Off |
| Default value | **FG_OFF** |

**Example 14.34. 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 */
}
```

### 14.24.6.2. FG_DEBUGOUTXPOS

This parameter is for internal testing. Please DON'T use this parameter.

**Table 14.35. 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<br>**Maximum** 16384<br>**Stepsize** 1 |
| Unit of measure | **pixel** |

**Example 14.35. 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 */
}
```

### 14.24.6.3. FG_DEBUGOUTYPOS

This parameter is for internal testing. Please DON'T use this parameter.

**Table 14.36. 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**<br>**Maximum 65535**<br>**Stepsize 1** |
| Unit of measure | **pixel** |

**Example 14.36. 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 */
}
```

### 14.24.6.4. FG_DEBUGOUTPIXEL

This parameter is for internal testing. Please DON'T use this parameter.

**Table 14.37. 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**<br>**Maximum -1**<br>**Stepsize 1** |
| Unit of measure | **pixel** |

**Example 14.37. 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 */
}
```

# Glossary

| | |
|---|---|
| Area of Interest (AOI) | See Region of Interest. |
| Board | A Silicon Software hardware. Usually, a board is represented by a frame grabber. Boards might comprise multiple devices. |
| Board ID Number | An identification number of a Silicon Software 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 frame grabber. The first camera will have index zero. Mind the difference between the camera index and the frame grabber camera port.<br>See also Camera Port. |
| Camera Port | The Silicon Software frame grabber 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).<br><br>Silicon Software uses DMAs for data transfer such as image data between a board e.g. a frame grabber and a PC. Data transfers can be established in multiple directions i.e. from a frame grabber to the PC (download) and from the PC to a frame grabber (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.<br>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 frame grabber internal functionality.<br><br>Silicon Software uses hardware interrupts for the event transfer and processing is absolutely optimized for low latency. These interrupts are only produced by the frame grabber 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.<br><br>For example these events can be used to check the reliability between a frame trigger input and the resulting and expected camera frame.<br><br>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.<br>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.<br>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 frame grabber 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.<br><br>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.<br><br>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