# CXP-12 Interface Card 4C

AcquisitionApplets User Documentation for

## Acq_DualCXP12Area

Functional Description
For Framegrabber SDK Usage

Document Number: AW001883
Part Number: 000 (English)
Document Version: 01
Release Date: 5 September 2024
Applet Version 2.4.7.0

# 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


**All material in this publication is subject to change without notice and is copyright Basler AG.**

# Table of Contents


1. Introduction ... 1
    1.1. Features of Applet Acq_DualCXP12Area ... 1
        1.1.1. Parameterization Order ... 2
    1.2. Bandwidth ... 3
    1.3. Requirements ... 3
        1.3.1. Software Requirements ... 3
        1.3.2. Hardware Requirements ... 4
        1.3.3. License ... 4
    1.4. Camera Interface ... 4
    1.5. Frame ID ... 4
    1.6. Image Transfer to PC Memory ... 4
    1.7. DMA Image Tag ... 5
2. Software Interface ... 6
3. CoaXPress ... 7
    3.1. FG_SYSTEMMONITOR_POWER_OVER_CXP_STATE ... 7
    3.2. FG_SYSTEMMONITOR_POWER_OVER_CXP_CONTROLLER_ENABLED ... 7
    3.3. FG_SYSTEMMONITOR_PORT_BIT_RATE ... 8
    3.4. FG_SYSTEMMONITOR_STREAM_PACKET_SIZE ... 8
    3.5. FG_SYSTEMMONITOR_CXP_STANDARD ... 9
    3.6. FG_CXP_STREAM_PACKET_COUNT ... 10
    3.7. FG_PIXELFORMAT ... 10
    3.8. FG_SYSTEMMONITOR_USED_CXP_CONNECTIONS ... 11
    3.9. FG_SYSTEMMONITOR_CXP_IMAGE_LINE_MODE ... 12
    3.10. FG_TAPGEOMETRY ... 12
    3.11. Test ... 13
        3.11.1. FG_CXP_TRANSMITTED_PACKET_COUNT ... 14
        3.11.2. FG_CXP_RECEIVED_PACKET_COUNT ... 14
        3.11.3. FG_CXP_CORRUPTED_WORD_COUNT ... 14
        3.11.4. FG_CXP_PACKET_LENGTH_ERROR_COUNT ... 15
        3.11.5. FG_CXP_CLEAR_TEST_STATISTIC_PORT ... 15
4. Camera ... 17
    4.1. Events ... 17
        4.1.1. FG_CAMERA_STREAM_STATUS0 ... 17
        4.1.2. FG_START_OF_FRAME_CAM_PORT_0 ... 19
        4.1.3. FG_END_OF_FRAME_CAM_PORT_0 ... 19
5. Sensor Geometry ... 20
    5.1. FG_VANTAGEPOINT ... 20
    5.2. FG_SENSORWIDTH ... 20
    5.3. FG_SENSORHEIGHT ... 21
6. ROI ... 23
    6.1. FG_WIDTH ... 24
    6.2. FG_HEIGHT ... 25
    6.3. FG_XOFFSET ... 25
    6.4. FG_YOFFSET ... 26
7. Trigger ... 27
    7.1. Features and Functional Blocks of Area Trigger ... 27
    7.2. Digital Input/Output Mapping ... 31
    7.3. Event Overview ... 31
    7.4. Trigger Scenarios ... 32
        7.4.1. Internal Frequency Generator / interface card Controlled ... 32
        7.4.2. External Trigger Signals / IO Triggered ... 34
        7.4.3. Control of Two Flash Lights ... 36
        7.4.4. Software Trigger ... 39
        7.4.5. Software Trigger with Trigger Queue ... 41
        7.4.6. External Trigger with Trigger Queue ... 43
        7.4.7. Bypass External Trigger Signals ... 44

7.4.8. Multi Camera Applications / Synchronized Cameras ... 44
7.4.9. Hardware System Analysis and Error Detection / Trigger Debugging ... 45
7.5. Parameters ... 46
7.5.1. FG_AREATRIGGERMODE ... 46
7.5.2. FG_TRIGGERSTATE ... 47
7.5.3. FG_TRIGGER_FRAMESPERSECOND ... 48
7.5.4. Trigger Input ... 49
7.5.4.1. External ... 49
7.5.4.1.1. FG_TRIGGERIN_DEBOUNCE ... 49
7.5.4.1.2. FG_FRONT_GPI ... 50
7.5.4.1.3. FG_TRIGGERIN_SRC ... 51
7.5.4.1.4. FG_TRIGGERIN_POLARITY ... 51
7.5.4.1.5. FG_TRIGGERIN_DOWNSCALE ... 52
7.5.4.1.6. FG_TRIGGERIN_DOWNSCALE_PHASE ... 52
7.5.4.2. Software Trigger ... 53
7.5.4.2.1. FG_SENDSOFTWARETRIGGER ... 53
7.5.4.2.2. FG_SOFTWARETRIGGER_IS_BUSY ... 54
7.5.4.2.3. FG_SOFTWARETRIGGER_QUEUE_FILLLEVEL ... 54
7.5.4.3. Statistics ... 55
7.5.4.3.1. FG_TRIGGERIN_STATS_SOURCE ... 55
7.5.4.3.2. FG_TRIGGERIN_STATS_POLARITY ... 56
7.5.4.3.3. FG_TRIGGERIN_STATS_PULSECOUNT ... 56
7.5.4.3.4. FG_TRIGGERIN_STATS_PULSECOUNT_CLEAR ... 57
7.5.4.3.5. FG_TRIGGERIN_STATS_FREQUENCY ... 57
7.5.4.3.6. FG_TRIGGERIN_STATS_MINFREQUENCY ... 58
7.5.4.3.7. FG_TRIGGERIN_STATS_MAXFREQUENCY ... 58
7.5.4.3.8. FG_TRIGGERIN_STATS_MINMAXFREQUENCY_CLEAR ... 59
7.5.4.3.9. FG_TRIGGER_FRONT_GPI0_RISING ... 59
7.5.4.3.10. FG_TRIGGER_FRONT_GPI0_FALLING ... 60
7.5.5. Sequencer ... 60
7.5.5.1. FG_TRIGGER_MULTIPLY_PULSES ... 60
7.5.6. Queue ... 61
7.5.6.1. FG_TRIGGERQUEUE_MODE ... 61
7.5.6.2. FG_TRIGGERQUEUE_FILLLEVEL ... 62
7.5.6.3. FG_TRIGGER_QUEUE_FILLLEVEL_EVENT_ON_THRESHOLD ... 62
7.5.6.4. FG_TRIGGER_QUEUE_FILLLEVEL_EVENT_OFF_THRESHOLD ... 63
7.5.6.5. FG_TRIGGER_QUEUE_FILLLEVEL_THRESHOLD_CAM0_ON ... 63
7.5.6.6. FG_TRIGGER_QUEUE_FILLLEVEL_THRESHOLD_CAM0_OFF ... 63
7.5.7. Pulse Form Generator 0 ... 63
7.5.7.1. FG_TRIGGER_PULSEFORMGEN0_DOWNSCALE et al. ... 64
7.5.7.2. FG_TRIGGER_PULSEFORMGEN0_DOWNSCALE_PHASE et al. ... 65
7.5.7.3. FG_TRIGGER_PULSEFORMGEN0_DELAY et al. ... 66
7.5.7.4. FG_TRIGGER_PULSEFORMGEN0_WIDTH et al. ... 67
7.5.8. Pulse Form Generator 1 ... 67
7.5.9. Pulse Form Generator 2 ... 67
7.5.10. Pulse Form Generator 3 ... 68
7.5.11. Camera Out Signal Mapping ... 68
7.5.11.1. FG_TRIGGERCAMERA_SOURCE_CXP0 ... 68
7.5.11.2. FG_TRIGGERCAMERA_SOURCE_CXP1 ... 70
7.5.11.3. FG_TRIGGERCAMERA_SOURCE_CXP2 ... 71
7.5.11.4. FG_TRIGGERCAMERA_SOURCE_CXP3 ... 72
7.5.12. Digital Output ... 73
7.5.12.1. FG_TRIGGEROUT_SELECT_FRONT_GPO_0 ... 73
7.5.12.2. FG_TRIGGEROUT_SELECT_FRONT_GPO_1 ... 75
7.5.12.3. Statistics ... 76
7.5.12.3.1. FG_TRIGGER_EXCEEDED_PERIOD_LIMITS ... 76
7.5.12.3.2. FG_TRIGGER_EXCEEDED_PERIOD_LIMITS_CLEAR ... 76
7.5.12.3.3. FG_TRIGGEROUT_STATS_SOURCE ... 77

7.5.12.3.4. FG_TRIGGEROUT_STATS_PULSECOUNT ..... 77
7.5.12.3.5. FG_TRIGGEROUT_STATS_PULSECOUNT_CLEAR ..... 78
7.5.12.3.6. FG_MISSING_CAMERA_FRAME_RESPONSE ..... 78
7.5.12.3.6.1. ..... 78
7.5.12.3.7. FG_MISSING_CAMERA_FRAME_RESPONSE_CLEAR ..... 79
7.5.12.3.8. FG_TRIGGER_EXCEEDED_PERIOD_LIMITS_CAM0 ..... 80
7.5.12.3.9. FG_MISSING_CAM0_FRAME_RESPONSE ..... 80
7.5.13. Output Event ..... 80
7.5.13.1. FG_TRIGGER_OUTPUT_EVENT_SELECT ..... 80
7.5.13.2. FG_TRIGGER_OUTPUT_CAM0 ..... 81
8. Overflow ..... 82
8.1. FG_FILLLEVEL ..... 82
8.2. FG_OVERFLOW ..... 83
8.3. FG_OVERFLOW_OFF_THRESHOLD ..... 83
8.4. FG_OVERFLOW_ON_THRESHOLD ..... 84
8.5. FG_OVERFLOW_ON_SYNC_THRESHOLD ..... 85
8.6. FG_OVERFLOW_EVENT_SELECT ..... 85
8.7. Events ..... 86
8.7.1. FG_OVERFLOW_CAM0 ..... 87
9. Image Selector ..... 88
9.1. FG_IMG_SELECT_PERIOD ..... 88
9.2. FG_IMG_SELECT ..... 89
10. White Balance ..... 90
10.1. FG_SCALINGFACTOR_GREEN ..... 90
10.2. FG_SCALINGFACTOR_RED ..... 90
10.3. FG_SCALINGFACTOR_BLUE ..... 91
11. Color Converter ..... 92
12. Output Format ..... 93
12.1. FG_FORMAT ..... 93
12.2. FG_BITALIGNMENT ..... 97
12.3. FG_PIXELDEPTH ..... 97
12.4. FG_CUSTOM_BIT_SHIFT_RIGHT ..... 98
13. Camera Simulator ..... 100
13.1. FG_CAMERASIMULATOR_ENABLE ..... 100
13.2. FG_CAMERASIMULATOR_WIDTH ..... 101
13.3. FG_CAMERASIMULATOR_LINE_GAP ..... 102
13.4. FG_CAMERASIMULATOR_HEIGHT ..... 102
13.5. FG_CAMERASIMULATOR_FRAME_GAP ..... 103
13.6. FG_CAMERASIMULATOR_PATTERN ..... 104
13.7. FG_CAMERASIMULATOR_PATTERN_OFFSET ..... 104
13.8. FG_CAMERASIMULATOR_ROLL ..... 105
13.9. FG_CAMERASIMULATOR_SELECT_MODE ..... 106
13.10. FG_CAMERASIMULATOR_PIXEL_FREQUENCY ..... 106
13.11. FG_CAMERASIMULATOR_LINERATE ..... 107
13.12. FG_CAMERASIMULATOR_FRAMERATE ..... 107
13.13. FG_CAMERASIMULATOR_TRIGGER_MODE ..... 108
13.14. FG_CAMERASIMULATOR_ACTIVE ..... 109
13.15. FG_CAMERASIMULATOR_PASSIVE ..... 109
14. Miscellaneous ..... 111
14.1. FG_TIMEOUT ..... 111
14.2. FG_APPLET_ID ..... 111
14.3. FG_APPLET_BUILD_TIME ..... 112
14.4. FG_HAP_FILE ..... 112
14.5. FG_CAMSTATUS ..... 112
14.6. FG_CAMSTATUS_EXTENDED ..... 113
14.7. FG_SYSTEMMONITOR_FPGA_DNA_LOW ..... 114
14.8. FG_SYSTEMMONITOR_FPGA_DNA_HIGH ..... 114
14.9. Version ..... 115

14.9.1. FG_APPLET_VERSION ... 115
14.9.2. FG_APPLET_REVISION ... 115
14.9.3. FG_VISUALAPPLETS_BUILD_VERSION ... 116
14.10. Legacy ... 116
14.10.1. FG_CXP_TRIGGER_PACKET_MODE ... 116
14.10.2. FG_TRIGGERCAMERA_OUT_SELECT ... 117
14.11. Debug ... 119
14.11.1. FG_DEBUGSOURCE ... 119
14.11.2. FG_DEBUGSOURCENAME ... 119
14.11.3. FG_DEBUGSAVECONFIG ... 119
14.11.4. FG_DEBUG_SLOWMODE ... 120
14.11.5. FG_DEBUG_SOFTWRAE_SLOWGATE ... 120
14.11.6. FG_DEBUG_PWM_SLOWRATE ... 121
14.11.7. FG_DEBUG_VERSION ... 122
14.11.8. FG_DEBUG_FRAMEID_TO_FIRSTPIXEL ... 122
14.11.9. Input ... 122
14.11.9.1. FG_DEBUGINENABLE ... 123
14.11.9.2. FG_DEBUGFILE ... 123
14.11.9.3. FG_DEBUGINSERT ... 123
14.11.9.4. FG_DEBUGWRITEPIXEL ... 124
14.11.9.5. FG_DEBUGWRITEFLAG ... 125
14.11.9.6. FG_DEBUGREADY ... 125
14.11.9.7. FG_DEBUG_FORCE_FRAMEID ... 125
14.11.9.8. FG_DEBUG_FRAMEID ... 126
14.11.10. Output ... 127
14.11.10.1. FG_DEBUGOUTENABLE ... 127
14.11.10.2. FG_DEBUGOUTXPOS ... 127
14.11.10.3. FG_DEBUGOUTYPOS ... 128
14.11.10.4. FG_DEBUGOUTPIXEL ... 128
14.12. GenTL ... 129
14.12.1. FG_GENTL_INFO_VERSION ... 129
14.12.2. FG_GENTL_INFO_IGNOREFGFORMAT ... 129
14.12.3. FG_GENTL_INFO_OVERFLOWCAPABLE ... 129
15. Boardstatus ... 131
15.1. FG_SYSTEMMONITOR_CHANNEL_CURRENT ... 131
15.2. FG_SYSTEMMONITOR_CHANNEL_VOLTAGE ... 131
15.3. FG_SYSTEMMONITOR_MAPPED_TO_FG_PORT ... 132
15.4. FG_DMASTATUS ... 132
15.5. FG_SYSTEMMONITOR_FPGA_TEMPERATURE ... 133
15.6. FG_SYSTEMMONITOR_FPGA_VCC_INT ... 133
15.7. FG_SYSTEMMONITOR_FPGA_VCC_AUX ... 134
15.8. FG_SYSTEMMONITOR_FPGA_VCC_BRAM ... 134
15.9. FG_SYSTEMMONITOR_CURRENT_LINK_WIDTH ... 135
15.10. FG_SYSTEMMONITOR_CURRENT_LINK_SPEED ... 135
15.11. FG_SYSTEMMONITOR_PCIE_TRAINED_PAYLOAD_SIZE ... 136
15.12. FG_SYSTEMMONITOR_PCIE_TRAINED_REQUEST_SIZE ... 136
15.13. FG_SYSTEMMONITOR_EXTERNAL_POWER ... 137
15.14. FG_CXP_INPUT_MAPPED_FW_PORT_PORT ... 137
16. Errors ... 138
16.1. FG_SYSTEMMONITOR_DECODER_8B_10B_ERROR ... 138
16.2. FG_SYSTEMMONITOR_BYTE_ALIGNMENT_8B_10B_LOCKED ... 138
16.3. FG_SYSTEMMONITOR_RX_STREAM_INCOMPLETE_COUNT ... 139
16.4. FG_SYSTEMMONITOR_RX_UNKNOWN_DATA_RECEIVED_COUNT ... 139
16.5. FG_CXP_OVERTRIGGER_REQUEST_PULSECOUNT ... 140
16.6. FG_CXP_TRIGGER_ACK_MISSING_COUNT ... 140
16.7. FG_CXP_CONTROL_ACK_LOST_COUNT ... 141
16.8. FG_CXP_CONTROL_TAG_ERROR_COUNT ... 141
16.9. FG_CXP_CONTROL_ACK_INCOMPLETE_COUNT ... 142

16.10. FG_CXP_HEARTBEAT_INCOMPLETE_COUNT ..... 142
16.11. FG_CXP_HEARTBEAT_MAX_PERIOD_VIOLATION_COUNT ..... 143
16.12. FG_PACKET_TAG_ERROR_COUNT ..... 143
16.13. FG_SYSTEMMONITOR_PACKETBUFFER_OVERFLOW_COUNT ..... 144
16.14. FG_SYSTEMMONITOR_PACKETBUFFER_OVERFLOW_SOURCE ..... 144
16.15. FG_CXP_IMAGETAG_ERROR_COUNT ..... 145
16.16. FG_CXP_STREAMID_ERROR_COUNT ..... 145
16.17. FG_CXP_CAMERA_MARKER_ERROR_COUNT ..... 146
16.18. FG_CXP_CAMERA_UNEXPECTED_STARTUP_DATA ..... 146
16.19. FG_CXP_CAMERA_FRAME_LOST_COUNT ..... 147
16.20. FG_CXP_CAMERA_FRAME_CORRUPT_COUNT ..... 147
16.21. CRC ..... 148
16.21.1. FG_SYSTEMMONITOR_RX_PACKET_CRC_ERROR_COUNT ..... 148
16.21.2. FG_CXP_STREAMPACKET_CRC_ERROR ..... 148
16.21.3. FG_CXP_CONTROL_ACK_PACKET_CRC_ERROR ..... 149
16.22. LengthErrors ..... 149
16.22.1. FG_SYSTEMMONITOR_RX_LENGTH_ERROR_COUNT ..... 149
16.22.2. FG_CXP_STREAMPACKET_LENGTH_ERROR ..... 150
16.23. ReceivedPacketsCorrected ..... 150
16.23.1. FG_CXP_ERROR_CORRECTED ..... 150
16.23.2. FG_CXP_ERROR_CORRECTED_TRIGGER ..... 151
16.23.3. FG_CXP_ERROR_CORRECTED_TRIGGER_ACK ..... 151
16.23.4. FG_CXP_ERROR_CORRECTED_STREAM ..... 152
16.23.5. FG_CXP_ERROR_CORRECTED_CONTROL_ACK ..... 152
16.23.6. FG_CXP_ERROR_CORRECTED_LINKTEST ..... 153
16.23.7. FG_CXP_ERROR_CORRECTED_HEARTBEAT ..... 153
16.23.8. FG_CORRECTED_ERROR_COUNT ..... 154
16.24. ReceivedPacketsUncorrected ..... 154
16.24.1. FG_CXP_ERROR_UNCORRECTED ..... 154
16.24.2. FG_CXP_ERROR_UNCORRECTED_TRIGGER ..... 155
16.24.3. FG_CXP_ERROR_UNCORRECTED_TRIGGER_ACK ..... 155
16.24.4. FG_CXP_ERROR_UNCORRECTED_STREAM ..... 156
16.24.5. FG_CXP_ERROR_UNCORRECTED_CONTROL_ACK ..... 156
16.24.6. FG_CXP_ERROR_UNCORRECTED_LINKTEST ..... 157
16.24.7. FG_CXP_ERROR_UNCORRECTED_HEARTBEAT ..... 157
16.24.8. FG_UNCORRECTED_ERROR_COUNT ..... 158
16.25. UnsupportedPackets ..... 158
16.25.1. FG_SYSTEMMONITOR_RX_UNSUPPORTED_PACKET_COUNT ..... 158
16.25.2. FG_CXP_UNSUPPORTED_GPIO_RECEIVED ..... 159
16.25.3. FG_CXP_UNSUPPORTED_EVENT_RECEIVED ..... 159
16.25.4. FG_CXP_UNSUPPORTED_HEARTBEAT_RECEIVED ..... 159
16.25.5. FG_CXP_UNSUPPORTED_GPIO_ACK_RECEIVED ..... 160
16.25.6. FG_CXP_UNSUPPORTED_GPIO_REQUEST_RECEIVED ..... 160
17. Revision History ..... 162
17.1. Fixed Issues ..... 162
17.1.1. Fixed in Version 2.4.7.0 ..... 162
17.1.2. Fixed in Version 2.2.7.0 ..... 163
17.1.3. Fixed in Version 2.0.5.0 ..... 163
17.2. Known Issues ..... 163
Glossary ..... 164
Index ..... 167

# Chapter 1. Introduction

This document provides you with detailed information on applet "Acq_DualCXP12Area" for CXP-12 Interface Card 4C .

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_DualCXP12Area

"Acq_DualCXP12Area" is a dual-camera applet. Up to two individual cameras can be used in parallel. All features of this applet are available for both camera ports. You can configure the CoaXPress camera interface for CoaXPress cameras version 1.1.1 and 2.0, transferring grayscale (monochrome), Bayer pattern , or color pixels. Allowed pixel formats are Gray (Mono8, Mono10, Mono12, Mono14, Mono16), Bayer (BayerGR8, BayerGR10, BayerGR12, BayerGR14, BayerRG8, BayerRG10, BayerRG12, BayerRG14, BayerGB8, BayerGB10, BayerGB12, BayerGB14, BayerBG8, BayerBG10, BayerBG12, BayerBG14), Color (RGB8, RGB10, RGB12, RGB14, RGB16), and YCbCr422_8. You can use cameras with CoaXPress link aggregation of 2 and 1 with this applet. The maximum link speed is CXP-12. A multi-functional area trigger is included in the applet. This allows you to control the camera or external devices using interface card generated, external, or software generated trigger pulses. Area scan cameras transferring images with a resolution of up to 32768 by 65536 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 interface card and PCs. The applet supports tap geometry sorting 1X-1Y as well as 1X-2YE. For reverse operation, you can mirror the image in x-direction and y-direction before cutting the ROI. Acquired images are buffered in interface card memory. You can select a region of interest (ROI) for further processing. The stepsize of the ROI width is 16 pixel. The ROI stepsize for the image height is 1 line. The high quality Bayer pattern de-mosaicing is based on a 5x5 kernel size. A color converter automatically converts the input pixel formats to the output formats. In this applet conversions from monochrome, RGB or Bayer to monochrome and RGB can be performed.

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

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

**Table 1.1. Feature Summary of Acq_DualCXP12Area**

| Feature | Applet Property |
|---|---|
| Applet Name | <br>Acq_DualCXP12Area |
| Type of Applet | AcquisitionApplets |
| Board | CXP-12 Interface Card 4C |
| No. of Cameras | 2 , asynchronous or synchronous |
| Camera Type | CoaXPress, link aggregation max. 2, maximum speed CXP-12, Version 1.1.1 and 2.0 |
| Sensor Type | Area Scan |
| Camera Format | Monochrome, Bayer pattern or RGB |
| Pixel Format | Gray (Mono8, Mono10, Mono12, Mono14, Mono16), Bayer (BayerGR8, BayerGR10, BayerGR12, BayerGR14, BayerRG8, BayerRG10, BayerRG12, BayerRG14, BayerGB8, BayerGB10, BayerGB12, BayerGB14, BayerBG8, BayerBG10, BayerBG12, BayerBG14), Color (RGB8, RGB10, RGB12, RGB14, RGB16), and YCbCr422_8. |
| Processing Bit Depth | 16 Bit per color component |
| Maximum Images Dimensions | 32768 * 65536 |
| ROI Stepsize | x: 16, y: 1 |
| Tap Geometry Sorting | 1X-1Y, 1X-2YE |
| Mirroring | Yes, horizontal and vertical (set the parameter *FG_VANTAGEPOINT*) |
| Image Selector | Yes |
| Noise Filter | No |
| Shading Correction | No |
| Dead Pixel Interpolation | No |
| Bayer Filter | Yes, High Quality Extended (HQe) |
| Color White Balancing | Yes |
| Color Converter | yes, Mono, RGB or Bayer to Mono or RGB |
| Lookup Table | No |
| DMA | Full Speed |
| DMA Image Output Format | All grayscale and color formats. See description above. |
| Event Generation | yes |
| Overflow Control | yes |

### 1.1.1. Parameterization Order

We recommend to configure the functional blocks which are responsible for sensor setup/correction first. This will be the camera settings, shading correction, and dead pixel interpolation (if available). Afterwards, you can

configure other image enhancement functional blocks such as white balancing, noise filter, and lookup table. By default, all presets are configured for receiving images directly.

## 1.2. Bandwidth

The maximum bandwidths of applet Acq_DualCXP12Area are listed in the following table.

**Table 1.2. Bandwidth of Acq_DualCXP12Area**

| Description | Bandwidth |
|---|---|
| Max. CXP Speed | CXP-12 |
| Peak Bandwidth per Camera | 2400 MPixel/s |
| Mean Bandwidth per Camera | 2400 MPixel/s |
| DMA Bandwidth | 7200 MByte/s (depends on PC mainboard) |

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

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

The required output bandwidth of an applet can differ from the input bandwidth. A region of interest (ROI) and the output format can change the required output bandwidth and the maximum mean bandwidth. Moreover, this applet is a Bayer applet. The required output bandwidth will be three times higher than the input bandwidth. (This applies only when debayering is switched to ON.) Mind that the DMA bandwidth is the total bandwidth. The sum of all camera channel bandwidths has to be less than the maximum DMA bandwidth to avoid overflows.

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 7200 MByte/s for the possible DMA bandwidth might be lower due to the chipset and its configuration. Additionally, some PCIe slots do not support the required number of lanes to transfer the requested or expected bandwidth. In these cases, have a look at the mainboard specification. A behaviour like multiplexing between several PCIe slots can be seen in rare cases. Some mainboard manufacturers provide a BIOS feature where you can select the PCIe payload size: Always try to set this to its maximum value or simply to automatic. This can help in specific cases.

## 1.3. Requirements

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

### 1.3.1. Software Requirements

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

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

### 1.3.2. Hardware Requirements

To run applet "Acq_DualCXP12Area", a Basler CXP-12 Interface Card 4C is required.

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

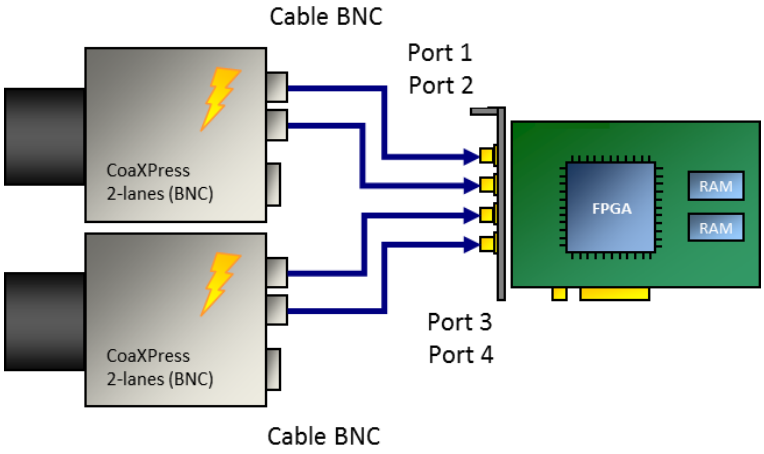
### 1.3.3. License

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

## 1.4. Camera Interface

Applet "Acq_DualCXP12Area" supports 2 CXP cameras. The interface card has 4 connectors. Use two or one CoaXPress cables to connect each of the cameras with the interface card. The maximum link aggregation of this applet is two. The mapping of the ports between the camera and the interface card is not important. You can chose any order.

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




## 1.5. Frame ID

For CoaXPress cameras, each frame includes a source tag also called frame ID. This applet will output each frame to the host PC attached with this frame ID. Moreover, overflow events will also include this frame ID. By this, the exact mapping of a given frame in the host PC to a camera frame is possible.

Check chapter Chapter 8, '*Overflow*' for more information about overflow conditions and the overflow event data structure including the frame ID.

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

## 1.6. Image Transfer to PC Memory

The image transfer between interface card and PC is performed via DMA transfers. In this applet, 2 DMA channels exist for transferring image data. One channel for each camera. The DMA channels have the same indices as the cameras, starting with 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.

## 1.7. DMA Image Tag

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

**Table 1.3. Structure of FG_IMAGE_TAG**

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

You may check for lost or corrupted frames using the overflow module described in Chapter 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 dual camera applet and is using only one DMA channel for each camera. All parameterizations for the first camera are made using index 0 and all parameterizations for the second camera are made using index 1.

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

# Chapter 3. CoaXPress

This applet can be used with up to 2 area scan cameras. To receive correct image data from your camera, it is crucial that the camera output format matches the selected interface card input format. The following parameters configure the interface card's camera interface to match with the individual camera pixel format. Most cameras support different operation modes. Consult the manual of your camera to obtain the necessary information how to configure the camera to the desired pixel format.

Ensure that the images transferred by the camera do not exceed the maximum allowed image dimensions for this applet (32768 x 65536).

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

This applet allows the usage of 1x-2ye and 1x-1y sorting of the image data according to the CoaXPress standard.

## 3.1. FG_SYSTEMMONITOR_POWER_OVER_CXP_STATE

Returns the power over CXP (PoCXP) state. Range: {BOOTING, POCXPOK, MAX_CURR, LOW_VOLT, OVER_VOLT, ADC_Chip_Error}. The first 5 states are defined by the CXP standard for the PoCXP state machine. The last state ADC_Chip_Error represents an error when the communication between the FPGA and the ADC chip is disrupted. The communication between the FPGA and ADC chip measures the voltage and current of the channel.

**Table 3.1. Parameter properties of FG_SYSTEMMONITOR_POWER_OVER_CXP_STATE**

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_POWER_OVER_CXP_STATE** |
| Display Name | **Systemmonitor Power Over CXP State** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 3.1. Usage of FG_SYSTEMMONITOR_POWER_OVER_CXP_STATE**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

## 3.2. FG_SYSTEMMONITOR_POWER_OVER_CXP_CONTROLLER_ENABLED

Indicates whether the power over CXP (PoCXP) controller is enabled. Range: {NO, YES}. YES: The camera is powered via the CXP cable when connected. NO: The camera is not powered via the CXP cable. This parameter doesn't indicate whether the camera is sourced or not, instead it indicates whether powering the camera via the CXP cable is enabled or not.

Table 3.2. Parameter properties of FG_SYSTEMMONITOR_POWER_OVER_CXP_CONTROLLER_ENABLED

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_POWER_OVER_CXP_CONTROLLER_ENABLED** |
| Display Name | **Systemmonitor Power Over CXP Controller Enabled** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

Example 3.2. Usage of FG_SYSTEMMONITOR_POWER_OVER_CXP_CONTROLLER_ENABLED

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

## 3.3. FG_SYSTEMMONITOR_PORT_BIT_RATE

Returns the port bit rate of the CXP channel.

Table 3.3. Parameter properties of FG_SYSTEMMONITOR_PORT_BIT_RATE

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_PORT_BIT_RATE** |
| Display Name | **Systemmonitor Port Bit Rate** |
| Type | **Double Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Unit of measure | **Gb/s** |

Example 3.3. Usage of FG_SYSTEMMONITOR_PORT_BIT_RATE

```
int result = 0;

FieldParameterDouble access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMDOUBLE;

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

## 3.4. FG_SYSTEMMONITOR_STREAM_PACKET_SIZE

Returns the stream packet size in bytes. Range: between 4 and 65535 bytes in steps of 4 bytes.

Table 3.4. Parameter properties of FG_SYSTEMMONITOR_STREAM_PACKET_SIZE

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_STREAM_PACKET_SIZE** |
| Display Name | **Systemmonitor Stream Packet Size** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

Example 3.4. Usage of FG_SYSTEMMONITOR_STREAM_PACKET_SIZE

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

## 3.5. FG_SYSTEMMONITOR_CXP_STANDARD

Returns the version of the used CXP standard.

Table 3.5. CXP Standard Version

| CXP Standard Version | | |
|---|---|---|
| CXP_1_0 | | |
| CXP_1_1_1 | | |
| CXP_2_0 | | |
| Unknown | | |

Table 3.6. Parameter properties of FG_SYSTEMMONITOR_CXP_STANDARD

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_CXP_STANDARD** |
| Display Name | **Systemmonitor CXP Standard** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

Example 3.5. Usage of FG_SYSTEMMONITOR_CXP_STANDARD

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

## 3.6. FG_CXP_STREAM_PACKET_COUNT

This parameter counts the amount of received stream packets. Bits [29:0] count the number of packets. Bit [30] is set when a counter overflow occurs. Range: 0 to 4294967295 (32 bit).

**Table 3.7. Parameter properties of FG_CXP_STREAM_PACKET_COUNT**

| Property | Value |
|---|---|
| Name | **FG_CXP_STREAM_PACKET_COUNT** |
| Display Name | **CXP Stream Packet Count** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 3.6. Usage of FG_CXP_STREAM_PACKET_COUNT**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

## 3.7. FG_PIXELFORMAT

This parameter specifies the data format of the connected camera.

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

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

This applet performs a Bayer de-mosaicing. The Bayer pattern is derived from the pixel format.

**Table 3.8. Parameter properties of FG_PIXELFORMAT**

| Property | Value |
|---|---|
| Name | **FG_PIXELFORMAT** |
| Display Name | **Pixel Format** |
| Type | **Enumeration** |
| Access policy | **Read/Write** |
| Storage policy | **Persistent** |
| Allowed values | **BayerGR8** Bayer GR 8<br>**BayerGR10** Bayer GR 10p<br>**BayerGR12** Bayer GR 12p<br>**BayerGR14** Bayer GR 14p<br>**BayerRG8** Bayer RG 8<br>**BayerRG10** Bayer RG 10p<br>**BayerRG12** Bayer RG 12p<br>**BayerRG14** Bayer RG 14p<br>**BayerGB8** Bayer GB 8<br>**BayerGB10** Bayer GB 10p<br>**BayerGB12** Bayer GB 12p<br>**BayerGB14** Bayer GB 14p<br>**BayerBG8** Bayer BG 8<br>**BayerBG10** Bayer BG 10p<br>**BayerBG12** Bayer BG 12p<br>**BayerBG14** Bayer BG 14p<br>**Mono8** Mono 8<br>**Mono10** Mono 10p<br>**Mono12** Mono 12p<br>**Mono14** Mono 14p<br>**Mono16** Mono 16p<br>**RGB8** RGB 8<br>**RGB10** RGB 10p<br>**RGB12** RGB 12p<br>**RGB14** RGB 14p<br>**RGB16** RGB 16<br>**YUV422_8** YCbCr422_8 |
| Default value | **Mono8** |

**Example 3.7. Usage of FG_PIXELFORMAT**

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

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

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

## 3.8. FG_SYSTEMMONITOR_USED_CXP_CONNECTIONS

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

Table 3.9. Parameter properties of FG_SYSTEMMONITOR_USED_CXP_CONNECTIONS

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

Example 3.8. Usage of FG_SYSTEMMONITOR_USED_CXP_CONNECTIONS

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

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

## 3.9. FG_SYSTEMMONITOR_CXP_IMAGE_LINE_MODE

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

Table 3.10. Parameter properties of FG_SYSTEMMONITOR_CXP_IMAGE_LINE_MODE

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

Example 3.9. Usage of FG_SYSTEMMONITOR_CXP_IMAGE_LINE_MODE

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

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

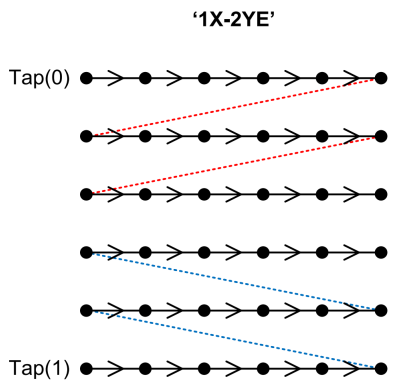
## 3.10. FG_TAPGEOMETRY

Defines the tap geometry sorting of the applet.

The applet supports **FG_GEOMETRY_1X_1Y** or **FG_GEOMETRY_1X_2YE**. With **FG_GEOMETRY_1X_1Y** mode you define that the whole image is transferred as one data stream. This mode is the default mode for most cameras.

As sensors grow bigger, a faster readout is needed. To achieve this, use the **FG_GEOMETRY_1X_2YE** mode. In **FG_GEOMETRY_1X_2YE** mode, the sensor is read out in two subframes at once. Both subframes are transferred simultaneously using two streams.

The following image shows how the frames are composed:




If tap sorting is used, the applet combines all subframes into one frame. In order to get correct frames, the tap sorting of the applet needs to match the camera tap sorting.

If *FG_TAPGEOMETRY* is set to **FG_GEOMETRY_1X_2YE**, the parameter *FG_SENSORHEIGHT* needs to be set to the camera ROI size and must be an even integer. The parameter dependency will then be: *FG_YOFFSET* + *FG_HEIGHT* <= *FG_SENSORHEIGHT*.

**Table 3.11. 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_2YE** 1X-2YE |
| Default value | **FG_GEOMETRY_1X_1Y** |

**Example 3.10. 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 */
}
```

## 3.11. Test

This category gives information about CXP link test statistics. It shows counters for received and transmitted CXP packets as well as statistics on corrupted words and length errors. Additionally, it enables the user to reset the statistics counter for each channel separately.

### 3.11.1. FG_CXP_TRANSMITTED_PACKET_COUNT

This parameter counts the amount of link test packets that were transmitted. This register is useful to see how many packets were sent since the start of the test. It is cleared with the clear bit from the test statistics clear register.

**Table 3.12. Parameter properties of FG_CXP_TRANSMITTED_PACKET_COUNT**

| Property | Value |
|---|---|
| Name | **FG_CXP_TRANSMITTED_PACKET_COUNT** |
| Display Name | **CXP Transmitted Packets Count** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Persistent** |

**Example 3.11. Usage of FG_CXP_TRANSMITTED_PACKET_COUNT**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

### 3.11.2. FG_CXP_RECEIVED_PACKET_COUNT

This parameter counts the amount of received link test packets. It is cleared with the clear bit from the test statistics clear register.

**Table 3.13. Parameter properties of FG_CXP_RECEIVED_PACKET_COUNT**

| Property | Value |
|---|---|
| Name | **FG_CXP_RECEIVED_PACKET_COUNT** |
| Display Name | **CXP Received Packet Count** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Persistent** |

**Example 3.12. Usage of FG_CXP_RECEIVED_PACKET_COUNT**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

### 3.11.3. FG_CXP_CORRUPTED_WORD_COUNT

This parameter counts the amount of measured packet word errors. A packet word is a 32-bit CXP native word which carries 4 test characters. It is cleared with the clear bit from the test statistics clear register.

**Table 3.14. Parameter properties of FG_CXP_CORRUPTED_WORD_COUNT**

| Property | Value |
|---|---|
| Name | **FG_CXP_CORRUPTED_WORD_COUNT** |
| Display Name | **CXP Corrupted Word Count** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Persistent** |

**Example 3.13. Usage of FG_CXP_CORRUPTED_WORD_COUNT**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

## 3.11.4. FG_CXP_PACKET_LENGTH_ERROR_COUNT

This parameter counts the amount of packets which didn't provide 1024 test words. CXP standard defines a test packet to contain 4096 test characters, i.e. 1024 x 32 bit words. This packet is repeated infinitely until the test is terminated. The count range is [0; 128]. The maximal value 128 means that there were at least 128 or more packets which violated the length requirements as defined in CXP 2.0 standard chapter 9.9.2.

**Table 3.15. Parameter properties of FG_CXP_PACKET_LENGTH_ERROR_COUNT**

| Property | Value |
|---|---|
| Name | **FG_CXP_PACKET_LENGTH_ERROR_COUNT** |
| Display Name | **CXP Packet Length Error Count** |
| Type | **Unsigned Integer Field (64 Bit)** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Persistent** |

**Example 3.14. Usage of FG_CXP_PACKET_LENGTH_ERROR_COUNT**

```
int result = 0;

FieldParameterAccess access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMACCESS;

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

## 3.11.5. FG_CXP_CLEAR_TEST_STATISTIC_PORT

This parameter clears all test counters to zero for the channel selected by the parameter field index.

**Table 3.16. Parameter properties of FG_CXP_CLEAR_TEST_STATISTIC_PORT**

| Property | Value |
|---|---|
| Name | **FG_CXP_CLEAR_TEST_STATISTIC_PORT** |
| Display Name | **CXP Clear Test Statistic Port** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read/Write** |
| Storage policy | **Persistent** |
| Default value | **0** |

**Example 3.15. Usage of FG_CXP_CLEAR_TEST_STATISTIC_PORT**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

for (unsigned int i = 0; i < 4; ++i)
{
    access.index = i;
    access.value = 0;

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

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

# Chapter 4. Camera

This applet Acq_DualCXP12Area for the CXP-12 Interface Card 4C 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.

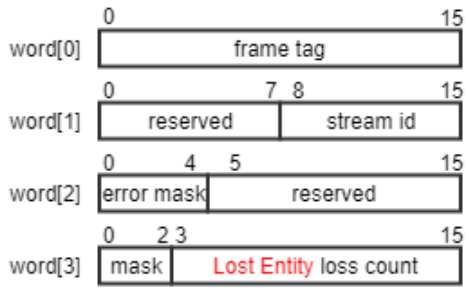
## 4.1. Events

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_CAMERA_STREAM_STATUS0

When the operator detects that the received reconstructed frame is larger or smaller than what was promoted by the camera in the CXP image header, a safety circuit gets activated. The operator then cuts off exceeding pixels and lines, so that the subsequent processing pipeline always sees the frame size which was defined in the image header. If the received frame is smaller in its dimensions than what was specified in the image header, the operator fills up the received frame with undefined data to achieve the specified frame dimensions which were defined in the image header. Filling up a smaller frame can cause the follow-up frames to get lost. The loss is then reported per event to the runtime software (Framegrabber SDK)(see the following paragraph). The size mismatch causes an event, too.




The event payload is provided as four 16-bit data words. The event format is defined as follows:

- word [0]
  - bits [0:15]: CXP image tag in which the event occurred.
- word [1]
  - bits [8:15]: Stream ID in which the event occurred.
  - bits [0:7]: Reserved, treat as don't care.
- word [2]
  - bit [0]: CRC error occurred.
  - bit [1]: Stream marker error detected in the image header.

- bit [2]: An error in the image header was detected which could not be corrected.
- bit [3]: A frame size error was detected, i.e. the image size defined in the CXP image header isn't matching the reconstructed frame size from the transmitted packets. This happens when the camera puts one info into the image header but transmits different amount of data as promoted in the header.
- bits [4:15]: Reserved, treat as don't care.

- word [3]
  - bit [0]: Event type, 0 = Corrupted Entity , 1 = Lost Entity.
    - **Corrupted Entity** means that the error happens within a frame and that this frame is already sourced into the VisualApplets pipeline.
    - **Lost Entity** means that the error occurred before the frame was forwarded to the following operators and the frame was discarded by the camera operator.
    - When a corrupted entity is observed, the operator will fill up the frame according to the CXP image header definition so that the following operators will not cause undefined behavior. During this fill-up, a new frame may arrive and will then get lost. The lost entity event will also be raised when the camera sends data with a gap according to the frame tag.
  - bit [1]: An event loss for type **Corrupted Entity** occurred. This means that preceding events of type **Corrupted Entity** got lost. This happens when the runtime software is not reacting to events and the internal event queues ran full.
  - bit [2]: An event loss for type **Lost Entity** occurred. This means that preceding events of type **Lost Entity** got lost. This happens when the runtime software (Framegrabber SDK) is not reacting to events and the internal event queues ran full.
  - bits [3:15]: amount of lost **Lost Entity** events.

There are two types of events: events for corrupted entities and events for lost entities. Bit 0 of word 3 describes which kind of event occurred. If the event buffers are full, it might happen that events get lost. When an event gets lost that marks a corrupted entity, bit 1 of word 3 will be set. When an event gets lost that marks a lost entity, bit 2 of word 3 will be set and bit 3 to 15 will provide the number of lost events indicating a lost frame. If bit 2 is set but the counter is 0, it means that a counter overflow happened.

Every event causes a software interrupt. To reduce the number of events, several events with the same frame tag might be merged together. In that case some error flags are combined. If an event was lost, the event before the lost event contains the information about the lost event and cannot be merged with further events with the same frame tag.

The events caused due to CRC errors report a frame tag, which may not be exactly related to the frame in which the CRC errors happen. The frame tag can be that of the preceding or following frame. This can only happen, when a camera sends a CXP packet, which contains a transition between 2 or more frames. The CRC computation is finished at the end of the packet, but the stream data is reconstructed on-the-fly. This means that a situation can happen, in which a CRC error is detected only after the preceding frame was already sent by the operator. In normal situations, in which the camera packets don't contain data both of the end of the ongoing frame and the beginning of the next frame, the frame tag during CRC error will always be correct. For all other cases as long as the complete frame stream data is less than the maximal packet size of 8k, there might be only 1 frame overlap within 1 packet. In that case, the software application should consider the preceding frame with the frame tag - 1 and the following frame with the frame tag + 1 as potentially corrupted as well.

## Differentiating Error Events Between Taps

The error handling and event system are common to both CXP tap streams. Use the stream ID field to relate the received event to the appropriate tap. Normally, tap 0 will get a lower stream ID, typically 0. Tap 1 will get a stream ID, which is larger than the one of tap 0.

### 4.1.2. FG_START_OF_FRAME_CAM_PORT_0

This event is generated when the first pixel of one camera frame arrives at the applet. 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.

The applet generates frames from linescan cameras unsing the image trigger module. The event is generated with the first pixel of the generated frame which is simultaneously to the arrival of a camera line. 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.

### 4.1.3. FG_END_OF_FRAME_CAM_PORT_0

This event is generated right after the last pixel of one camera frame arrives at the applet. 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.

The applet generates frames from linescan cameras unsing the image trigger module. The event is generated when the last pixel passes through the image trigger module. Note that this might not be at the same time as the pixel arrives from the camera at the framegrabber as the image trigger module needs to delay the data to wait for closing gates. 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. Sensor Geometry

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

## 5.1. FG_VANTAGEPOINT

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

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

If a vertical mirroring is active or *FG_TAPGEOMETRY* is set to **FG_GEOMETRY_1X_2YE** , the parameter *FG_SENSORHEIGHT* limits the maximum height. The parameter dependency will then be *FG_YOFFSET* + *FG_HEIGHT* <= *FG_SENSORHEIGHT*.

**Table 5.1. Parameter properties of FG_VANTAGEPOINT**

| Property | Value |
|---|---|
| Name | **FG_VANTAGEPOINT** |
| Display Name | **Vantage Point** |
| Type | **Enumeration** |
| Access policy | **Read/Write** |
| Storage policy | **Persistent** |
| Allowed values | **FG_VANTAGEPOINT_TOP_LEFT** Top Left<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 5.1. Usage of FG_VANTAGEPOINT**

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

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

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

## 5.2. FG_SENSORWIDTH

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

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

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

**Table 5.2. Parameter properties of FG_SENSORWIDTH**

| Property | Value |
|---|---|
| Name | **FG_SENSORWIDTH** |
| Display Name | **Sensor Width** |
| Type | **Unsigned Integer** |
| Access policy | **Read/Write** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum 48**<br>**Maximum 32768**<br>**Stepsize 16** |
| Default value | **1024** |
| Unit of measure | **pixel** |

**Example 5.2. Usage of FG_SENSORWIDTH**

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

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

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

## 5.3. FG_SENSORHEIGHT

For vertical mirroring or tap geometry sorting in vertical direction, the applet needs to be parameterized with the exact height transferred from the camera to the frame grabber. If you have set a region of interest in the camera, the parameter *FG_SENSORHEIGHT* needs to be set to the ROI size, otherwise use the sensor height. If *FG_TAPGEOMETRY* is set to **FG_GEOMETRY_1X_2YE**, the parameter *FG_SENSORHEIGHT* needs to be set to the ROI size and must be an even integer. The parameter dependency will then be: *FG_YOFFSET* + *FG_HEIGHT* <= *FG_SENSORHEIGHT*.

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

If no vertical mirroring is configured and if *FG_TAPGEOMETRY* is set to **FG_GEOMETRY_1X_1Y**, 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 2**<br>**Maximum 65536**<br>**Stepsize 1** |
| Default value | **1024** |
| Unit of measure | **pixel** |

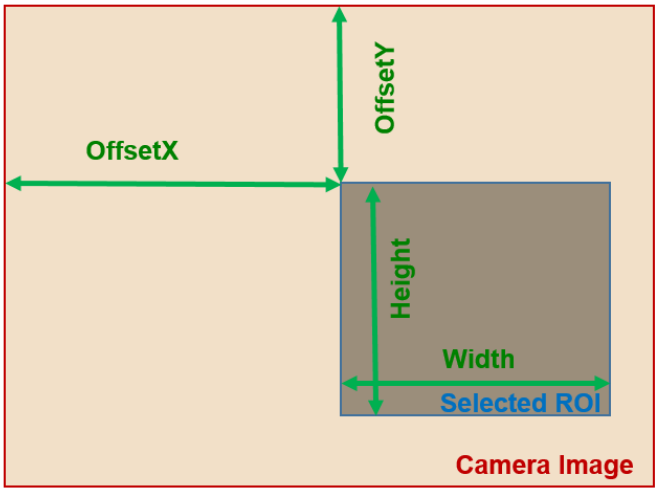
**Example 5.3. Usage of FG_SENSORHEIGHT**

```
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 */
}
```

# Chapter 6. ROI

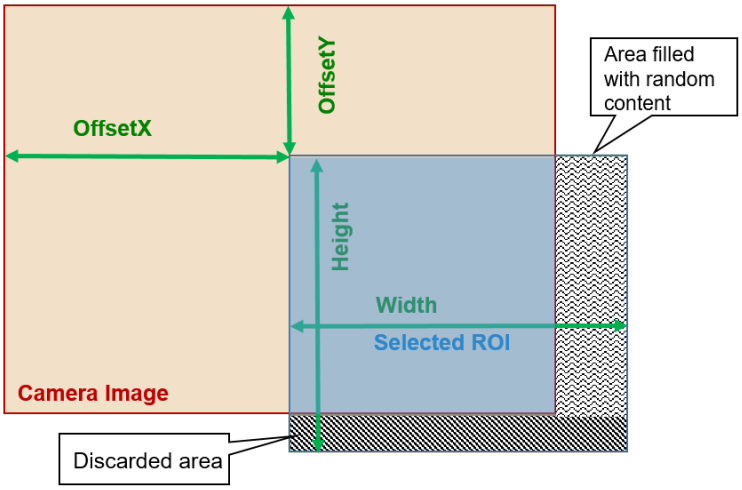
This module allows the definition of a region of interest (ROI), also called area of interest (AOI). A ROI allows the selection of a smaller subset pixel area from the input image. It is defined by using parameters *FG_XOFFSET*, *FG_WIDTH*, *FG_YOFFSET* and *FG_HEIGHT*. The following figure illustrates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

**Figure 6.1. Region of Interest**




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

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




Furthermore, mind that the image sent by the camera must not exceed the maximum allowed image dimensions. This applet allows a maximum image width of 32768 pixels and a maximum image height of 65536 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 16 pixel for the width and the x-offset, while the step size for the height and the y-offset is 1.

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

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

### Camera ROI

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

### Influence on Bandwidth

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

## 6.1. FG_WIDTH

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

### Maximum image width is reduced for horizontal mirrored images

Limitations of the available BRAM in the FPGA allow only to store smaller lines and there for the images that can be mirrored needs to be smaller. A mirrored image can only have width of 21840, the not mirrored image can have the full width of 32768.

**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 48**<br>**Maximum 32768**<br>**Stepsize 16** |
| 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 65536 pixels. If a vertical mirroring is active the sensor height limits the maximum height (Height + YOffset). Furthermore he maximum height is limited by the DRAM and the sensor width (the sensor dimension needs to fit into the DRAM).

**Table 6.2. Parameter properties of FG_HEIGHT**

| Property | Value |
|---|---|
| Name | **FG_HEIGHT** |
| Display Name | **Height** |
| Type | **Unsigned Integer** |
| Access policy | **Read/Write** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum 2**<br>**Maximum 65536**<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. If a horizontal mirroring is active the sensor width limits the maximum width (Width + XOffset). If furthermore vertical mirroring is active the maximum width is limited by the DRAM and the sensor height (the sensor dimension needs to fit into the DRAM).

**Table 6.3. Parameter properties of FG_XOFFSET**

| Property | Value |
|---|---|
| Name | **FG_XOFFSET** |
| Display Name | **Offset X** |
| Type | **Unsigned Integer** |
| Access policy | **Read/Write** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum 0**<br>**Maximum 32720**<br>**Stepsize 16** |
| Default value | **0** |
| Unit of measure | **pixel** |

**Example 6.3. Usage of FG_XOFFSET**

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

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

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

## 6.4. FG_YOFFSET

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

**Table 6.4. Parameter properties of FG_YOFFSET**

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

**Example 6.4. Usage of FG_YOFFSET**

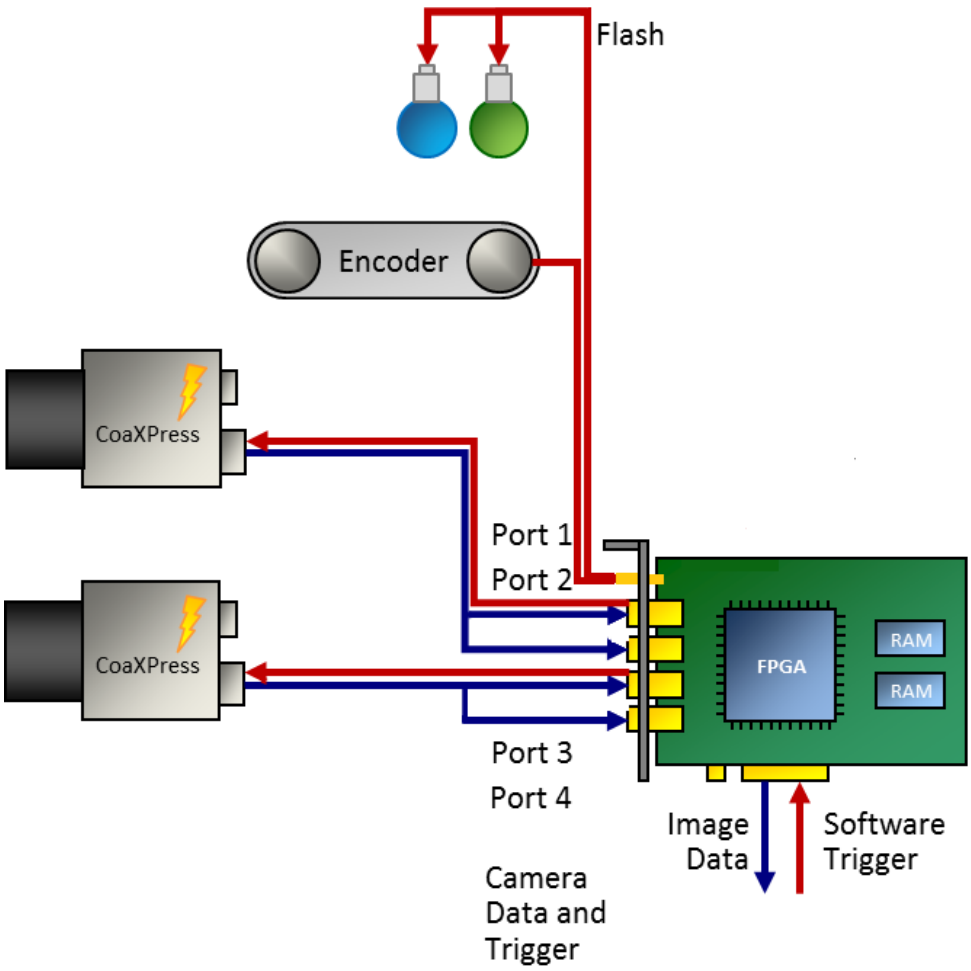
```
int result = 0;
unsigned int value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

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 interface card and the connected cameras. 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 CXP-12 Interface Card 4C interface card has 4 inputs on the front IO connector. Check the hardware documentation for more information. The CXP-12 Interface Card 4C 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 CoaXPress link. 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. CXP-12 Interface Card 4C Trigger System**




The following is an introduction into the Basler CXP-12 Interface Card 4C trigger system. Several trigger scenarios show the possibilities and functionalities and 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

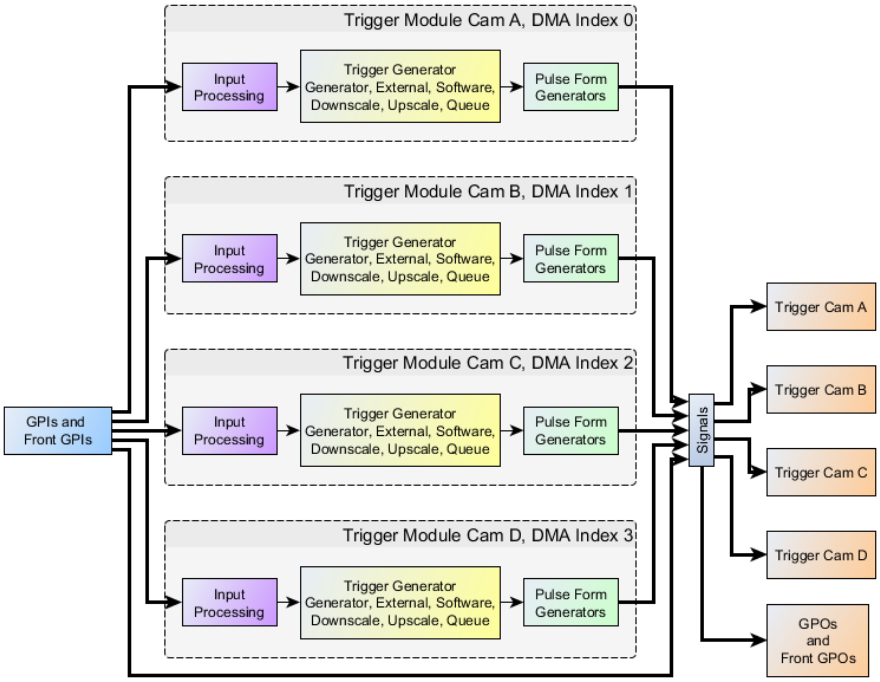
The Basler trigger system was designed to fulfill the requirements of various applications. Powerful features for trigger generation, controlling and monitoring were included in the implementation. This includes:

- 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 interface card type** plus the CoaXPress trigger 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 interface card 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**

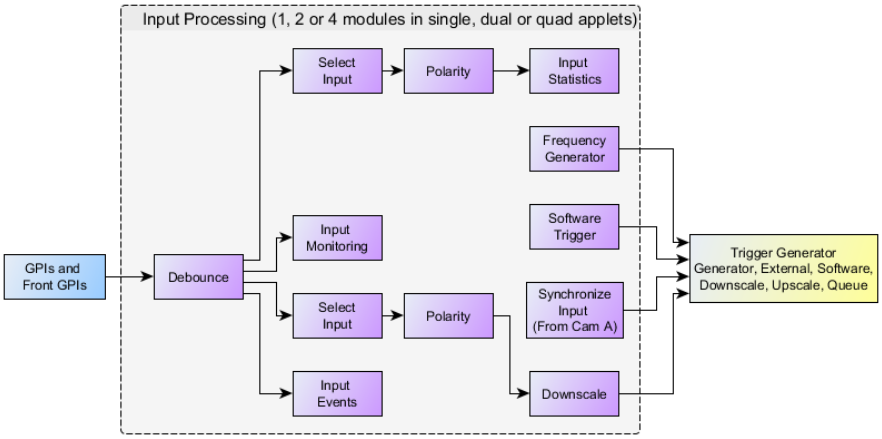



1. Trigger Input:

   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.

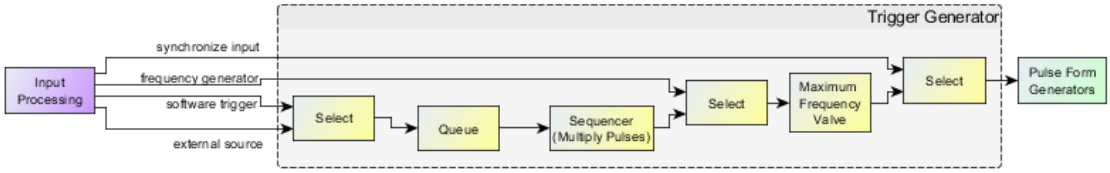
   **Figure 7.3. Trigger Input Block Diagram**




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.

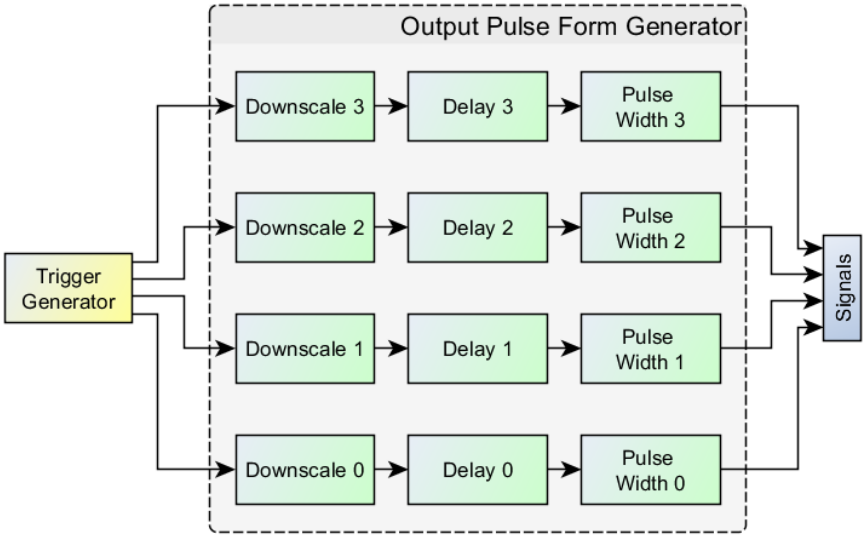
**Figure 7.4. Trigger Pulse Processing Block Diagram**




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.

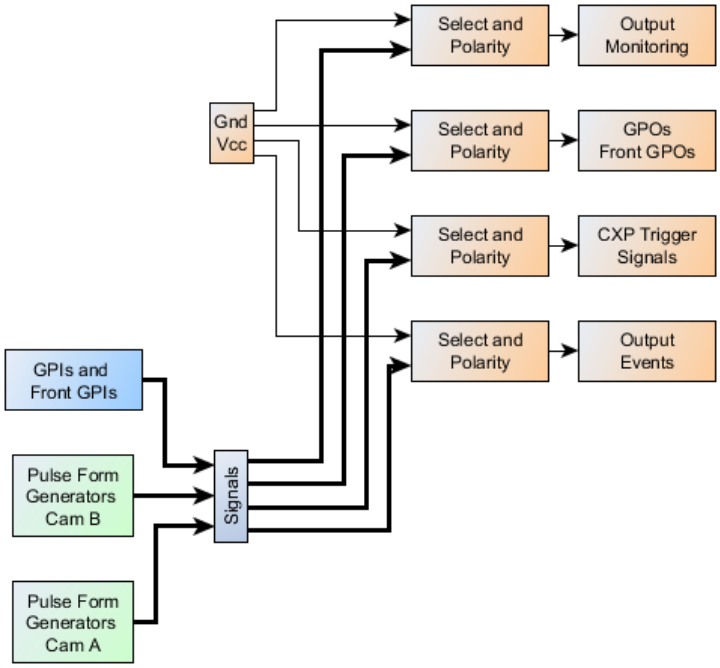
**Figure 7.5. Trigger Pulse Processing Block Diagram**




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 CXP-12 Interface Card 4C supports four digital front inputs. It has two front trigger outputs.

The four front inputs have the indices 0 to 3. In the documentation of the trigger IO boards and CXP-12 Interface Card 4C 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. The events are available for all cameras. Replace CAM0 by the respective camera index if necessary.

- *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*

This event is generated when the trigger queue exceeds the upper ON-threshold or falls below 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.

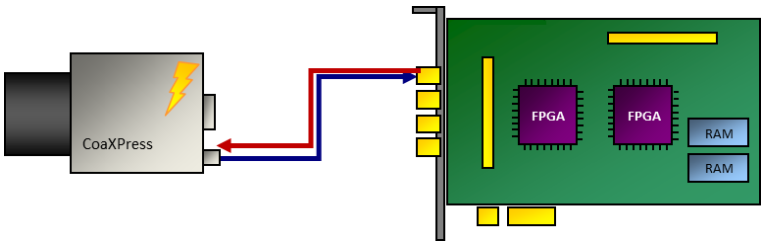
The scenarios show real life configurations. They explain the requirements, illustrate the inputs and outputs and list the required parameters and their values.

### 7.4.1. Internal Frequency Generator / interface card 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 interface card. Assume that there is no additional external source for trigger events and we do not need to control any flash lights. Thus the interface card'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 interface card is required.

**Figure 7.7. Generator Controlled Trigger Scenario**

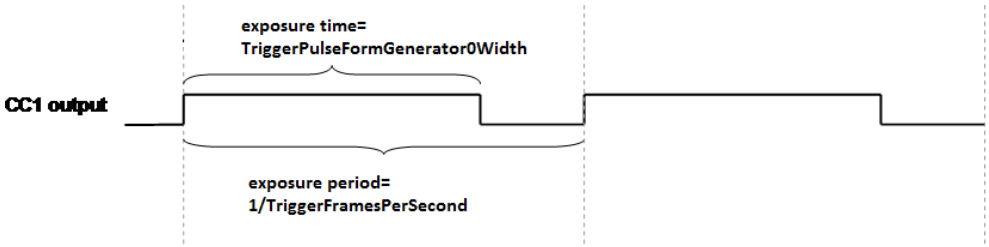



To put this scenario into practice, you will need to set your camera into an external trigger mode. Consult the vendor's user manual for more information. CXP cameras are configured using GenICam. Use the Basler GenICam explorer to set the camera into external trigger mode.

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. Use the CXP cable as trigger source. 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 interface card, the respective trigger mode needs to support this, too.

In the following, a waveform is shown which illustrates the interface card 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 interface card 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 interface card'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 interface card'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_TRIGGERCAMERA_SOURCE_CXP0* = **PULSEGEN0_RISING** and *FG_TRIGGERCAMERA_SOURCE_CXP1* = **PULSEGEN0_FALLING**

  The only thing left to do is to allocate the output of pulse form generator 0 to the camera trigger output.

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 interface card. These images are discarded as no acquisition is started.

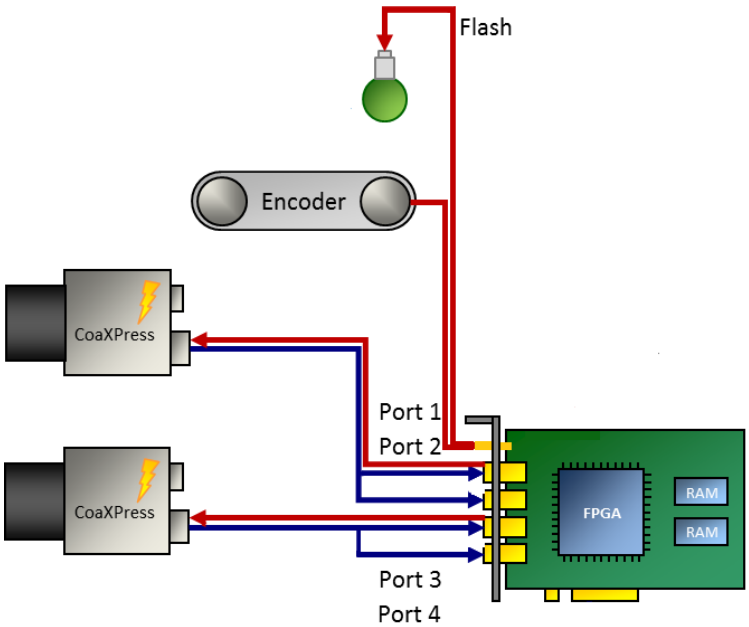
You will now receive images from your camera. Change the frequency and the signal width to see the influence of these parameters. A higher frequency will give you a higher frame rater. A shorter exposure time will make the images 'darker'. You will realize, that it is not possible to set an exposure 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.

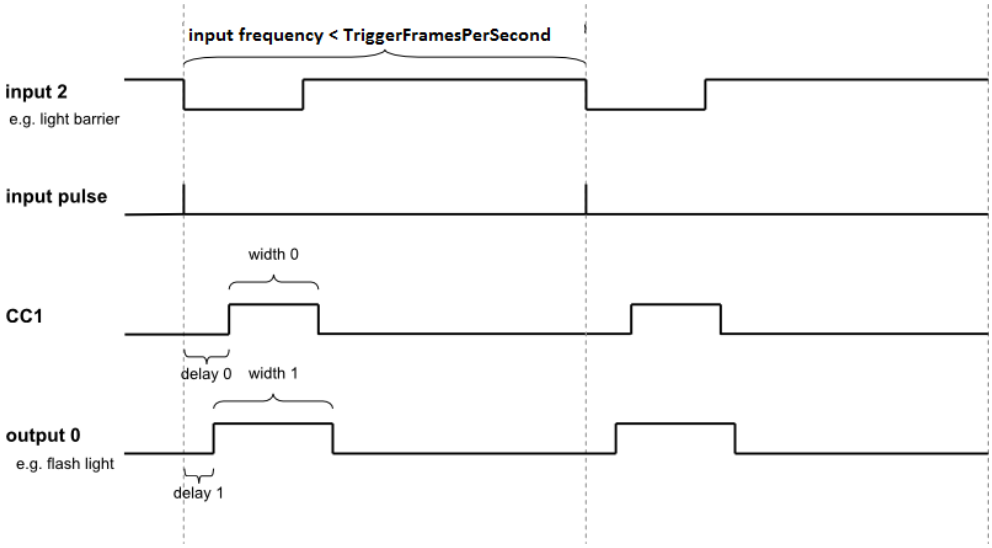
**Figure 7.9. External Controlled Trigger Scenario**




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 the trigger camera output 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 trigger camera 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_FRONT_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_TRIGGERCAMERA_SOURCE_CXP0* = **PULSEGEN0_RISING** and *FG_TRIGGERCAMERA_SOURCE_CXP1* = **PULSEGEN0_FALLING**

  Finally, we have to allocate the camera trigger output 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.

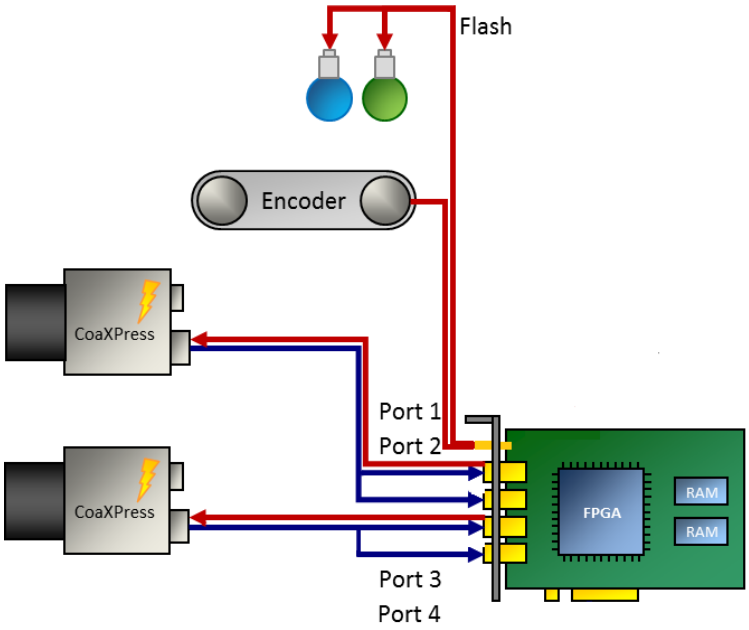
### 7.4.3. Control of Two Flash Lights

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. Images one and three out of the sequence of three images have to use the first light source and image two has to use the second light source. Thus, in this scenario we will learn on how to use a trigger pulse multiplication and on how to control two lights connected to the interface card.

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 two 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 two images with the different illuminations.

**Figure 7.11. External Controlled Trigger Scenario**



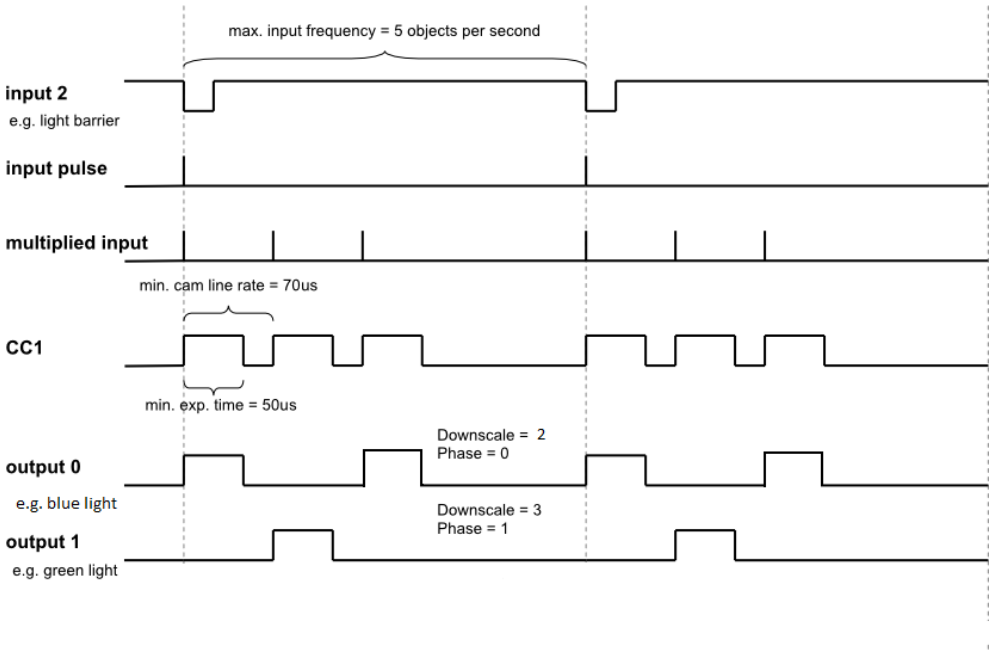

Before looking at the waveform, let's have a look at our fictitious hardware specifications.

**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 and 1 are downscaled by two and three and a delay is added.

**Figure 7.12. Waveform of External Trigger Scenario Controlling Two 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_PULSEFORMGEN1_DOWNSCALE* = 2
- *FG_TRIGGER_PULSEFORMGEN2_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_TRIGGERCAMERA_SOURCE_CXP0* = **PULSEGEN0_RISING** and *FG_TRIGGERCAMERA_SOURCE_CXP1* = **PULSEGEN0_FALLING**

  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 interface card 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

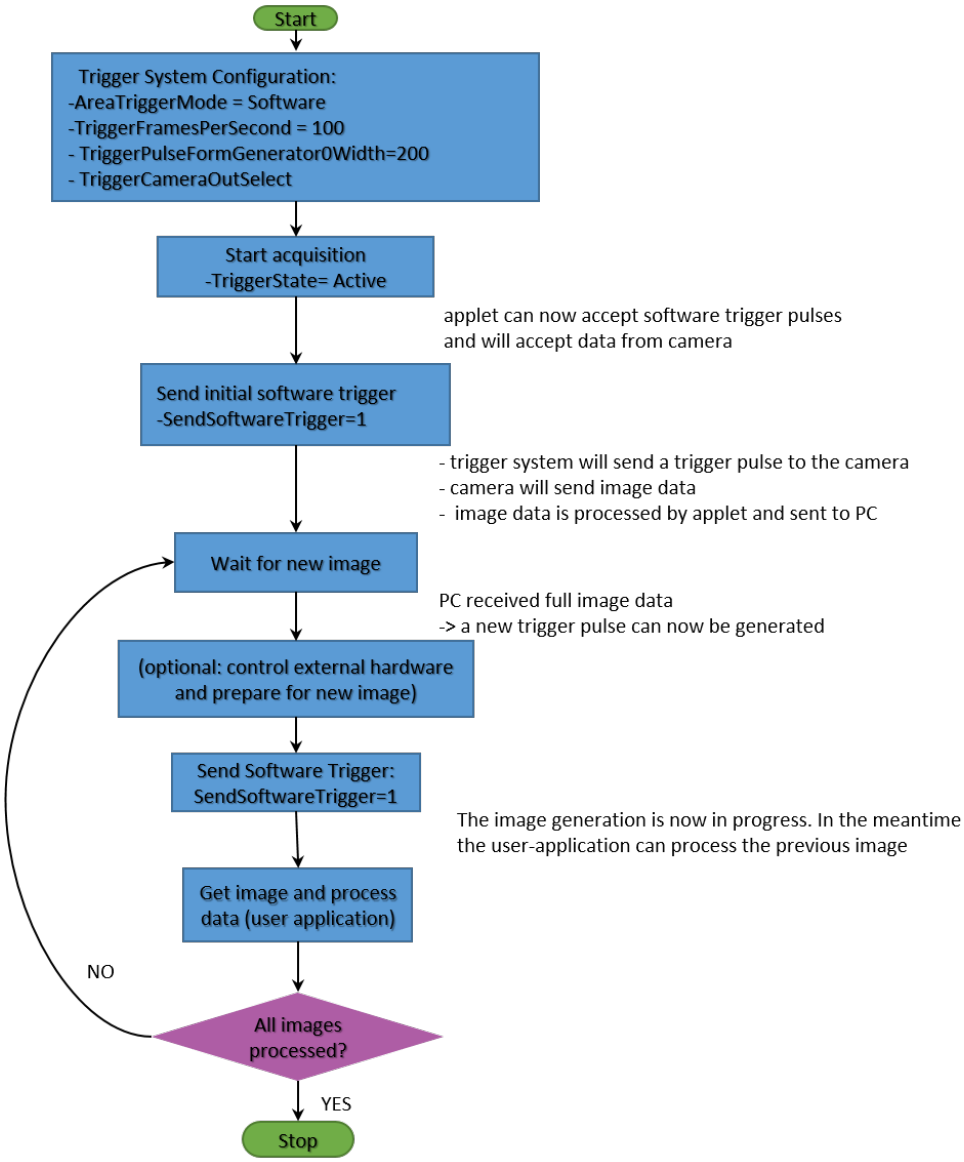
The previous examples showed the use of the internal frequency generator and the use of external trigger pulses to trigger your camera and generate digital output signals. Another trigger mode is the software trigger. In this mode, you can control the generation of each trigger pulse using your software 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 interface card 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 interface card 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.

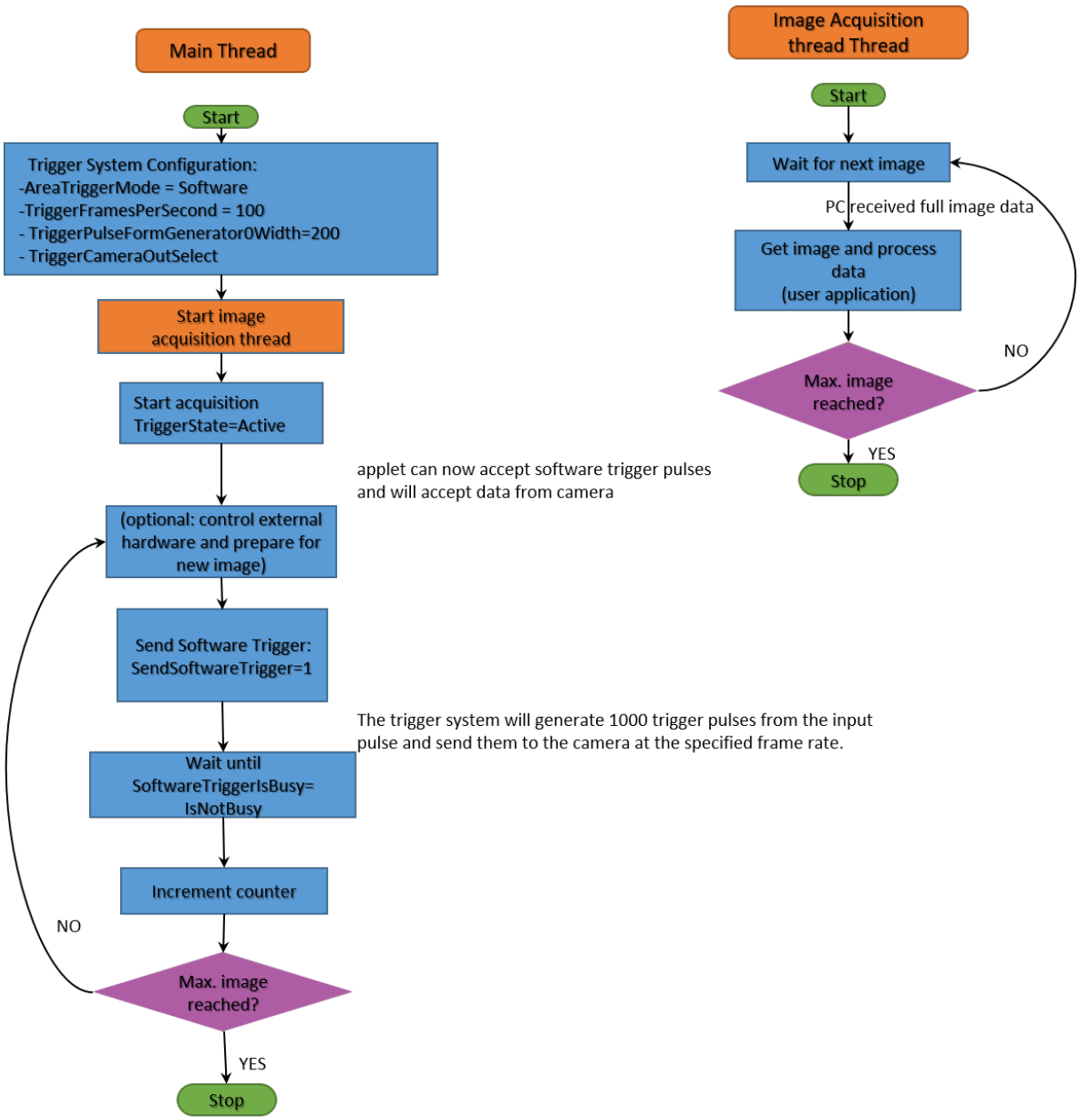
**Figure 7.13. Flowchart of Software Application Using the Software Trigger**




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 interface card, 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 interface card 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 interface card 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

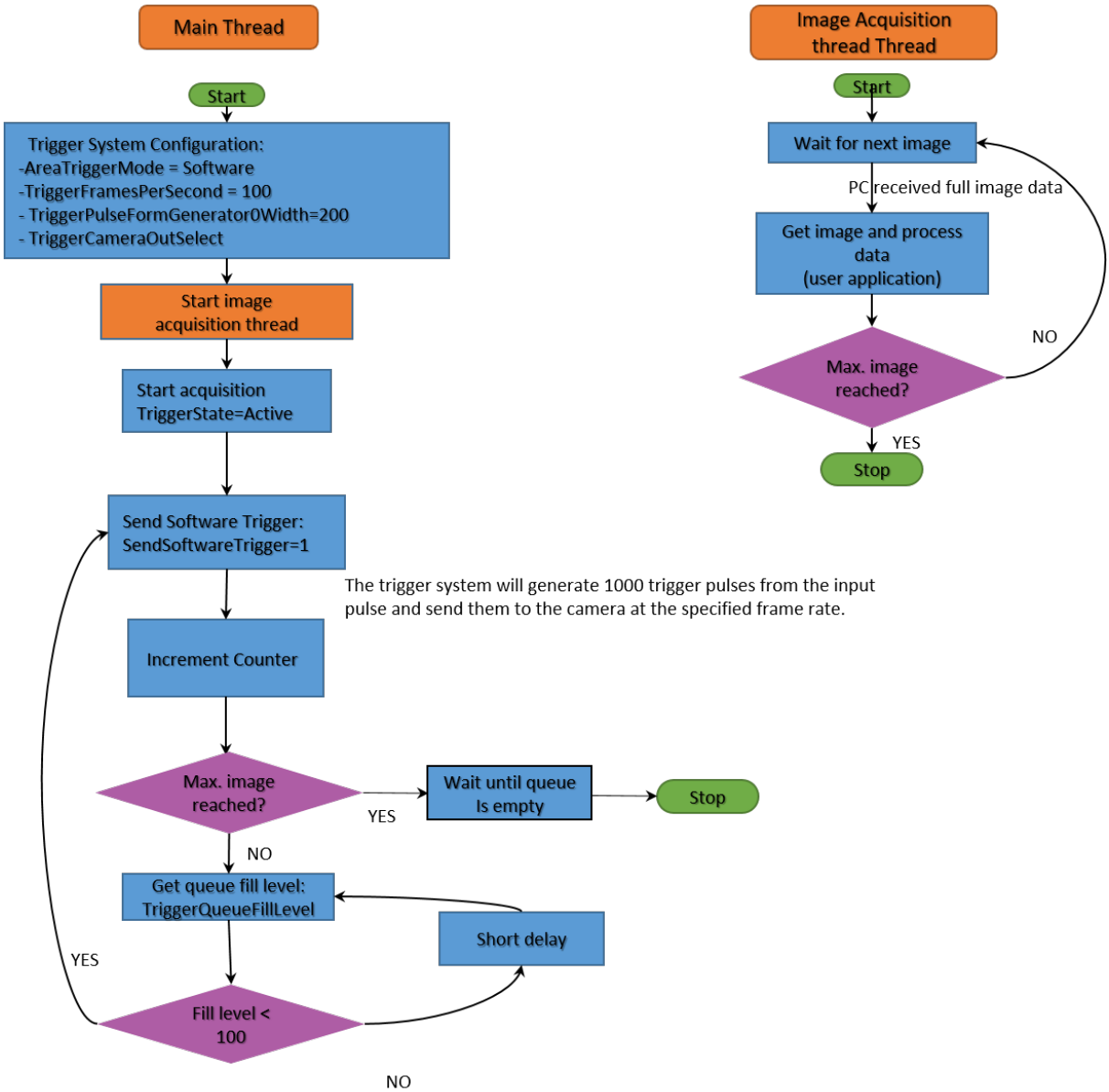
To understand the following scenario you should have read the previous scenario first. In the following we will have a look at the software trigger once again. This time, we use the trigger queue. The trigger queue enables the buffering of trigger pulses from external sources or from the software trigger 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.

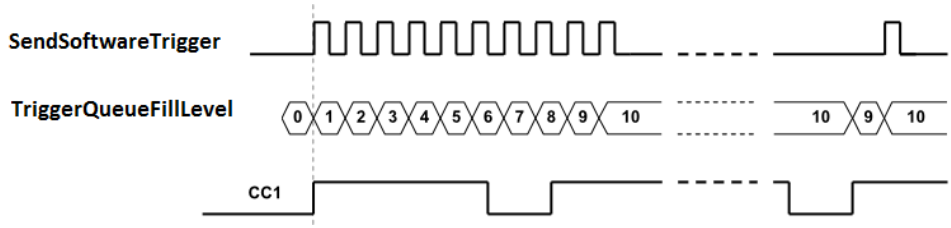
**Figure 7.15. Flowchart of Software Application Using the Software Trigger with Trigger Queue**




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

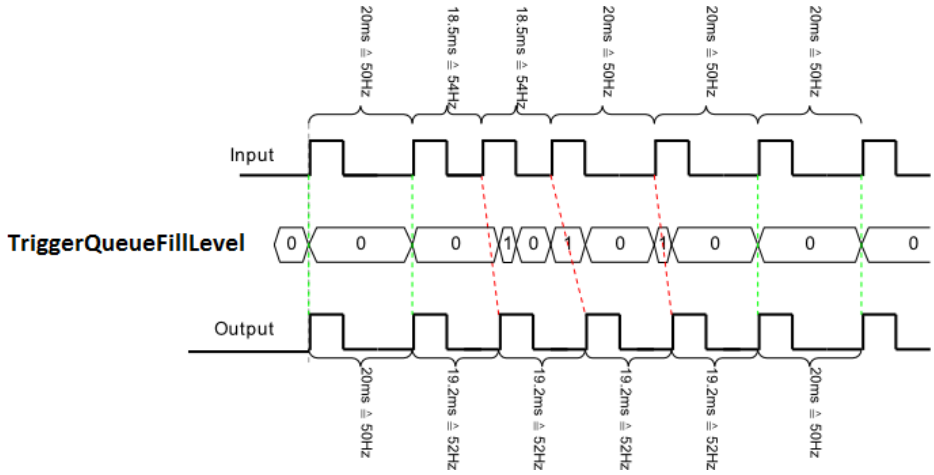
Of course, we can use the trigger queue with external triggers, too. This will give us a possibility to buffer 'jumpy' external encoders or any other external trigger signal generators. Let's suppose an external encoder which is configured to generate trigger pulses with a frequency of 50Hz and a camera 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.

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

- *FG_TRIGGERCAMERA_SOURCE_CXP0* = BYPASS_FRONT_GPI_0_RISING and *FG_TRIGGERCAMERA_SOURCE_CXP1* = BYPASS_FRONT_GPI_0_FALLING
- *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 interface cards 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 interface cards and set the same trigger configuration for all cameras. This applet supports up to two cameras. Set the same parameters for both cameras. Multiple trigger systems are allowed to share the same trigger input, so you do not have to connect your trigger source to two inputs.

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 interface cards which have to be synchronized with the master. In the slave applets bypass the input to the camera control outputs. In addition to that, this applet includes a special trigger mode called **ATM_SYNCHRONIZED**. This mode can be chosen for the second camera. The second camera uses the trigger pulses at the output of the first camera as input source. However, users will still have to configure the pulse form generators in the trigger system of the second camera and will still have to allocate them to the trigger outputs.

### Arbitrary Output Allocation

In multiple camera applets you can also select another camera trigger module source. For example, CXP trigger source for camera 1 can use **CAM_A_PULSEGEN0**.

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

The Basler 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 interface card 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 interface card 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 for the first camera. If the area trigger mode of a process (Process) is set to synchronized mode, the trigger source of the process will be the output of the previous process. For example, if the area trigger mode of process 1 is set to synchronized mode, the trigger system is sourced by the output of process 0.

  In Section 7.4.8, 'Multi Camera Applications / Synchronized Cameras' an example of the usage is presented. The block diagram in Figure 7.2, 'Trigger System' illustrates the sources of the synchronize outputs and inputs.

**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**<br>**ATM_EXTERNAL**<br>**ATM_SOFTWARE**<br>**ATM_SYNCHRONIZED** | Generator<br>External<br>Software<br>Synchronized |
| 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 for the respective camera process. 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**<br>**TS_ASYNC_STOP**<br>**TS_SYNC_STOP** | Active<br>Async Stop<br>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**

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

- fps = *FG_TRIGGER_FRAMESPERSECOND*
- WIDTH[0..3] = *FG_TRIGGER_PULSEFORMGEN[0..3]_WIDTH* *FG_TRIGGER_PULSEFORMGEN[0..3]_WIDTH*
- DELAY[0..3] = *FG_TRIGGER_PULSEFORMGEN[0..3]_DELAY* *FG_TRIGGER_PULSEFORMGEN[0..3]_DELAY*
- DOWNSCALE[0..3] = *FG_TRIGGER_PULSEFORMGEN[0..3]_DOWNSCALE* *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 | **Trigger Output Frequency** |
| Type | **Double** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum 0.01455191523**<br>**Maximum 3.1249999999999996E7**<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

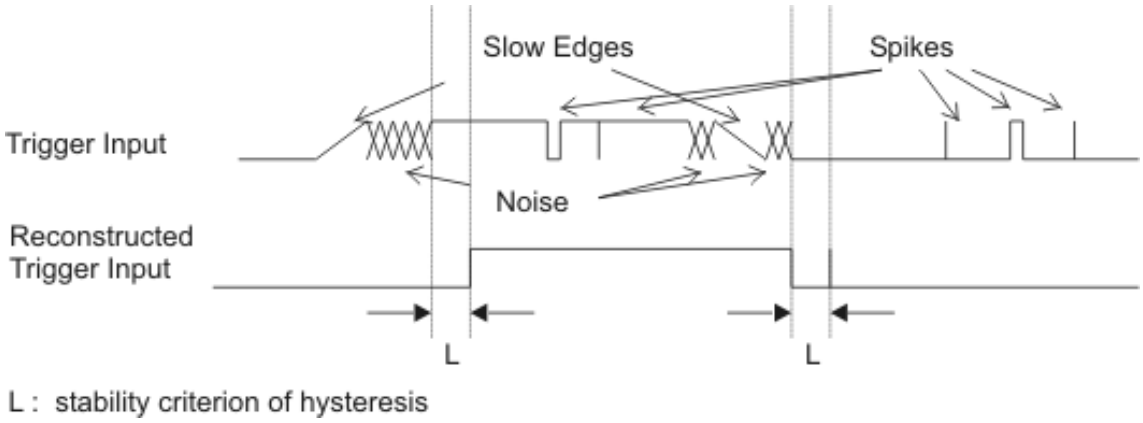
The parameters of category Trigger Input are used to configure the input source of the trigger system. The category is divided into sub categories. All external sources are configured in category external. Category software trigger allows the configuration, monitoring and controlling of software trigger 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. The parameter is camera process independent i.e. the latest settings will apply for all camera 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 0.0**<br>**Maximum 209.71200000000002**<br>**Stepsize 0.0032** |
| 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_FRONT_GPI

Parameter *FG_FRONT_GPI* is used to monitor the digital inputs of the interface card.

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 interface card will have high level on it's digital inputs 1 and 3.

**Table 7.6. Parameter properties of FG_FRONT_GPI**

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

**Example 7.5. 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 */
}
```

### 7.5.4.1.3. 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. four Front GPI inputs. If *FG_AREATRIGGERMODE* is not set to external, this parameter will select the input for the input statistics only.

**Table 7.7. Parameter properties of FG_TRIGGERIN_SRC**

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

**Example 7.6. Usage of FG_TRIGGERIN_SRC**

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

### 7.5.4.1.4. FG_TRIGGERIN_POLARITY

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

**Table 7.8. Parameter properties of FG_TRIGGERIN_POLARITY**

| Property | Value |
|---|---|
| Name | **FG_TRIGGERIN_POLARITY** |
| Display Name | **Trigger In 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 7.7. 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 */
}
```

### 7.5.4.1.5. 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 7.9. Parameter properties of FG_TRIGGERIN_DOWNSCALE**

| Property | Value |
|---|---|
| Name | **FG_TRIGGERIN_DOWNSCALE** |
| Display Name | **Trigger In 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** |

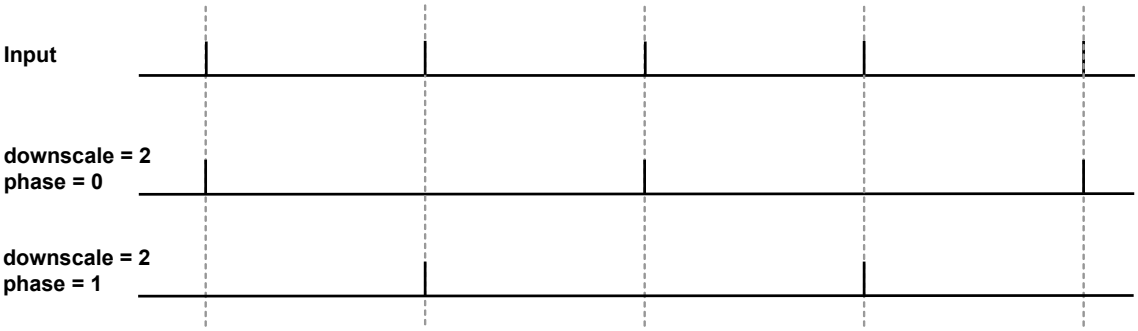
**Example 7.8. Usage of FG_TRIGGERIN_DOWNSCALE**

```
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 */
}
```

### 7.5.4.1.6. 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 7.10. Parameter properties of FG_TRIGGERIN_DOWNSCALE_PHASE**

| Property | Value |
|---|---|
| Name | **FG_TRIGGERIN_DOWNSCALE_PHASE** |
| Display Name | **Trigger In 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 7.9. 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 interface card. 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 7.11. Parameter properties of FG_SENDSOFTWARETRIGGER**

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

**Example 7.10. 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 7.12. Parameter properties of FG_SOFTWARETRIGGER_IS_BUSY**

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

**Example 7.11. 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 7.13. Parameter properties of FG_SOFTWARETRIGGER_QUEUE_FILLLEVEL**

| Property | Value |
|---|---|
| Name | **FG_SOFTWARETRIGGER_QUEUE_FILLLEVEL** |
| Display Name | **Software Trigger Queue Fill Level** |
| 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 7.12. 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 inputs.

**Table 7.14. Parameter properties of FG_TRIGGERIN_STATS_SOURCE**

| Property | Value |
|---|---|
| Name | **FG_TRIGGERIN_STATS_SOURCE** |
| Display Name | **Trigger In Statistics Source** |
| Type | **Enumeration** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **TRGINSRC_FRONT_GPI_0** Trigger In Source Front GPI 0<br>**TRGINSRC_FRONT_GPI_1** Trigger In Source Front GPI 1<br>**TRGINSRC_FRONT_GPI_2** Trigger In Source Front GPI 2<br>**TRGINSRC_FRONT_GPI_3** Trigger In Source Front GPI 3 |
| Default value | **TRGINSRC_FRONT_GPI_0** |

**Example 7.13. Usage of FG_TRIGGERIN_STATS_SOURCE**

```
int result = 0;
int value = TRGINSRC_FRONT_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 7.15. Parameter properties of FG_TRIGGERIN_STATS_POLARITY**

| Property | Value |
|---|---|
| Name | **FG_TRIGGERIN_STATS_POLARITY** |
| Display Name | **Trigger In 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 7.14. 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 7.16. Parameter properties of FG_TRIGGERIN_STATS_PULSECOUNT**

| Property | Value |
|---|---|
| Name | **FG_TRIGGERIN_STATS_PULSECOUNT** |
| Display Name | **Trigger In Statistics 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 7.15. 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 7.17. Parameter properties of FG_TRIGGERIN_STATS_PULSECOUNT_CLEAR**

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

**Example 7.16. 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 interface card.

Table 7.18. Parameter properties of FG_TRIGGERIN_STATS_FREQUENCY

| Property | Value |
|---|---|
| Name | **FG_TRIGGERIN_STATS_FREQUENCY** |
| Display Name | **Trigger In Statistics Frequency** |
| Type | **Double** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **Minimum 0.0**<br>**Maximum 3.125E8**<br>**Stepsize 2.220446049250313E-16** |
| Unit of measure | **Hz** |

Example 7.17. 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 7.19. Parameter properties of FG_TRIGGERIN_STATS_MINFREQUENCY

| Property | Value |
|---|---|
| Name | **FG_TRIGGERIN_STATS_MINFREQUENCY** |
| Display Name | **Trigger In Statistics Minimum Frequency** |
| Type | **Double** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **Minimum 0.0**<br>**Maximum 3.125E8**<br>**Stepsize 2.220446049250313E-16** |
| Unit of measure | **Hz** |

Example 7.18. 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 7.20. Parameter properties of FG_TRIGGERIN_STATS_MAXFREQUENCY

| Property | Value |
|---|---|
| Name | **FG_TRIGGERIN_STATS_MAXFREQUENCY** |
| Display Name | **Trigger In Statistics Maximum Frequency** |
| Type | **Double** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **Minimum** 0.0<br>**Maximum** 3.125E8<br>**Stepsize** 2.220446049250313E-16 |
| Unit of measure | **Hz** |

Example 7.19. 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 7.21. Parameter properties of FG_TRIGGERIN_STATS_MINMAXFREQUENCY_CLEAR

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

Example 7.20. 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_FRONT_GPI0_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_FRONT_GPI0_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. Basler 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 7.22. 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 7.21. 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 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 thresholds. 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 7.23. Parameter properties of FG_TRIGGERQUEUE_MODE**

| Property | Value |
|---|---|
| Name | **FG_TRIGGERQUEUE_MODE** |
| Display Name | **Trigger 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 7.22. 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 7.24. Parameter properties of FG_TRIGGERQUEUE_FILLLEVEL**

| Property | Value |
|---|---|
| Name | **FG_TRIGGERQUEUE_FILLLEVEL** |
| Display Name | **Trigger Queue Fill Level** |
| 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 7.23. 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 7.25. Parameter properties of FG_TRIGGER_QUEUE_FILLLEVEL_EVENT_ON_THRESHOLD**

| Property | Value |
|---|---|
| Name | **FG_TRIGGER_QUEUE_FILLLEVEL_EVENT_ON_THRESHOLD** |
| Display Name | **Trigger 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 7.24. 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 7.26. Parameter properties of FG_TRIGGER_QUEUE_FILLLEVEL_EVENT_OFF_THRESHOLD**

| Property | Value |
|---|---|
| Name | **FG_TRIGGER_QUEUE_FILLLEVEL_EVENT_OFF_THRESHOLD** |
| Display Name | **Trigger 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 7.25. 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.

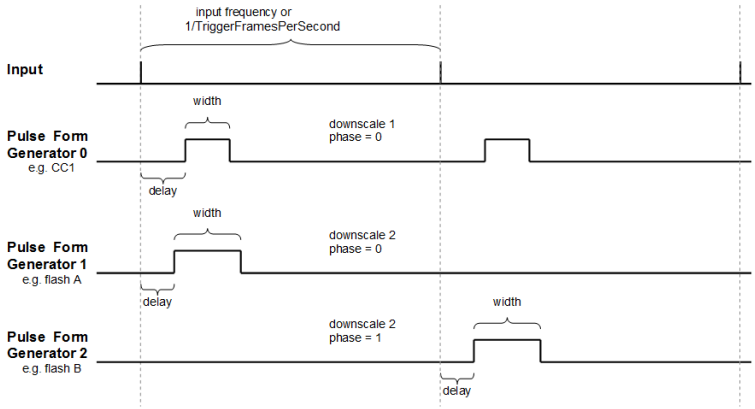
## 7.5.7. Pulse Form Generator 0

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* *FG_TRIGGER_PULSEFORMGEN[0..3]_WIDTH*
- DELAY[0..3] = *FG_TRIGGER_PULSEFORMGEN[0..3]_DELAY* *FG_TRIGGER_PULSEFORMGEN[0..3]_DELAY*
- DOWNSCALE[0..3] = *FG_TRIGGER_PULSEFORMGEN[0..3]_DOWNSCALE* *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* *FG_TRIGGER_PULSEFORMGEN[0..3]_DOWNSCALE_PHASE* for more information.

**Table 7.27. Parameter properties of FG_TRIGGER_PULSEFORMGEN0_DOWNSCALE**

| Property | Value |
|---|---|
| Name | **FG_TRIGGER_PULSEFORMGEN0_DOWNSCALE** |
| Display Name | **Trigger Pulse Form Generator0 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 7.26. 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* *FG_TRIGGER_PULSEFORMGEN[0..3]_DOWNSCALE* defines the number of phases and parameter *FG_TRIGGER_PULSEFORMGEN[0..3]_DOWNSCALE_PHASE* *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 7.28. Parameter properties of FG_TRIGGER_PULSEFORMGEN0_DOWNSCALE_PHASE**

| Property | Value |
|---|---|
| Name | **FG_TRIGGER_PULSEFORMGEN0_DOWNSCALE_PHASE** |
| Display Name | **Trigger Pulse Form Generator0 Downscale 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 7.27. 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 7.29. Parameter properties of FG_TRIGGER_PULSEFORMGEN0_DELAY**

| Property | Value |
|---|---|
| Name | **FG_TRIGGER_PULSEFORMGEN0_DELAY** |
| Display Name | **Trigger Pulse Form Generator0 Delay** |
| Type | **Double** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum 0.0**<br>**Maximum 3.4E7**<br>**Stepsize 0.0032** |
| Default value | **0.0** |
| Unit of measure | **µs** |

**Example 7.28. 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 7.30. Parameter properties of FG_TRIGGER_PULSEFORMGEN0_WIDTH**

| Property | Value |
|---|---|
| Name | **FG_TRIGGER_PULSEFORMGEN0_WIDTH** |
| Display Name | **Trigger Pulse Form Generator0 Width** |
| Type | **Double** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum 0.0032**<br>**Maximum 6.8E7**<br>**Stepsize 0.0032** |
| Default value | **4.0** |
| Unit of measure | **µs** |

**Example 7.29. 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. Camera Out Signal Mapping

The camera interface of the CXP-12 Interface Card 4C is equipped with a trigger output channel to trigger the camera.

Moreover, two front general purpose outputs (GPOs) to the camera exist.
To identify the required signals and their mapping, consult the vendor's manual of your camera.

The trigger system of this applet provides several possibilities of mapping pulse sources to the camera channels:

- Pulse form generators 0 to 3

  The pulse form generators are the main output sources of the trigger system. You can map either the start or the end of the signals to the CXP camera port. If your camera runs in **`Timed`** mode with external CXP trigger, you only need to send the start signal, usually on CXP LinkTrigger0. If your camera runs in **`TriggerControlled`** mode, you need to send the start of exposure signal on CXP LinkTrigger0 and the end of exposure signal on CXP LinkTrigger1.

- If you don't need to send any pulses on CXP LinkTrigger, set *FG_TRIGGERCAMERA_SOURCE_CXP0* to **`GND`**.

- The input bypass

  The frame grabber trigger system ignores the signal length of the input signals. If you want to directly bypass one of the input signals to a CXP LinkTrigger, you can select the input start or end signal of pulse, i.e. rising or falling edge.

#### 7.5.11.1. FG_TRIGGERCAMERA_SOURCE_CXP0

**Table 7.31. Parameter properties of FG_TRIGGERCAMERA_SOURCE_CXP0**

| Property | Value | |
|---|---|---|
| Name | **FG_TRIGGERCAMERA_SOURCE_CXP0** | |
| Display Name | **CXP Link Trigger 0 Source** | |
| Type | **Enumeration** | |
| Access policy | **Read/Write/Change** | |
| Storage policy | **Persistent** | |
| Allowed values | **GND** | GND |
| | **CAM_A_PULSEGEN0_RISING** | Cam A Pulse Generator 0 Rising |
| | **CAM_A_PULSEGEN1_RISING** | Cam A Pulse Generator 1 Rising |
| | **CAM_A_PULSEGEN2_RISING** | Cam A Pulse Generator 2 Rising |
| | **CAM_A_PULSEGEN3_RISING** | Cam A Pulse Generator 3 Rising |
| | **CAM_A_PULSEGEN0_FALLING** | Cam A Pulse Generator 0 Falling |
| | **CAM_A_PULSEGEN1_FALLING** | Cam A Pulse Generator 1 Falling |
| | **CAM_A_PULSEGEN2_FALLING** | Cam A Pulse Generator 2 Falling |
| | **CAM_A_PULSEGEN3_FALLING** | Cam A Pulse Generator 3 Falling |
| | **CAM_B_PULSEGEN0_RISING** | Cam B Pulse Generator 0 Rising |
| | **CAM_B_PULSEGEN1_RISING** | Cam B Pulse Generator 1 Rising |
| | **CAM_B_PULSEGEN2_RISING** | Cam B Pulse Generator 2 Rising |
| | **CAM_B_PULSEGEN3_RISING** | Cam B Pulse Generator 3 Rising |
| | **CAM_B_PULSEGEN0_FALLING** | Cam B Pulse Generator 0 Falling |
| | **CAM_B_PULSEGEN1_FALLING** | Cam B Pulse Generator 1 Falling |
| | **CAM_B_PULSEGEN2_FALLING** | Cam B Pulse Generator 2 Falling |
| | **CAM_B_PULSEGEN3_FALLING** | Cam B Pulse Generator 3 Falling |
| | **BYPASS_FRONT_GPI_0_RISING** | Bypass Front GPI 0 Rising |
| | **BYPASS_FRONT_GPI_0_FALLING** | Bypass Front GPI 0 Falling |
| | **BYPASS_FRONT_GPI_1_RISING** | Bypass Front GPI 1 Rising |
| | **BYPASS_FRONT_GPI_1_FALLING** | Bypass Front GPI 1 Falling |
| | **BYPASS_FRONT_GPI_2_RISING** | Bypass Front GPI 2 Rising |
| | **BYPASS_FRONT_GPI_2_FALLING** | Bypass Front GPI 2 Falling |
| | **BYPASS_FRONT_GPI_3_RISING** | Bypass Front GPI 3 Rising |
| | **BYPASS_FRONT_GPI_3_FALLING** | Bypass Front GPI 3 Falling |
| | **PULSEGEN0_RISING** | Pulse Generator 0 Rising |
| | **PULSEGEN1_RISING** | Pulse Generator 1 Rising |
| | **PULSEGEN2_RISING** | Pulse Generator 2 Rising |
| | **PULSEGEN3_RISING** | Pulse Generator 3 Rising |
| | **PULSEGEN0_FALLING** | Pulse Generator 0 Falling |
| | **PULSEGEN1_FALLING** | Pulse Generator 1 Falling |
| | **PULSEGEN2_FALLING** | Pulse Generator 2 Falling |
| | **PULSEGEN3_FALLING** | Pulse Generator 3 Falling |
| Default value | **PULSEGEN0_RISING** | |

**Example 7.30. Usage of FG_TRIGGERCAMERA_SOURCE_CXP0**

```
int result = 0;
int value = PULSEGEN0_RISING;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

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

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

### 7.5.11.2. FG_TRIGGERCAMERA_SOURCE_CXP1

**Table 7.32. Parameter properties of FG_TRIGGERCAMERA_SOURCE_CXP1**

| Property | Value | |
|---|---|---|
| Name | **FG_TRIGGERCAMERA_SOURCE_CXP1** | |
| Display Name | **CXP Link Trigger 1 Source** | |
| Type | **Enumeration** | |
| Access policy | **Read/Write/Change** | |
| Storage policy | **Persistent** | |
| Allowed values | **GND** | GND |
| | **CAM_A_PULSEGEN0_RISING** | Cam A Pulse Generator 0 Rising |
| | **CAM_A_PULSEGEN1_RISING** | Cam A Pulse Generator 1 Rising |
| | **CAM_A_PULSEGEN2_RISING** | Cam A Pulse Generator 2 Rising |
| | **CAM_A_PULSEGEN3_RISING** | Cam A Pulse Generator 3 Rising |
| | **CAM_A_PULSEGEN0_FALLING** | Cam A Pulse Generator 0 Falling |
| | **CAM_A_PULSEGEN1_FALLING** | Cam A Pulse Generator 1 Falling |
| | **CAM_A_PULSEGEN2_FALLING** | Cam A Pulse Generator 2 Falling |
| | **CAM_A_PULSEGEN3_FALLING** | Cam A Pulse Generator 3 Falling |
| | **CAM_B_PULSEGEN0_RISING** | Cam B Pulse Generator 0 Rising |
| | **CAM_B_PULSEGEN1_RISING** | Cam B Pulse Generator 1 Rising |
| | **CAM_B_PULSEGEN2_RISING** | Cam B Pulse Generator 2 Rising |
| | **CAM_B_PULSEGEN3_RISING** | Cam B Pulse Generator 3 Rising |
| | **CAM_B_PULSEGEN0_FALLING** | Cam B Pulse Generator 0 Falling |
| | **CAM_B_PULSEGEN1_FALLING** | Cam B Pulse Generator 1 Falling |
| | **CAM_B_PULSEGEN2_FALLING** | Cam B Pulse Generator 2 Falling |
| | **CAM_B_PULSEGEN3_FALLING** | Cam B Pulse Generator 3 Falling |
| | **BYPASS_FRONT_GPI_0_RISING** | Bypass Front GPI 0 Rising |
| | **BYPASS_FRONT_GPI_0_FALLING** | Bypass Front GPI 0 Falling |
| | **BYPASS_FRONT_GPI_1_RISING** | Bypass Front GPI 1 Rising |
| | **BYPASS_FRONT_GPI_1_FALLING** | Bypass Front GPI 1 Falling |
| | **BYPASS_FRONT_GPI_2_RISING** | Bypass Front GPI 2 Rising |
| | **BYPASS_FRONT_GPI_2_FALLING** | Bypass Front GPI 2 Falling |
| | **BYPASS_FRONT_GPI_3_RISING** | Bypass Front GPI 3 Rising |
| | **BYPASS_FRONT_GPI_3_FALLING** | Bypass Front GPI 3 Falling |
| | **PULSEGEN0_RISING** | Pulse Generator 0 Rising |
| | **PULSEGEN1_RISING** | Pulse Generator 1 Rising |
| | **PULSEGEN2_RISING** | Pulse Generator 2 Rising |
| | **PULSEGEN3_RISING** | Pulse Generator 3 Rising |
| | **PULSEGEN0_FALLING** | Pulse Generator 0 Falling |
| | **PULSEGEN1_FALLING** | Pulse Generator 1 Falling |
| | **PULSEGEN2_FALLING** | Pulse Generator 2 Falling |
| | **PULSEGEN3_FALLING** | Pulse Generator 3 Falling |
| Default value | **PULSEGEN0_FALLING** | |

**Example 7.31. Usage of FG_TRIGGERCAMERA_SOURCE_CXP1**

```
int result = 0;
int value = PULSEGEN0_FALLING;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

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

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

### 7.5.11.3. FG_TRIGGERCAMERA_SOURCE_CXP2

**Table 7.33. Parameter properties of FG_TRIGGERCAMERA_SOURCE_CXP2**

| Property | Value | |
|---|---|---|
| Name | **FG_TRIGGERCAMERA_SOURCE_CXP2** | |
| Display Name | **CXP Link Trigger 2 Source** | |
| Type | **Enumeration** | |
| Access policy | **Read/Write/Change** | |
| Storage policy | **Persistent** | |
| Allowed values | **GND** | GND |
| | **CAM_A_PULSEGEN0_RISING** | Cam A Pulse Generator 0 Rising |
| | **CAM_A_PULSEGEN1_RISING** | Cam A Pulse Generator 1 Rising |
| | **CAM_A_PULSEGEN2_RISING** | Cam A Pulse Generator 2 Rising |
| | **CAM_A_PULSEGEN3_RISING** | Cam A Pulse Generator 3 Rising |
| | **CAM_A_PULSEGEN0_FALLING** | Cam A Pulse Generator 0 Falling |
| | **CAM_A_PULSEGEN1_FALLING** | Cam A Pulse Generator 1 Falling |
| | **CAM_A_PULSEGEN2_FALLING** | Cam A Pulse Generator 2 Falling |
| | **CAM_A_PULSEGEN3_FALLING** | Cam A Pulse Generator 3 Falling |
| | **CAM_B_PULSEGEN0_RISING** | Cam B Pulse Generator 0 Rising |
| | **CAM_B_PULSEGEN1_RISING** | Cam B Pulse Generator 1 Rising |
| | **CAM_B_PULSEGEN2_RISING** | Cam B Pulse Generator 2 Rising |
| | **CAM_B_PULSEGEN3_RISING** | Cam B Pulse Generator 3 Rising |
| | **CAM_B_PULSEGEN0_FALLING** | Cam B Pulse Generator 0 Falling |
| | **CAM_B_PULSEGEN1_FALLING** | Cam B Pulse Generator 1 Falling |
| | **CAM_B_PULSEGEN2_FALLING** | Cam B Pulse Generator 2 Falling |
| | **CAM_B_PULSEGEN3_FALLING** | Cam B Pulse Generator 3 Falling |
| | **BYPASS_FRONT_GPI_0_RISING** | Bypass Front GPI 0 Rising |
| | **BYPASS_FRONT_GPI_0_FALLING** | Bypass Front GPI 0 Falling |
| | **BYPASS_FRONT_GPI_1_RISING** | Bypass Front GPI 1 Rising |
| | **BYPASS_FRONT_GPI_1_FALLING** | Bypass Front GPI 1 Falling |
| | **BYPASS_FRONT_GPI_2_RISING** | Bypass Front GPI 2 Rising |
| | **BYPASS_FRONT_GPI_2_FALLING** | Bypass Front GPI 2 Falling |
| | **BYPASS_FRONT_GPI_3_RISING** | Bypass Front GPI 3 Rising |
| | **BYPASS_FRONT_GPI_3_FALLING** | Bypass Front GPI 3 Falling |
| | **PULSEGEN0_RISING** | Pulse Generator 0 Rising |
| | **PULSEGEN1_RISING** | Pulse Generator 1 Rising |
| | **PULSEGEN2_RISING** | Pulse Generator 2 Rising |
| | **PULSEGEN3_RISING** | Pulse Generator 3 Rising |
| | **PULSEGEN0_FALLING** | Pulse Generator 0 Falling |
| | **PULSEGEN1_FALLING** | Pulse Generator 1 Falling |
| | **PULSEGEN2_FALLING** | Pulse Generator 2 Falling |
| | **PULSEGEN3_FALLING** | Pulse Generator 3 Falling |
| Default value | **GND** | |

**Example 7.32. Usage of FG_TRIGGERCAMERA_SOURCE_CXP2**

```
int result = 0;
int value = GND;
```

```
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

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

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

### 7.5.11.4. FG_TRIGGERCAMERA_SOURCE_CXP3

**Table 7.34. Parameter properties of FG_TRIGGERCAMERA_SOURCE_CXP3**

| Property | Value | |
|---|---|---|
| Name | **FG_TRIGGERCAMERA_SOURCE_CXP3** | |
| Display Name | **CXP Link Trigger 3 Source** | |
| Type | **Enumeration** | |
| Access policy | **Read/Write/Change** | |
| Storage policy | **Persistent** | |
| Allowed values | **GND** | GND |
| | **CAM_A_PULSEGEN0_RISING** | Cam A Pulse Generator 0 Rising |
| | **CAM_A_PULSEGEN1_RISING** | Cam A Pulse Generator 1 Rising |
| | **CAM_A_PULSEGEN2_RISING** | Cam A Pulse Generator 2 Rising |
| | **CAM_A_PULSEGEN3_RISING** | Cam A Pulse Generator 3 Rising |
| | **CAM_A_PULSEGEN0_FALLING** | Cam A Pulse Generator 0 Falling |
| | **CAM_A_PULSEGEN1_FALLING** | Cam A Pulse Generator 1 Falling |
| | **CAM_A_PULSEGEN2_FALLING** | Cam A Pulse Generator 2 Falling |
| | **CAM_A_PULSEGEN3_FALLING** | Cam A Pulse Generator 3 Falling |
| | **CAM_B_PULSEGEN0_RISING** | Cam B Pulse Generator 0 Rising |
| | **CAM_B_PULSEGEN1_RISING** | Cam B Pulse Generator 1 Rising |
| | **CAM_B_PULSEGEN2_RISING** | Cam B Pulse Generator 2 Rising |
| | **CAM_B_PULSEGEN3_RISING** | Cam B Pulse Generator 3 Rising |
| | **CAM_B_PULSEGEN0_FALLING** | Cam B Pulse Generator 0 Falling |
| | **CAM_B_PULSEGEN1_FALLING** | Cam B Pulse Generator 1 Falling |
| | **CAM_B_PULSEGEN2_FALLING** | Cam B Pulse Generator 2 Falling |
| | **CAM_B_PULSEGEN3_FALLING** | Cam B Pulse Generator 3 Falling |
| | **BYPASS_FRONT_GPI_0_RISING** | Bypass Front GPI 0 Rising |
| | **BYPASS_FRONT_GPI_0_FALLING** | Bypass Front GPI 0 Falling |
| | **BYPASS_FRONT_GPI_1_RISING** | Bypass Front GPI 1 Rising |
| | **BYPASS_FRONT_GPI_1_FALLING** | Bypass Front GPI 1 Falling |
| | **BYPASS_FRONT_GPI_2_RISING** | Bypass Front GPI 2 Rising |
| | **BYPASS_FRONT_GPI_2_FALLING** | Bypass Front GPI 2 Falling |
| | **BYPASS_FRONT_GPI_3_RISING** | Bypass Front GPI 3 Rising |
| | **BYPASS_FRONT_GPI_3_FALLING** | Bypass Front GPI 3 Falling |
| | **PULSEGEN0_RISING** | Pulse Generator 0 Rising |
| | **PULSEGEN1_RISING** | Pulse Generator 1 Rising |
| | **PULSEGEN2_RISING** | Pulse Generator 2 Rising |
| | **PULSEGEN3_RISING** | Pulse Generator 3 Rising |
| | **PULSEGEN0_FALLING** | Pulse Generator 0 Falling |
| | **PULSEGEN1_FALLING** | Pulse Generator 1 Falling |
| | **PULSEGEN2_FALLING** | Pulse Generator 2 Falling |
| | **PULSEGEN3_FALLING** | Pulse Generator 3 Falling |
| Default value | **GND** | |

**Example 7.33. Usage of FG_TRIGGERCAMERA_SOURCE_CXP3**

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

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

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

## 7.5.12. Digital Output

The CXP-12 Interface Card 4C has two front general purpose outputs (GPOs).

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. As this is a multi camera applet you can choose any of the trigger modules for each output. For example you can select Cam A pulse form generator 0 for all outputs.

- 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_FRONT_GPO_0

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

**Table 7.35. Parameter properties of FG_TRIGGEROUT_SELECT_FRONT_GPO_0**

| Property | Value | |
|---|---|---|
| Name | **FG_TRIGGEROUT_SELECT_FRONT_GPO_0** | |
| Display Name | **Trigger Out Select 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 |
| | **CAM_B_PULSEGEN0** | Cam B Pulse Generator 0 |
| | **CAM_B_PULSEGEN1** | Cam B Pulse Generator 1 |
| | **CAM_B_PULSEGEN2** | Cam B Pulse Generator 2 |
| | **CAM_B_PULSEGEN3** | Cam B Pulse Generator 3 |
| | **CAM_B_NOT_PULSEGEN0** | Not Cam B Pulse Generator 0 |
| | **CAM_B_NOT_PULSEGEN1** | Not Cam B Pulse Generator 1 |
| | **CAM_B_NOT_PULSEGEN2** | Not Cam B Pulse Generator 2 |
| | **CAM_B_NOT_PULSEGEN3** | Not Cam B Pulse Generator 3 |
| | **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 7.34. 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 */
}
```

### 7.5.12.2. FG_TRIGGEROUT_SELECT_FRONT_GPO_1

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

**Table 7.36. Parameter properties of FG_TRIGGEROUT_SELECT_FRONT_GPO_1**

| Property | Value | |
|---|---|---|
| Name | **FG_TRIGGEROUT_SELECT_FRONT_GPO_1** | |
| Display Name | **Trigger Out Select Front GPO 1** | |
| 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 |
| | **CAM_B_PULSEGEN0** | Cam B Pulse Generator 0 |
| | **CAM_B_PULSEGEN1** | Cam B Pulse Generator 1 |
| | **CAM_B_PULSEGEN2** | Cam B Pulse Generator 2 |
| | **CAM_B_PULSEGEN3** | Cam B Pulse Generator 3 |
| | **CAM_B_NOT_PULSEGEN0** | Not Cam B Pulse Generator 0 |
| | **CAM_B_NOT_PULSEGEN1** | Not Cam B Pulse Generator 1 |
| | **CAM_B_NOT_PULSEGEN2** | Not Cam B Pulse Generator 2 |
| | **CAM_B_NOT_PULSEGEN3** | Not Cam B Pulse Generator 3 |
| | **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_B_NOT_PULSEGEN0** | |

**Example 7.35. Usage of FG_TRIGGEROUT_SELECT_FRONT_GPO_1**

```
int result = 0;
int value = CAM_B_NOT_PULSEGEN0;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

if ((result = Fg_setParameterWithType(fg, FG_TRIGGEROUT_SELECT_FRONT_GPO_1, &value, 0, type)) < 0) {
    /* error handling */
}
```

```
if ((result = Fg_getParameterWithType(fg, FG_TRIGGEROUT_SELECT_FRONT_GPO_1, &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 7.37. 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 7.36. 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 7.38. 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 7.37. 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 7.39. Parameter properties of FG_TRIGGEROUT_STATS_SOURCE**

| Property | Value |
|---|---|
| Name | **FG_TRIGGEROUT_STATS_SOURCE** |
| Display Name | **Trigger Out Statistics 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 7.38. 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 7.40. Parameter properties of FG_TRIGGEROUT_STATS_PULSECOUNT**

| Property | Value |
|---|---|
| Name | **FG_TRIGGEROUT_STATS_PULSECOUNT** |
| Display Name | **Trigger Out Statistics 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 7.39. 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 7.41. Parameter properties of FG_TRIGGEROUT_STATS_PULSECOUNT_CLEAR**

| Property | Value |
|---|---|
| Name | **FG_TRIGGEROUT_STATS_PULSECOUNT_CLEAR** |
| Display Name | **Trigger Out Statistics Pulse Count Clear** |
| 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_TRIGGEROUT_STATS_PULSECOUNT_CLEAR**

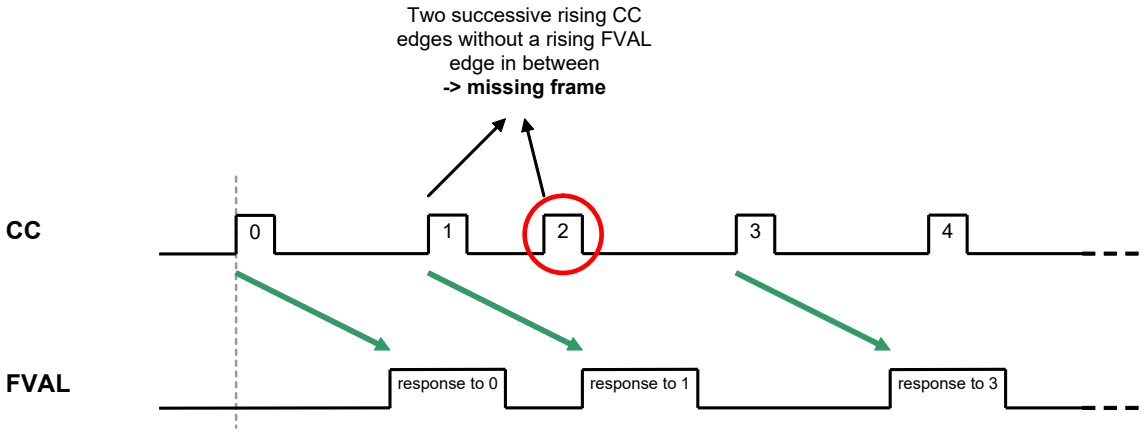
```
int result = 0;
int value = FG_APPLY;
const enum FgParamTypes type = FG_PARAM_TYPE_INT32_T;

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 does 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 sends a frame to the interface card. If a trigger pulse gets lost, or the camera cannot send a frame, this register is set to **FG_YES**. Technically, between two output signal edges, an 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 behavior depends on the camera timing and interface. For very high frame rates the latency of sending an image can get bigger so that the detection mechanism of missing responses is not working. In this case you cannot use this parameter to detect missing triggers. Use the trigger and frame counters instead. You can also use the CXP source tag and trigger acknowledges to detect lost frames or trigger. 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 correctly for all camera models due to different timings.

## Select Camera Control/Trigger Signal Line

Ensure you 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

You must start the acquisition before activating the trigger. Otherwise, the trigger pulses sent will get lost. 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 response and the exact number of missing frames, use event *FG_MISSING_CAM0_FRAME_RESPONSE*. This approach exists for all 2 cameras connected to the applet addressed by its index.

**Table 7.42. 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 7.41. 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 7.43. Parameter properties of FG_MISSING_CAMERA_FRAME_RESPONSE_CLEAR**

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

**Example 7.42. 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 and synchronized 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. Please read the description of the detection and the documentation on how to configure the mechanism in Chapter 7.5.12.3.6.1 carefully. 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* respectively *FG_TRIGGER_OUTPUT_CAM1* with this register. One of the pulse form generators can be selected.

**Table 7.44. 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 7.43. 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 interface card. However, the transfer bandwidth to the PC via DMA channel can vary caused by interrupts, other hardware and the current CPU load. Furthermore, if operated in **selective mode**, it is possible to queue buffer slower than the camera offers new images and therefore generate an overflow condition on the frame grabber. Also, the camera frame rate can vary due to an fluctuating trigger. For these cases, the applet is equipped with a memory to buffer the input frames. The fill level of the buffer can be obtained by reading from parameter *FG_FILLLEVEL*.

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

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

- Corrupted Frames:

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

- Lost Frames:

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

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

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

## 8.1. FG_FILLLEVEL

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

**Table 8.1. Parameter properties of FG_FILLLEVEL**

| Property | Value |
|---|---|
| Name | **FG_FILLLEVEL** |
| Display Name | **Fill Level** |
| Type | **Unsigned Integer** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **Minimum 0**<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. FG_OVERFLOW_OFF_THRESHOLD

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

**Table 8.3. Parameter properties of FG_OVERFLOW_OFF_THRESHOLD**

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

**Example 8.3. Usage of FG_OVERFLOW_OFF_THRESHOLD**

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

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

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

## 8.4. FG_OVERFLOW_ON_THRESHOLD

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

**Table 8.4. Parameter properties of FG_OVERFLOW_ON_THRESHOLD**

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

**Example 8.4. Usage of FG_OVERFLOW_ON_THRESHOLD**

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

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

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

## 8.5. FG_OVERFLOW_ON_SYNC_THRESHOLD

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

**Table 8.5. Parameter properties of FG_OVERFLOW_ON_SYNC_THRESHOLD**

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

**Example 8.5. Usage of FG_OVERFLOW_ON_SYNC_THRESHOLD**

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

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

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

## 8.6. FG_OVERFLOW_EVENT_SELECT

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

**Table 8.6. Event select for *FG_OVERFLOW_CAM0***

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

**Table 8.7. Parameter properties of FG_OVERFLOW_EVENT_SELECT**

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

**Example 8.6. Usage of FG_OVERFLOW_EVENT_SELECT**

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

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

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

## 8.7. Events

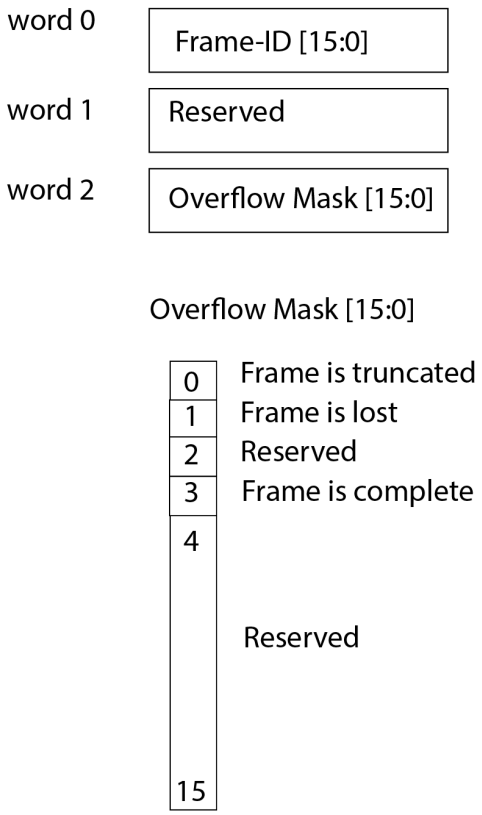
In programming or runtime environments, a callback function is a piece of executable code that is passed as an argument, which is expected to call back (execute) exactly that time an event is triggered. This applet can generate some software callback events based on the memory overflow condition as explained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

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

### 8.7.1. FG_OVERFLOW_CAM0

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

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




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

# Chapter 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. White Balance

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

The applet Acq_DualCXP12Area performs a Bayer de-mosaicing. The white balancing is performed prior to the Bayer de-mosaicing, to ensure the correction of the raw data and avoid subsequent faults during processing.

## 10.1. FG_SCALINGFACTOR_GREEN

**Table 10.1. Parameter properties of FG_SCALINGFACTOR_GREEN**

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

**Example 10.1. Usage of FG_SCALINGFACTOR_GREEN**

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

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

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

## 10.2. FG_SCALINGFACTOR_RED

**Table 10.2. Parameter properties of FG_SCALINGFACTOR_RED**

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

**Example 10.2. Usage of FG_SCALINGFACTOR_RED**

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

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

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

## 10.3. FG_SCALINGFACTOR_BLUE

**Table 10.3. Parameter properties of FG_SCALINGFACTOR_BLUE**

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

**Example 10.3. Usage of FG_SCALINGFACTOR_BLUE**

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

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

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

# Chapter 11. Color Converter

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

This applet can perform the following conversions.

**Table 11.1. Color Conversion**

| Input Format | Mono | RGB | Bayer | YCbCr |
|---|---|---|---|---|
| **Output Format** | | | | |
| Mono | yes | yes | yes | N/A |
| RGB | yes | yes | yes | N/A |
| Bayer | N/A | N/A | yes | N/A |
| YCbCr | N/A | N/A | N/A | yes |

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

# Chapter 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*.

Moreover, the color converter of this applet can convert between different color formats of the input and output. Check Chapter 11, '*Color Converter*' for more information.

This applet supports the following output formats:

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

### 30 Bit Output Format

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

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

### 36 Bit Output Format

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

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

### 42 Bit Output Format

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

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

### BGR vs. RGB Memory Alignement

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

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

### 10 Bit Output Format

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

- **FG_MONO12**: 12 bit grayscale format

### 12 Bit Output Format

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

- **FG_MONO14**: 14 bit grayscale format

### 14 Bit Output Format

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

- **FG_MONO16**: 16 bit grayscale format

### DMA Bandwidth

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

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

### 10 Bit Output Format

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

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

### 12 Bit Output Format

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

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

## 14 Bit Output Format

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

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

## DMA Bandwidth

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

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

**Table 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_MONO8** Mono 8<br>**FG_MONO10** Mono 10p<br>**FG_MONO12** Mono 12p<br>**FG_MONO14** Mono 14p<br>**FG_MONO16** Mono 16<br>**FG_BGR8** BGR 8bit<br>**FG_BGR10** BGR 10bit<br>**FG_BGR12** BGR 12bit<br>**FG_BGR14** BGR 14p<br>**FG_BGR16** BGR 16bit<br>**FG_RGB8** RGB 8<br>**FG_RGB10** RGB 10p<br>**FG_RGB12** RGB 12p<br>**FG_RGB14** RGB 14p<br>**FG_RGB16** RGB 16<br>**FG_BGRA8** BGRA 8<br>**FG_RGBA8** RGBA 8<br>**FG_BAYERGR8** Bayer GR 8<br>**FG_BAYERGR10** Bayer GR 10p<br>**FG_BAYERGR12** Bayer GR 12p<br>**FG_BAYERGR14** Bayer GR 14p<br>**FG_BAYERGR16** Bayer GR 16<br>**FG_BAYERRG8** Bayer RG 8<br>**FG_BAYERRG10** Bayer RG 10p<br>**FG_BAYERRG12** Bayer RG 12p<br>**FG_BAYERRG14** Bayer RG 14p<br>**FG_BAYERRG16** Bayer RG 16<br>**FG_BAYERGB8** Bayer GB 8<br>**FG_BAYERGB10** Bayer GB 10p<br>**FG_BAYERGB12** Bayer GB 12p<br>**FG_BAYERGB14** Bayer GB 14p<br>**FG_BAYERGB16** Bayer GB 16<br>**FG_BAYERBG8** Bayer BG 8<br>**FG_BAYERBG10** Bayer BG 10p<br>**FG_BAYERBG12** Bayer BG 12p<br>**FG_BAYERBG14** Bayer BG 14p<br>**FG_BAYERBG16** Bayer BG 16<br>**FG_YUV422_8** YCbCr422_8 |
| Default value | **FG_MONO8** |

**Example 12.1. Usage of FG_FORMAT**

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

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

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

## 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 | **Bit 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 interface card. A shift of 0 means that the most significant bits (MSB) of the internal processing bit width are mapped to the output MSB. For example, if the applet has an internal processing bit width of 12 bit and you select a 10 bit output, the upper 10 bits are mapped to the output. If you select however a bit width of two, the lower 10 bits are mapped to the output. Note that this applet has an internal bit width of 16 bits.

**Table 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) {
```

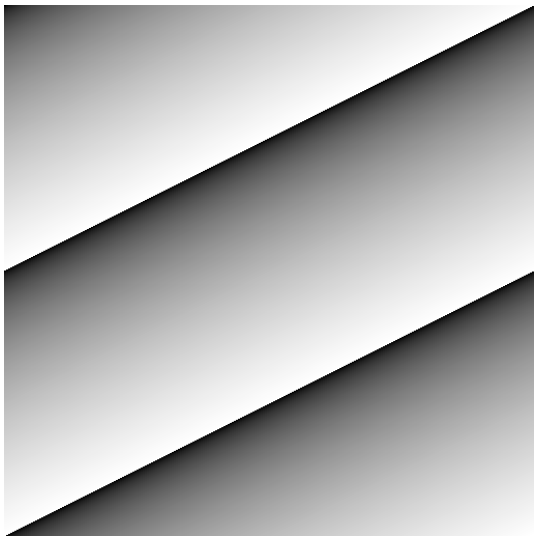
```
    /* error handling */
}
```

# 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.

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*.

### Only 8bit support

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

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

**Table 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 65544**<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 65544**<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. As this applet is capable of using monochrome bayer or RGB inputs.

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

- **FG_HORIZONTAL**

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

- **FG_VERTICAL**

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

- **FG_DIAGONAL**

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

**Table 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** Pixel Frequency<br>**FG_LINERATE** Line Rate<br>**FG_FRAMERATE** 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** 0.9375<br>**Maximum** 3708.0<br>**Stepsize** 1.8749999999999996 |
| 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 1.03E8**<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 1.03E8**<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 uses a CoaxPress camera interface, the CXP trigger output of the respective camera port is used as camera simulator trigger input. The rising edge of the trigger will be used.

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

### Trigger frequency must not exceed the speed of the camera simulator

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

**Table 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 2**<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 2**<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

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

## 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_ID

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

**Table 14.2. 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 14.2. 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 */
}
```

## 14.3. 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 14.3. 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 14.3. 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 */
}
```

## 14.4. 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 14.4. 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 14.4. 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 */
}
```

## 14.5. FG_CAMSTATUS

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

**Table 14.5. 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 14.5. 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 */
}
```

## 14.6. FG_CAMSTATUS_EXTENDED

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

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

**Table 14.6. 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 14.6. 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 */
}
```

## 14.7. FG_SYSTEMMONITOR_FPGA_DNA_LOW

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

**Table 14.7. Parameter properties of FG_SYSTEMMONITOR_FPGA_DNA_LOW**

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

**Example 14.7. Usage of FG_SYSTEMMONITOR_FPGA_DNA_LOW**

```
int result = 0;
uint64_t value = 0;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT64_T;

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

## 14.8. FG_SYSTEMMONITOR_FPGA_DNA_HIGH

The parameter *FG_SYSTEMMONITOR_FPGA_DNA_HIGH* provides the upper 32s bit unique FPGA DNA.

**Table 14.8. Parameter properties of FG_SYSTEMMONITOR_FPGA_DNA_HIGH**

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_FPGA_DNA_HIGH** |
| Display Name | **FPGA DNA High** |
| Type | **Unsigned Integer** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **Minimum 0**<br>**Maximum 4294967295**<br>**Stepsize 1** |

**Example 14.8. Usage of FG_SYSTEMMONITOR_FPGA_DNA_HIGH**

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

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

## 14.9. Version

The category provides version information.

### 14.9.1. FG_APPLET_VERSION

This parameter indicates the version number of the applet. Report this value when contacting the Basler support.

**Table 14.9. Parameter properties of FG_APPLET_VERSION**

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

**Example 14.9. 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 */
}
```

### 14.9.2. FG_APPLET_REVISION

This parameter indicates the revision number of the applet. Report this value when contacting the Basler support.

**Table 14.10. Parameter properties of FG_APPLET_REVISION**

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

**Example 14.10. 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 */
}
```

### 14.9.3. FG_VISUALAPPLETS_BUILD_VERSION

Returns the VisualApplets version used to build the applets.

**Table 14.11. Parameter properties of FG_VISUALAPPLETS_BUILD_VERSION**

| Property | Value |
|---|---|
| Name | **FG_VISUALAPPLETS_BUILD_VERSION** |
| Display Name | **VisualApplets Build Version** |
| Type | **String** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 14.11. Usage of FG_VISUALAPPLETS_BUILD_VERSION**

```
int result = 0;
char* value = n/a;
const enum FgParamTypes type = FG_PARAM_TYPE_CHAR_PTR;

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

## 14.10. Legacy

This category includes the legacy parameter for user software compatibility. The parameter of this category shouldn't be used anymore.

### 14.10.1. FG_CXP_TRIGGER_PACKET_MODE

This parameter is for legacy use only and shouldn't be used anymore.

Setting the parameter to the CXP standard triggers re-writing the *FG_TRIGGERCAMERA_OUT_SELECT* legacy parameter.

However, if you set the parameter to rising edge only, the legacy compatibility mode applies. In this case, *FG_TRIGGERCAMERA_SOURCE_CXP1*, *FG_TRIGGERCAMERA_SOURCE_CXP2* and *FG_TRIGGERCAMERA_SOURCE_CXP3* are set to **GND**.

Be careful with this parameter as it overwrites the *FG_TRIGGERCAMERA_SOURCE_CXP0* and *FG_TRIGGERCAMERA_SOURCE_CXP0* parameters.

This legacy parameter can't be used in configuration files anymore.

**Table 14.12. Parameter properties of FG_CXP_TRIGGER_PACKET_MODE**

| Property | Value | |
|---|---|---|
| Name | **FG_CXP_TRIGGER_PACKET_MODE** | |
| Display Name | **CXP Trigger Packet Mode** | |
| Type | **Enumeration** | |
| Access policy | **Read/Write/Change** | |
| Storage policy | **Persistent** | |
| Allowed values | **FG_STANDARD**<br>**FG_RISING_EDGE_ONLY** | CXP Trigger Standard<br>CXP Trigger Rising |
| Default value | **FG_STANDARD** | |

**Example 14.12. Usage of FG_CXP_TRIGGER_PACKET_MODE**

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

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

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

## 14.10.2. FG_TRIGGERCAMERA_OUT_SELECT

This is a legacy parameter. It is replaced by the parameters *FG_TRIGGERCAMERA_SOURCE_CXP0* and *FG_TRIGGERCAMERA_SOURCE_CXP1*. Before, the legacy parameter controlled the generation for CXP LinkTrigger0 associated with the start of a pulse and CXP LinkTrigger1 associated with the end of a pulse. To keep the compatibility, when writing to this parameter, the value is copied to *FG_TRIGGERCAMERA_SOURCE_CXP0* and furthermore sets the inverted value to *FG_TRIGGERCAMERA_SOURCE_CXP1*.

Reading the value only represents the status of *FG_TRIGGERCAMERA_SOURCE_CXP0* and can be ambiguous as the new parameters offer more possibilities which can't be represented with the legacy parameter.

This legacy parameter can't be used in configuration files anymore.

**Table 14.13. Parameter properties of FG_TRIGGERCAMERA_OUT_SELECT**

| Property | Value | |
|---|---|---|
| Name | **FG_TRIGGERCAMERA_OUT_SELECT** | |
| Display Name | **Legacy Trigger Camera Out Select** | |
| Type | **Enumeration** | |
| Access policy | **Read/Write/Change** | |
| Storage policy | **Transient** | |
| 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 |
| | **CAM_B_PULSEGEN0** | Cam B Pulse Generator 0 |
| | **CAM_B_PULSEGEN1** | Cam B Pulse Generator 1 |
| | **CAM_B_PULSEGEN2** | Cam B Pulse Generator 2 |
| | **CAM_B_PULSEGEN3** | Cam B Pulse Generator 3 |
| | **CAM_B_NOT_PULSEGEN0** | Not Cam B Pulse Generator 0 |
| | **CAM_B_NOT_PULSEGEN1** | Not Cam B Pulse Generator 1 |
| | **CAM_B_NOT_PULSEGEN2** | Not Cam B Pulse Generator 2 |
| | **CAM_B_NOT_PULSEGEN3** | Not Cam B Pulse Generator 3 |
| | **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 | **PULSEGEN0** | |

**Example 14.13. Usage of FG_TRIGGERCAMERA_OUT_SELECT**

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

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

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

## 14.11. Debug

### 14.11.1. FG_DEBUGSOURCE

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

**Table 14.14. 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 1**<br>**Stepsize 1** |
| Default value | **0** |

**Example 14.14. 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 */
}
```

### 14.11.2. FG_DEBUGSOURCENAME

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

**Table 14.15. 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 14.15. 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 */
}
```

### 14.11.3. FG_DEBUGSAVECONFIG

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

**Table 14.16. 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 14.16. 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 */
}
```

## 14.11.4. FG_DEBUG_SLOWMODE

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

**Table 14.17. Parameter properties of FG_DEBUG_SLOWMODE**

| Property | Value |
|---|---|
| Name | **FG_DEBUG_SLOWMODE** |
| Display Name | **Debug Slow Output** |
| Type | **Enumeration** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **FG_SLOW_OFF** Off<br>**FG_SLOW_SOFTWARE** Software<br>**FG_SLOW_PWM** PWM |
| Default value | **FG_SLOW_OFF** |

**Example 14.17. Usage of FG_DEBUG_SLOWMODE**

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

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

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

## 14.11.5. FG_DEBUG_SOFTWRAE_SLOWGATE

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

**Table 14.18. Parameter properties of FG_DEBUG_SOFTWRAE_SLOWGATE**

| Property | Value |
|---|---|
| Name | **FG_DEBUG_SOFTWRAE_SLOWGATE** |
| Display Name | **Debug Slow Software** |
| Type | **Enumeration** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **FG_ON** On<br>**FG_OFF** Off<br>**FG_PULSE** Pulse |
| Default value | **FG_OFF** |

**Example 14.18. Usage of FG_DEBUG_SOFTWRAE_SLOWGATE**

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

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

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

## 14.11.6. FG_DEBUG_PWM_SLOWRATE

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

**Table 14.19. Parameter properties of FG_DEBUG_PWM_SLOWRATE**

| Property | Value |
|---|---|
| Name | **FG_DEBUG_PWM_SLOWRATE** |
| Display Name | **Slowrate PWM** |
| Type | **Unsigned Integer (64 Bit)** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Transient** |
| Allowed values | **Minimum** 0<br>**Maximum** 144115188075855872<br>**Stepsize** 1 |
| Default value | **10000000** |

**Example 14.19. Usage of FG_DEBUG_PWM_SLOWRATE**

```
int result = 0;
uint64_t value = 10000000;
const enum FgParamTypes type = FG_PARAM_TYPE_UINT64_T;

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

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

```
}
```

## 14.11.7. FG_DEBUG_VERSION

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

**Table 14.20. 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 14.20. 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 */
}
```

## 14.11.8. FG_DEBUG_FRAMEID_TO_FIRSTPIXEL

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

**Table 14.21. Parameter properties of FG_DEBUG_FRAMEID_TO_FIRSTPIXEL**

| Property | Value |
|---|---|
| Name | **FG_DEBUG_FRAMEID_TO_FIRSTPIXEL** |
| Display Name | **FrameID mapped** |
| Type | **Enumeration** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Transient** |
| Allowed values | **FG_ON** On<br>**FG_OFF** Off |
| Default value | **FG_OFF** |

**Example 14.21. Usage of FG_DEBUG_FRAMEID_TO_FIRSTPIXEL**

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

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

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

## 14.11.9. Input

### 14.11.9.1. FG_DEBUGINENABLE

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

**Table 14.22. 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 14.22. 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 */
}
```

### 14.11.9.2. FG_DEBUGFILE

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

**Table 14.23. 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 14.23. 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 */
}
```

### 14.11.9.3. FG_DEBUGINSERT

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

**Table 14.24. 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 14.24. 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 */
}
```

## 14.11.9.4. FG_DEBUGWRITEPIXEL

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

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

### 14.11.9.5. FG_DEBUGWRITEFLAG

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

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

### 14.11.9.6. FG_DEBUGREADY

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

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

### 14.11.9.7. FG_DEBUG_FORCE_FRAMEID

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

**Table 14.28. Parameter properties of FG_DEBUG_FORCE_FRAMEID**

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

**Example 14.28. Usage of FG_DEBUG_FORCE_FRAMEID**

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

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

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

## 14.11.9.8. FG_DEBUG_FRAMEID

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

**Table 14.29. Parameter properties of FG_DEBUG_FRAMEID**

| Property | Value |
|---|---|
| Name | **FG_DEBUG_FRAMEID** |
| Display Name | **Debug FrameID** |
| Type | **Unsigned Integer** |
| Access policy | **Read/Write/Change** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum** 0<br>**Maximum** 65535<br>**Stepsize** 1 |
| Default value | **0** |

**Example 14.29. Usage of FG_DEBUG_FRAMEID**

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

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

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

## 14.11.10. Output

### 14.11.10.1. FG_DEBUGOUTENABLE

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

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

### 14.11.10.2. FG_DEBUGOUTXPOS

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

**Table 14.31. 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** 48<br>**Maximum** 32768<br>**Stepsize** 1 |
| Unit of measure | **pixel** |

**Example 14.31. 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 */
}
```

### 14.11.10.3. FG_DEBUGOUTYPOS

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

**Table 14.32. 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 2**<br>**Maximum 65536**<br>**Stepsize 1** |
| Unit of measure | **pixel** |

**Example 14.32. 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 */
}
```

### 14.11.10.4. FG_DEBUGOUTPIXEL

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

**Table 14.33. 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 14.33. 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 */
```

```
}
```

## 14.12. GenTL

### 14.12.1. FG_GENTL_INFO_VERSION

This paramter gives the version of the GenTL description used by our GenTL producer. This parameter is of internal use only.

**Table 14.34. Parameter properties of FG_GENTL_INFO_VERSION**

| Property | Value |
|---|---|
| Name | **FG_GENTL_INFO_VERSION** |
| Display Name | **Version** |
| Type | **String** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Unit of measure | |

**Example 14.34. Usage of FG_GENTL_INFO_VERSION**

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

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

### 14.12.2. FG_GENTL_INFO_IGNOREFGFORMAT

This paramter describs the handling of outputformats in the GenTL producer. If the parameter is set to 1 the handling of Output formats is done by the producer ignoring the Outputformatsettings of the Applet.

**Table 14.35. Parameter properties of FG_GENTL_INFO_IGNOREFGFORMAT**

| Property | Value |
|---|---|
| Name | **FG_GENTL_INFO_IGNOREFGFORMAT** |
| Display Name | **IgnoreFGFormat** |
| Type | **String** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 14.35. Usage of FG_GENTL_INFO_IGNOREFGFORMAT**

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

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

### 14.12.3. FG_GENTL_INFO_OVERFLOWCAPABLE

This parameter informs the producer that the Applet is capable of extended overflow management.

**Table 14.36. Parameter properties of FG_GENTL_INFO_OVERFLOWCAPABLE**

| Property | Value |
|---|---|
| Name | **FG_GENTL_INFO_OVERFLOWCAPABLE** |
| Display Name | **OverflowCapable** |
| Type | **String** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 14.36. Usage of FG_GENTL_INFO_OVERFLOWCAPABLE**

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

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

# Chapter 15. Boardstatus

This category gives information about the current framegrabber board status. For example, the number of used PCIe lanes, or the mapping of the physical and logical CXP ports. For imaWorx and imaFLex, it also shows if a trigger board is connected.

## 15.1. FG_SYSTEMMONITOR_CHANNEL_CURRENT

Returns the power consumption of the CXP channel (PoCXP) in Ampere.

**Table 15.1. Parameter properties of FG_SYSTEMMONITOR_CHANNEL_CURRENT**

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_CHANNEL_CURRENT** |
| Display Name | **Systemmonitor Channel Current** |
| Type | **Double Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Unit of measure | **A** |

**Example 15.1. Usage of FG_SYSTEMMONITOR_CHANNEL_CURRENT**

```
int result = 0;

FieldParameterDouble access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMDOUBLE;

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

## 15.2. FG_SYSTEMMONITOR_CHANNEL_VOLTAGE

Returns the voltage of the CXP channel (PoCXP).

**Table 15.2. Parameter properties of FG_SYSTEMMONITOR_CHANNEL_VOLTAGE**

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_CHANNEL_VOLTAGE** |
| Display Name | **Systemmonitor Channel Voltage** |
| Type | **Double Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Unit of measure | **V** |

**Example 15.2. Usage of FG_SYSTEMMONITOR_CHANNEL_VOLTAGE**

```
int result = 0;
```

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

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

## 15.3. FG_SYSTEMMONITOR_MAPPED_TO_FG_PORT

Indicates the frame grabber port mapping. Range: between 0 and 3.

**Table 15.3. Parameter properties of FG_SYSTEMMONITOR_MAPPED_TO_FG_PORT**

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_MAPPED_TO_FG_PORT** |
| Display Name | **Systemmonitor Mapped to Fg Port** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 15.3. Usage of FG_SYSTEMMONITOR_MAPPED_TO_FG_PORT**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

## 15.4. FG_DMASTATUS

Returns the status of the DMA transmission, i.e. the acquisition state. 0 = stopped DMA, 1 = started DMA.

**Table 15.4. Parameter properties of FG_DMASTATUS**

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

**Example 15.4. 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 */
```

```
}
```

## 15.5. FG_SYSTEMMONITOR_FPGA_TEMPERATURE

Returns the current FGPA temperature.

**Table 15.5. Parameter properties of FG_SYSTEMMONITOR_FPGA_TEMPERATURE**

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_FPGA_TEMPERATURE** |
| Display Name | **Systemmonitor 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 15.5. 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 */
}
```

## 15.6. FG_SYSTEMMONITOR_FPGA_VCC_INT

Returns the internal voltage of the FPGA.

**Table 15.6. Parameter properties of FG_SYSTEMMONITOR_FPGA_VCC_INT**

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_FPGA_VCC_INT** |
| Display Name | **Systemmonitor 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 15.6. 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 */
```

```
}
```

## 15.7. FG_SYSTEMMONITOR_FPGA_VCC_AUX

Returns the VCC auxiliary voltage of the FPGA.

**Table 15.7. 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 15.7. 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 */
}
```

## 15.8. FG_SYSTEMMONITOR_FPGA_VCC_BRAM

Returns the VCC of the BlockRAM voltage of the FPGA.

**Table 15.8. Parameter properties of FG_SYSTEMMONITOR_FPGA_VCC_BRAM**

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_FPGA_VCC_BRAM** |
| Display Name | **Systemmonitor 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 15.8. 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 */
}
```

## 15.9. FG_SYSTEMMONITOR_CURRENT_LINK_WIDTH

Returns the current link width of the frame grabber representing the number of PCIe lanes that are 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 15.9. 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 15.9. 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 */
}
```

## 15.10. FG_SYSTEMMONITOR_CURRENT_LINK_SPEED

Returns the current link width of the frame grabber representing the number of PCIe lanes that are 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 15.10. Parameter properties of FG_SYSTEMMONITOR_CURRENT_LINK_SPEED**

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_CURRENT_LINK_SPEED** |
| Display Name | **Systemmonitor 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 15.10. 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 */
```

```
}
```

## 15.11. FG_SYSTEMMONITOR_PCIE_TRAINED_PAYLOAD_SIZE

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

**Table 15.11. Parameter properties of FG_SYSTEMMONITOR_PCIE_TRAINED_PAYLOAD_SIZE**

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_PCIE_TRAINED_PAYLOAD_SIZE** |
| Display Name | **Systemmonitor 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 15.11. 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 */
}
```

## 15.12. FG_SYSTEMMONITOR_PCIE_TRAINED_REQUEST_SIZE

Returns the size (in bytes) of the PCIe packets payload that are used for the data transmission between the frame grabber and the PCIe bridge.

**Table 15.12. Parameter properties of FG_SYSTEMMONITOR_PCIE_TRAINED_REQUEST_SIZE**

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

**Example 15.12. Usage of FG_SYSTEMMONITOR_PCIE_TRAINED_REQUEST_SIZE**

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

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

```
}
```

## 15.13. FG_SYSTEMMONITOR_EXTERNAL_POWER

Indicates whether the external power connector is connected.

**Table 15.13. Parameter properties of FG_SYSTEMMONITOR_EXTERNAL_POWER**

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_EXTERNAL_POWER** |
| Display Name | **Systemmonitor External Power** |
| Type | **Enumeration** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **FG_GOOD** Good<br>**FG_NO_POWER** No Power |

**Example 15.13. Usage of FG_SYSTEMMONITOR_EXTERNAL_POWER**

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

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

## 15.14. FG_CXP_INPUT_MAPPED_FW_PORT_PORT

This parameter returns the firmware CXP channel, which is currently monitored by the module. There is not necessarily a one-by-one mapping between firmware port (i.e. the camera port resource) and frame grabber port (i.e. the physical connector). Instead, the mapping can be any permutation. The software discovery process reorders the channels and ports to achieve correct virtual interconnect. Range: 0 to 3 (2 bit).

**Table 15.14. Parameter properties of FG_CXP_INPUT_MAPPED_FW_PORT_PORT**

| Property | Value |
|---|---|
| Name | **FG_CXP_INPUT_MAPPED_FW_PORT_PORT** |
| Display Name | **CXP Input Mapped to Firmware Port Port** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 15.14. Usage of FG_CXP_INPUT_MAPPED_FW_PORT_PORT**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

# Chapter 16. Errors

This category gives information about the current error status. It shows error counters for different error types, such as packet errors, missing connection, undefined data or overtriggering. Additionally, it reports warning type errors, like the number of both corrected and uncorrected packets.

## 16.1. FG_SYSTEMMONITOR_DECODER_8B_10B_ERROR

Link stability counter. It is incremented when the number of measured symbols received by the channel transceiver are not in 8b10b encoding or/and have wrong disparity. Range: 0 to (2^48 - 1) (48 bit).

**Table 16.1. Parameter properties of FG_SYSTEMMONITOR_DECODER_8B_10B_ERROR**

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_DECODER_8B_10B_ERROR** |
| Display Name | **Systemmonitor Decoder 8b10b Error** |
| Type | **Unsigned Integer Field (64 Bit)** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.1. Usage of FG_SYSTEMMONITOR_DECODER_8B_10B_ERROR**

```
int result = 0;

FieldParameterAccess access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMACCESS;

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

## 16.2. FG_SYSTEMMONITOR_BYTE_ALIGNMENT_8B_10B_LOCKED

Monitors whether the clock recovery has worked and valid 8b/10b signals are recognized.

**Table 16.2. Parameter properties of FG_SYSTEMMONITOR_BYTE_ALIGNMENT_8B_10B_LOCKED**

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_BYTE_ALIGNMENT_8B_10B_LOCKED** |
| Display Name | **Systemmonitor Byte Alignment 8B 10 B Locked** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.2. Usage of FG_SYSTEMMONITOR_BYTE_ALIGNMENT_8B_10B_LOCKED**

```
int result = 0;

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

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

## 16.3. FG_SYSTEMMONITOR_RX_STREAM_INCOMPLETE_COUNT

Returns the number of received incomplete stream counts. Range: between 0 and 8191 in steps of 1.

**Table 16.3. Parameter properties of FG_SYSTEMMONITOR_RX_STREAM_INCOMPLETE_COUNT**

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_RX_STREAM_INCOMPLETE_COUNT** |
| Display Name | **Systemmonitor Rx Stream Incomplete Count** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.3. Usage of FG_SYSTEMMONITOR_RX_STREAM_INCOMPLETE_COUNT**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

## 16.4. FG_SYSTEMMONITOR_RX_UNKNOWN_DATA_RECEIVED_COUNT

Returns the number of received unknown data, i.e. packets received that aren't defined in the CXP standard. Range: between 0 and 8191 in steps of 1.

**Table 16.4. Parameter properties of FG_SYSTEMMONITOR_RX_UNKNOWN_DATA_RECEIVED_COUNT**

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_RX_UNKNOWN_DATA_RECEIVED_COUNT** |
| Display Name | **Systemmonitor Rx Unknown Data Received Count** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.4. Usage of FG_SYSTEMMONITOR_RX_UNKNOWN_DATA_RECEIVED_COUNT**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

```
}
```

## 16.5. FG_CXP_OVERTRIGGER_REQUEST_PULSECOUNT

This parameter counts the trigger requests that were skipped, because the transmitter was still busy by sending the previous trigger packet. See CXP 2.0 standard, chapter 9.3.2. Bits [11:0] count the amount of violations. Bit [12] is set when a counter overflow occurs. Range: 0 to 4095 (12 bit). Bit 12 indicates an overflow.

**Table 16.5. Parameter properties of FG_CXP_OVERTRIGGER_REQUEST_PULSECOUNT**

| Property | Value |
|---|---|
| Name | **FG_CXP_OVERTRIGGER_REQUEST_PULSECOUNT** |
| Display Name | **CXP Overtrigger Request Pulse Count** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.5. Usage of FG_CXP_OVERTRIGGER_REQUEST_PULSECOUNT**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

## 16.6. FG_CXP_TRIGGER_ACK_MISSING_COUNT

This parameter counts the situations in which a trigger packet was sent, but no acknowledgment packet was received for it yet, which then led to a timeout (480ns for 1-6Gb/s, 240ns for 10-12.5Gb/s). See CXP 2.0 standard, chapter 9.3.2. Bits [11:0] count the amount of violations. Bit [12] is set when a counter overflow occurs. Range: 0 to 8191 (13 bit).

**Table 16.6. Parameter properties of FG_CXP_TRIGGER_ACK_MISSING_COUNT**

| Property | Value |
|---|---|
| Name | **FG_CXP_TRIGGER_ACK_MISSING_COUNT** |
| Display Name | **CXP Lost Trigger ACK Missing Count** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.6. Usage of FG_CXP_TRIGGER_ACK_MISSING_COUNT**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

    if ((result = Fg_getParameterWithType(fg, FG_CXP_TRIGGER_ACK_MISSING_COUNT, &access, 0, type)) < 0) {
```

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

## 16.7. FG_CXP_CONTROL_ACK_LOST_COUNT

This parameter counts situations in which a control packet was sent but no acknowledgment packet was received for it yet and the timeout of 200 ms is reached. See CXP 2.0 standard, chapter 9.6.1.1. Bits [11:0] count the amount of violations. Bit [12] is set when a counter overflow occurs. Range 0 to 8191 (13 bit).

**Table 16.7. Parameter properties of FG_CXP_CONTROL_ACK_LOST_COUNT**

| Property | Value |
|---|---|
| Name | **FG_CXP_CONTROL_ACK_LOST_COUNT** |
| Display Name | **CXP Control ACK Lost Count** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.7. Usage of FG_CXP_CONTROL_ACK_LOST_COUNT**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

## 16.8. FG_CXP_CONTROL_TAG_ERROR_COUNT

This parameter counts situations in which an acknowledgment for a control packet was received with a tag that doesn't match the expected tag sent in the corresponding request control packet. See CXP 2.0 standard, chapter 9.6.1.2. Bits [11:0] count the amount of violations. Bit [12] is set when a counter overflow occurs. Range 0 to 8191 (13 bit).

**Table 16.8. Parameter properties of FG_CXP_CONTROL_TAG_ERROR_COUNT**

| Property | Value |
|---|---|
| Name | **FG_CXP_CONTROL_TAG_ERROR_COUNT** |
| Display Name | **CXP Control Tag Error Count** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.8. Usage of FG_CXP_CONTROL_TAG_ERROR_COUNT**

```
int result = 0;

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

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

## 16.9. FG_CXP_CONTROL_ACK_INCOMPLETE_COUNT

This parameter counts situations in which an incorrectly formatted acknowledgment for a control packet was received. Incorrectly formatted means that e.g. the end of packet indicator is missing etc. Bits [11:0] count the amount of violations. Bit [12] is set when a counter overflow occurs. Range 0 to 8191 (13 bit).

**Table 16.9. Parameter properties of FG_CXP_CONTROL_ACK_INCOMPLETE_COUNT**

| Property | Value |
|---|---|
| Name | **FG_CXP_CONTROL_ACK_INCOMPLETE_COUNT** |
| Display Name | **CXP Control ACK Incomplete Count** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.9. Usage of FG_CXP_CONTROL_ACK_INCOMPLETE_COUNT**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

## 16.10. FG_CXP_HEARTBEAT_INCOMPLETE_COUNT

This parameter counts situations in which the received heart beat packet is incomplete, e.g. it misses the end of the packet indicator. Bits [11:0] count the amount of violations. Bit [12] is set when a counter overflow occurs. Range 0 to 8191 (13 bit).

**Table 16.10. Parameter properties of FG_CXP_HEARTBEAT_INCOMPLETE_COUNT**

| Property | Value |
|---|---|
| Name | **FG_CXP_HEARTBEAT_INCOMPLETE_COUNT** |
| Display Name | **CXP Heartbeat Incomplete Count** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.10. Usage of FG_CXP_HEARTBEAT_INCOMPLETE_COUNT**

```
int result = 0;

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

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

## 16.11. FG_CXP_HEARTBEAT_MAX_PERIOD_VIOLATION_COUNT

The heartbeat period is defined in CXP 2.0 standard as 100ms maximum, i.e. within that time at least 1 heartbeat packet must be sent by the camera. This parameter counts the situations in which heartbeat packets exceeded this timeout (100ms). Bits [11:0] count the amount of violations. Bit [12] is set when a counter overflow occurs. Range 0 to 8191 (13 bit).

**Table 16.11. Parameter properties of FG_CXP_HEARTBEAT_MAX_PERIOD_VIOLATION_COUNT**

| Property | Value |
|---|---|
| Name | **FG_CXP_HEARTBEAT_MAX_PERIOD_VIOLATION_COUNT** |
| Display Name | **CXP Hearbeat Max Period Violation Count** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.11. Usage of FG_CXP_HEARTBEAT_MAX_PERIOD_VIOLATION_COUNT**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

## 16.12. FG_PACKET_TAG_ERROR_COUNT

The parameter counts the number of lost CXP stream packets.

**Table 16.12. Parameter properties of FG_PACKET_TAG_ERROR_COUNT**

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

**Example 16.12. Usage of FG_PACKET_TAG_ERROR_COUNT**

```
int result = 0;
unsigned int value = 0;
```

```
const enum FgParamTypes type = FG_PARAM_TYPE_UINT32_T;

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

## 16.13. FG_SYSTEMMONITOR_PACKETBUFFER_OVERFLOW_COUNT

This parameter counts the number of overflows that occur due to not correctly aligned package orders.

**Table 16.13. Parameter properties of FG_SYSTEMMONITOR_PACKETBUFFER_OVERFLOW_COUNT**

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_PACKETBUFFER_OVERFLOW_COUNT** |
| Display Name | **Systemmonitor Packetbuffer Overflow Count** |
| Type | **Unsigned Integer** |
| Access policy | **Read-Only** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum 0**<br>**Maximum 4095**<br>**Stepsize 1** |

**Example 16.13. Usage of FG_SYSTEMMONITOR_PACKETBUFFER_OVERFLOW_COUNT**

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

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

## 16.14. FG_SYSTEMMONITOR_PACKETBUFFER_OVERFLOW_SOURCE

This parameter returns the port that has overflows due to not correctly aligned package order.

**Table 16.14. Parameter properties of FG_SYSTEMMONITOR_PACKETBUFFER_OVERFLOW_SOURCE**

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_PACKETBUFFER_OVERFLOW_SOURCE** |
| Display Name | **Systemmonitor Packetbuffer Overflow Source** |
| Type | **Unsigned Integer** |
| Access policy | **Read-Only** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum 0**<br>**Maximum 15**<br>**Stepsize 1** |

**Example 16.14. Usage of FG_SYSTEMMONITOR_PACKETBUFFER_OVERFLOW_SOURCE**

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

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

```
}
```

## 16.15. FG_CXP_IMAGETAG_ERROR_COUNT

This parameter returns the number of image tag errors (jumps) in the CXP headers.

**Table 16.15. Parameter properties of FG_CXP_IMAGETAG_ERROR_COUNT**

| Property | Value |
|---|---|
| Name | **FG_CXP_IMAGETAG_ERROR_COUNT** |
| Display Name | **CXP Image Tag Error Count** |
| Type | **Unsigned Integer** |
| Access policy | **Read-Only** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum 0**<br>**Maximum 8191**<br>**Stepsize 1** |

**Example 16.15. Usage of FG_CXP_IMAGETAG_ERROR_COUNT**

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

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

## 16.16. FG_CXP_STREAMID_ERROR_COUNT

The parameter counts how often the received stream ID value in the stream packets mismatches the stream ID value specified in the image header. The parameter is 13 bit wide, where the bits [11:0] represent the actual counter value and the bit [12] stands for the counter overflow. When the overflow bit is set, the counter value shall be treated as don't care. Range: 0 to 8191 (13 bit).

**Table 16.16. Parameter properties of FG_CXP_STREAMID_ERROR_COUNT**

| Property | Value |
|---|---|
| Name | **FG_CXP_STREAMID_ERROR_COUNT** |
| Display Name | **CXP Stream ID Error Count** |
| Type | **Unsigned Integer** |
| Access policy | **Read-Only** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum 0**<br>**Maximum 8191**<br>**Stepsize 1** |

**Example 16.16. Usage of FG_CXP_STREAMID_ERROR_COUNT**

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

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

```
}
```

## 16.17. FG_CXP_CAMERA_MARKER_ERROR_COUNT

This parameter counts how often the sequence of the CXP stream marker and the header or the line markers were incorrect. The parameter is 13 bit wide, where the bits [11:0] represent the actual counter value and the bit [12] stands for the counter overflow. When the overflow bit is set, the counter value shall be treated as don't care. Range: 0 to 8192 (13 bit).

**Table 16.17. Parameter properties of FG_CXP_CAMERA_MARKER_ERROR_COUNT**

| Property | Value |
|---|---|
| Name | **FG_CXP_CAMERA_MARKER_ERROR_COUNT** |
| Display Name | **CXP Camera Marker Error Count** |
| Type | **Unsigned Integer** |
| Access policy | **Read-Only** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum 0**<br>**Maximum 8191**<br>**Stepsize 1** |

**Example 16.17. Usage of FG_CXP_CAMERA_MARKER_ERROR_COUNT**

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

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

## 16.18. FG_CXP_CAMERA_UNEXPECTED_STARTUP_DATA

This parameter detects the error situation in which the first data value after the operator reset was unexpected, i.e. no image header has been received. This situation can happen due to a buggy implementation of the camera, frame grabber firmware or wrong software control of the discovery procedure. Also, a hardware defect of the camera could theoretically cause such a situation. Range: NO or YES.

**Table 16.18. Parameter properties of FG_CXP_CAMERA_UNEXPECTED_STARTUP_DATA**

| Property | Value |
|---|---|
| Name | **FG_CXP_CAMERA_UNEXPECTED_STARTUP_DATA** |
| Display Name | **CXP Camera Unexpected Startup Data Status** |
| Type | **Enumeration** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |
| Allowed values | **FG_YES** Yes<br>**FG_NO** No |

**Example 16.18. Usage of FG_CXP_CAMERA_UNEXPECTED_STARTUP_DATA**

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

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

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

## 16.19. FG_CXP_CAMERA_FRAME_LOST_COUNT

This parameter counts the frames that were lost during acquisition and aren't sent into the applet image pipeline. Frames are lost when an error in the image header is detected or when a frame overlaps with another frame. The parameter is 25 bit wide where the bits [23:0] represent the actual counter value and bit [24] stands for the counter overflow. When the overflow bit is set, the counter value shall be treated as don't care. Range 0 to 33554431 (25 bit).

**Table 16.19. Parameter properties of FG_CXP_CAMERA_FRAME_LOST_COUNT**

| Property | Value |
|---|---|
| Name | **FG_CXP_CAMERA_FRAME_LOST_COUNT** |
| Display Name | **CXP Camera Frame Lost Count** |
| Type | **Unsigned Integer** |
| Access policy | **Read-Only** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum 0**<br>**Maximum 33554431**<br>**Stepsize 1** |

**Example 16.19. Usage of FG_CXP_CAMERA_FRAME_LOST_COUNT**

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

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

## 16.20. FG_CXP_CAMERA_FRAME_CORRUPT_COUNT

This parameter counts the corrupted frames during acquisition. Corrupted frames are frames with error pixels which are sent to the applet image pipeline. The parameter is 25 bit wide where the bits [23:0] represent the actual counter value and bit [24] stands for the counter overflow. When the overflow bit is set, the counter value shall be treated as don't care. Range 0 to 33554431 (25 bit).

**Table 16.20. Parameter properties of FG_CXP_CAMERA_FRAME_CORRUPT_COUNT**

| Property | Value |
|---|---|
| Name | **FG_CXP_CAMERA_FRAME_CORRUPT_COUNT** |
| Display Name | **CXP Camera Frame Corrupt Count** |
| Type | **Unsigned Integer** |
| Access policy | **Read-Only** |
| Storage policy | **Persistent** |
| Allowed values | **Minimum 0**<br>**Maximum 33554431**<br>**Stepsize 1** |

**Example 16.20. Usage of FG_CXP_CAMERA_FRAME_CORRUPT_COUNT**

```
int result = 0;
```

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

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

## 16.21. CRC

This category gives information about packet CRC errors detected for stream packets and control packets.

### 16.21.1. FG_SYSTEMMONITOR_RX_PACKET_CRC_ERROR_COUNT

Returns the number of received packet CRC errors. Range: between 0 and 8191 in steps of 1.

**Table 16.21. Parameter properties of FG_SYSTEMMONITOR_RX_PACKET_CRC_ERROR_COUNT**

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_RX_PACKET_CRC_ERROR_COUNT** |
| Display Name | **Systemmonitor Rx Packet CRC Error Count** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.21. Usage of FG_SYSTEMMONITOR_RX_PACKET_CRC_ERROR_COUNT**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

### 16.21.2. FG_CXP_STREAMPACKET_CRC_ERROR

This parameter returns information whether there were CRC errors in received stream packets. Range 0 (NO) to 1 (YES).

**Table 16.22. Parameter properties of FG_CXP_STREAMPACKET_CRC_ERROR**

| Property | Value |
|---|---|
| Name | **FG_CXP_STREAMPACKET_CRC_ERROR** |
| Display Name | **CXP Stream Packet CRC Error** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.22. Usage of FG_CXP_STREAMPACKET_CRC_ERROR**

```
int result = 0;

FieldParameterInt access;
```

```
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

### 16.21.3. FG_CXP_CONTROL_ACK_PACKET_CRC_ERROR

This parameter returns information whether there were CRC errors in received control acknowledgement packets. Range 0 (NO) to 1 (YES).

**Table 16.23. Parameter properties of FG_CXP_CONTROL_ACK_PACKET_CRC_ERROR**

| Property | Value |
|---|---|
| Name | **FG_CXP_CONTROL_ACK_PACKET_CRC_ERROR** |
| Display Name | **CXP Control ACK Packet CRC Error** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.23. Usage of FG_CXP_CONTROL_ACK_PACKET_CRC_ERROR**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

## 16.22. LengthErrors

This category gives information about packet length mismatches for different types of packets.

### 16.22.1. FG_SYSTEMMONITOR_RX_LENGTH_ERROR_COUNT

This parmeter counts how often the length of a CXP packet doesn't correspond to what is specified in the header and returns the number of length errors. Range: between 0 and 8191 in steps of 1.

**Table 16.24. Parameter properties of FG_SYSTEMMONITOR_RX_LENGTH_ERROR_COUNT**

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_RX_LENGTH_ERROR_COUNT** |
| Display Name | **Systemmonitor Rx Length Error Count** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.24. Usage of FG_SYSTEMMONITOR_RX_LENGTH_ERROR_COUNT**

```
int result = 0;
```

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

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

### 16.22.2. FG_CXP_STREAMPACKET_LENGTH_ERROR

This parameter returns information whether a length error in the stream packets was detected. Range: 0 (NO) to 1 (YES).

**Table 16.25. Parameter properties of FG_CXP_STREAMPACKET_LENGTH_ERROR**

| Property | Value |
|---|---|
| Name | **FG_CXP_STREAMPACKET_LENGTH_ERROR** |
| Display Name | **CXP Stream Packet Length Error** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.25. Usage of FG_CXP_STREAMPACKET_LENGTH_ERROR**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

## 16.23. ReceivedPacketsCorrected

This category gives information about errors which occurred in received packets which have been corrected.

### 16.23.1. FG_CXP_ERROR_CORRECTED

This parameter counts errors received in packet headers and trailers that were corrected. Bits [11:0] count the amount of violations. Bit [12] is set when a counter overflow occurs. Range 0 to 8191 (13 bit).

**Table 16.26. Parameter properties of FG_CXP_ERROR_CORRECTED**

| Property | Value |
|---|---|
| Name | **FG_CXP_ERROR_CORRECTED** |
| Display Name | **CXP Error Corrected** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.26. Usage of FG_CXP_ERROR_CORRECTED**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

## 16.23.2. FG_CXP_ERROR_CORRECTED_TRIGGER

This parameter returns the information whether errors were corrected in received trigger packets. Range 0 (NO) to 1 (YES).

**Table 16.27. Parameter properties of FG_CXP_ERROR_CORRECTED_TRIGGER**

| Property | Value |
|---|---|
| Name | **FG_CXP_ERROR_CORRECTED_TRIGGER** |
| Display Name | **CXP Error Corrected Trigger** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.27. Usage of FG_CXP_ERROR_CORRECTED_TRIGGER**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

## 16.23.3. FG_CXP_ERROR_CORRECTED_TRIGGER_ACK

This parameter returns the information whether errors were corrected in received trigger acknowledge packets. Range 0 (NO) to 1 (YES).

**Table 16.28. Parameter properties of FG_CXP_ERROR_CORRECTED_TRIGGER_ACK**

| Property | Value |
|---|---|
| Name | **FG_CXP_ERROR_CORRECTED_TRIGGER_ACK** |
| Display Name | **CXP Error Corrected Trigger ACK** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.28. Usage of FG_CXP_ERROR_CORRECTED_TRIGGER_ACK**

```
int result = 0;
```

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

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

## 16.23.4. FG_CXP_ERROR_CORRECTED_STREAM

This parameter returns the information whether errors were corrected in received stream packets. Range 0 (NO) to 1 (YES).

**Table 16.29. Parameter properties of FG_CXP_ERROR_CORRECTED_STREAM**

| Property | Value |
|---|---|
| Name | **FG_CXP_ERROR_CORRECTED_STREAM** |
| Display Name | **CXP Error Corrected Stream** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.29. Usage of FG_CXP_ERROR_CORRECTED_STREAM**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

## 16.23.5. FG_CXP_ERROR_CORRECTED_CONTROL_ACK

This parameter returns the information whether errors were corrected in received stream acknowledge packets. Range 0 (NO) to 1 (YES).

**Table 16.30. Parameter properties of FG_CXP_ERROR_CORRECTED_CONTROL_ACK**

| Property | Value |
|---|---|
| Name | **FG_CXP_ERROR_CORRECTED_CONTROL_ACK** |
| Display Name | **CXP Error Corrected Control ACK** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.30. Usage of FG_CXP_ERROR_CORRECTED_CONTROL_ACK**

```
int result = 0;

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

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

## 16.23.6. FG_CXP_ERROR_CORRECTED_LINKTEST

This parameter returns the information whether errors were corrected in received link test packets. Range 0 (NO) to 1 (YES).

**Table 16.31. Parameter properties of FG_CXP_ERROR_CORRECTED_LINKTEST**

| Property | Value |
|---|---|
| Name | **FG_CXP_ERROR_CORRECTED_LINKTEST** |
| Display Name | **CXP Error Corrected Link Test** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.31. Usage of FG_CXP_ERROR_CORRECTED_LINKTEST**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

## 16.23.7. FG_CXP_ERROR_CORRECTED_HEARTBEAT

This parameter returns the information whether errors were corrected in received heartbeat packets. Range 0 (NO) to 1 (YES).

**Table 16.32. Parameter properties of FG_CXP_ERROR_CORRECTED_HEARTBEAT**

| Property | Value |
|---|---|
| Name | **FG_CXP_ERROR_CORRECTED_HEARTBEAT** |
| Display Name | **CXP Error Corrected Heartbeat** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.32. Usage of FG_CXP_ERROR_CORRECTED_HEARTBEAT**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

```
    }
}
```

### 16.23.8. FG_CORRECTED_ERROR_COUNT

The parameter counts the number of single-byte error corrections in CXP stream packets.

**Table 16.33. Parameter properties of FG_CORRECTED_ERROR_COUNT**

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

**Example 16.33. Usage of FG_CORRECTED_ERROR_COUNT**

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

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

## 16.24. ReceivedPacketsUncorrected

This category gives information about errors which occurred in received packets and which could not be corrected.

### 16.24.1. FG_CXP_ERROR_UNCORRECTED

This parameter counts errors received in packet headers and trailers that haven't been corrected. Bits [11:0] count the amount of violations. Bit [12] is set when a counter overflow occurs. Range 0 to 8191 (13 bit).

**Table 16.34. Parameter properties of FG_CXP_ERROR_UNCORRECTED**

| Property | Value |
|---|---|
| Name | **FG_CXP_ERROR_UNCORRECTED** |
| Display Name | **CXP Error Uncorrected** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.34. Usage of FG_CXP_ERROR_UNCORRECTED**

```
int result = 0;
```

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

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

## 16.24.2. FG_CXP_ERROR_UNCORRECTED_TRIGGER

This parameter returns the information whether there were errors in received trigger packets that haven't been corrected. Range 0 (NO) to 1 (YES).

**Table 16.35. Parameter properties of FG_CXP_ERROR_UNCORRECTED_TRIGGER**

| Property | Value |
|---|---|
| Name | **FG_CXP_ERROR_UNCORRECTED_TRIGGER** |
| Display Name | **CXP Error Uncorrected Trigger** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.35. Usage of FG_CXP_ERROR_UNCORRECTED_TRIGGER**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

## 16.24.3. FG_CXP_ERROR_UNCORRECTED_TRIGGER_ACK

This parameter returns the information whether there were errors in received trigger acknowledgement packets that haven't been corrected. Range 0 (NO) to 1 (YES).

**Table 16.36. Parameter properties of FG_CXP_ERROR_UNCORRECTED_TRIGGER_ACK**

| Property | Value |
|---|---|
| Name | **FG_CXP_ERROR_UNCORRECTED_TRIGGER_ACK** |
| Display Name | **CXP Error Uncorrected Trigger ACK** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.36. Usage of FG_CXP_ERROR_UNCORRECTED_TRIGGER_ACK**

```
int result = 0;

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

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

## 16.24.4. FG_CXP_ERROR_UNCORRECTED_STREAM

This parameter returns the information whether there were errors in received stream packets that haven't been corrected. Range 0 (NO) to 1 (YES).

**Table 16.37. Parameter properties of FG_CXP_ERROR_UNCORRECTED_STREAM**

| Property | Value |
|---|---|
| Name | **FG_CXP_ERROR_UNCORRECTED_STREAM** |
| Display Name | **CXP Error Uncorrected Stream** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.37. Usage of FG_CXP_ERROR_UNCORRECTED_STREAM**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

## 16.24.5. FG_CXP_ERROR_UNCORRECTED_CONTROL_ACK

This parameter returns information whether there were errors in received control acknowledgement packets that haven't been corrected. Range 0 (NO) to 1 (YES).

**Table 16.38. Parameter properties of FG_CXP_ERROR_UNCORRECTED_CONTROL_ACK**

| Property | Value |
|---|---|
| Name | **FG_CXP_ERROR_UNCORRECTED_CONTROL_ACK** |
| Display Name | **CXP Error Uncorrected Control ACK** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.38. Usage of FG_CXP_ERROR_UNCORRECTED_CONTROL_ACK**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

```
    }
}
```

## 16.24.6. FG_CXP_ERROR_UNCORRECTED_LINKTEST

This parameter returns information whether there were errors in received link test packets that haven't been corrected. Range 0 (NO) to 1 (YES).

**Table 16.39. Parameter properties of FG_CXP_ERROR_UNCORRECTED_LINKTEST**

| Property | Value |
|---|---|
| Name | **FG_CXP_ERROR_UNCORRECTED_LINKTEST** |
| Display Name | **CXP Error Uncorrected Link Test** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.39. Usage of FG_CXP_ERROR_UNCORRECTED_LINKTEST**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

## 16.24.7. FG_CXP_ERROR_UNCORRECTED_HEARTBEAT

This parameter returns information whether there were errors in received heartbeat packets that haven't been corrected. Range 0 (NO) to 1 (YES).

**Table 16.40. Parameter properties of FG_CXP_ERROR_UNCORRECTED_HEARTBEAT**

| Property | Value |
|---|---|
| Name | **FG_CXP_ERROR_UNCORRECTED_HEARTBEAT** |
| Display Name | **CXP Error Uncorrected Heartbeat** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.40. Usage of FG_CXP_ERROR_UNCORRECTED_HEARTBEAT**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

### 16.24.8. FG_UNCORRECTED_ERROR_COUNT

This parameter counts the number of uncorrected errors. Bit[2] indicates multiple byte errors in CXP stream packets.

**Table 16.41. Parameter properties of FG_UNCORRECTED_ERROR_COUNT**

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

**Example 16.41. Usage of FG_UNCORRECTED_ERROR_COUNT**

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

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

## 16.25. UnsupportedPackets

This category gives information about unsupported packets that have been received.

### 16.25.1. FG_SYSTEMMONITOR_RX_UNSUPPORTED_PACKET_COUNT

This parameter returns the number of received unsupported packets. Range: between 0 and 8191 in steps of 1.

**Table 16.42. Parameter properties of FG_SYSTEMMONITOR_RX_UNSUPPORTED_PACKET_COUNT**

| Property | Value |
|---|---|
| Name | **FG_SYSTEMMONITOR_RX_UNSUPPORTED_PACKET_COUNT** |
| Display Name | **Systemmonitor Rx Unsupported Packet Unit** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.42. Usage of FG_SYSTEMMONITOR_RX_UNSUPPORTED_PACKET_COUNT**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

```
}
```

## 16.25.2. FG_CXP_UNSUPPORTED_GPIO_RECEIVED

This parameter returns information whether a GPIO packet was received while using a CXP standard higher than 1.0. Range: 0 (NO) to 1 (YES).

**Table 16.43. Parameter properties of FG_CXP_UNSUPPORTED_GPIO_RECEIVED**

| Property | Value |
|---|---|
| Name | **FG_CXP_UNSUPPORTED_GPIO_RECEIVED** |
| Display Name | **CXP Unsupported GPIO Received** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.43. Usage of FG_CXP_UNSUPPORTED_GPIO_RECEIVED**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

## 16.25.3. FG_CXP_UNSUPPORTED_EVENT_RECEIVED

This parameter returns information whether an event packet was received while using a CXP standard less than 2.0. Range: 0 (NO) to 1 (YES).

**Table 16.44. Parameter properties of FG_CXP_UNSUPPORTED_EVENT_RECEIVED**

| Property | Value |
|---|---|
| Name | **FG_CXP_UNSUPPORTED_EVENT_RECEIVED** |
| Display Name | **CXP Unsupported Event Received** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.44. Usage of FG_CXP_UNSUPPORTED_EVENT_RECEIVED**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

## 16.25.4. FG_CXP_UNSUPPORTED_HEARTBEAT_RECEIVED

This parameter returns information whether a heartbeat packet was received while using a CXP standard less than 2.0. Range: 0 (NO) to 1 (YES).

**Table 16.45. Parameter properties of FG_CXP_UNSUPPORTED_HEARTBEAT_RECEIVED**

| Property | Value |
|---|---|
| Name | **FG_CXP_UNSUPPORTED_HEARTBEAT_RECEIVED** |
| Display Name | **CXP Unsupported Hearbeat Received** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.45. Usage of FG_CXP_UNSUPPORTED_HEARTBEAT_RECEIVED**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

## 16.25.5. FG_CXP_UNSUPPORTED_GPIO_ACK_RECEIVED

This parameter returns information whether a GPIO acknowledgment was received while using a CXP standard higher than 1.0. Range: 0 (NO) to 1 (YES).

**Table 16.46. Parameter properties of FG_CXP_UNSUPPORTED_GPIO_ACK_RECEIVED**

| Property | Value |
|---|---|
| Name | **FG_CXP_UNSUPPORTED_GPIO_ACK_RECEIVED** |
| Display Name | **CXP Unsupported GPIO ACK Received** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.46. Usage of FG_CXP_UNSUPPORTED_GPIO_ACK_RECEIVED**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

## 16.25.6. FG_CXP_UNSUPPORTED_GPIO_REQUEST_RECEIVED

This parameter returns information whether a GPIO request from VisualApplets was received while using a CXP standard higher than 1.0. Range: 0 (NO) to 1 (YES).

**Table 16.47. Parameter properties of FG_CXP_UNSUPPORTED_GPIO_REQUEST_RECEIVED**

| Property | Value |
|---|---|
| Name | **FG_CXP_UNSUPPORTED_GPIO_REQUEST_RECEIVED** |
| Display Name | **CXP Unsupported GPIO Request Received** |
| Type | **Unsigned Integer Field** |
| Field Size | **4** |
| Access policy | **Read-Only** |
| Storage policy | **Transient** |

**Example 16.47. Usage of FG_CXP_UNSUPPORTED_GPIO_REQUEST_RECEIVED**

```
int result = 0;

FieldParameterInt access;
const enum FgParamTypes type = FG_PARAM_TYPE_STRUCT_FIELDPARAMINT;

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

# Chapter 17. Revision History

Revision history of AcquisitionApplets releases.

| Applet Version | Release Date | Change Log | Delivered with |
|---|---|---|---|
| 1.1.4.0 | 06 Sep 2021 | Initial version of this applet. | % |
| 2.0.5.0 | 1 Mar 2022 | Added an extended overflow capability, which allows to monitor frame-IDs from the device. | Framegrabber SDK 5.10 |
| 2.1.6.0 | 24 Jan 2023 | No functional changes. | % |
| 2.2.7.0 | 5 Apr 2023 | • The applet now supports mirroring, **ImageSelector**, white balancing, and color converter.<br>• The applet now supports tap geometry sorting.<br>• The CXP-12 frame grabber now supports stream packet sizes of maximum 16 KB.<br>• Bugfixes. | Framegrabber SDK 5.11 |
| 2.3.7.0 | 22 Dec 2023 | • CoaXPress trigger packets updated to CoaXPress 2.1: The applet now supports sending the CXP **`LinkTrigger0`**, **`LinkTrigger1`**, **`LinkTrigger2`** and **`LinkTrigger3`**. For each link trigger, an individual source can be selected. By default, **`LinkTrigger0`** is the rising edge of the trigger source and **`LinkTrigger1`** is the falling edge of the trigger, which usually represents the exposure time. The default setting is equal to previous versions. Thus, the applet is fully compatible to CXP 1.1 and 2.0. **`LinkTrigger2`** and **`LinkTrigger3`** are not used by default.<br>The parameters *FG_CXP_TRIGGER_PACKET_MODE* (CxpTriggerPacketMode) and *FG_TRIGGERCAMERA_OUT_SELECT* (TriggerCameraOutSelect) became legacy. | Framegrabber SDK 5.11.2 |
| 2.4.7.0 | 31 Jul 2024 | • This applet has been extended with additional parameters and events for problem analysis, e.g. to find errors in the CXP stream. The new parameters and events have been added to the parameter tree.<br>• Bug fixes. See Section 17.1, 'Fixed Issues' for a detailed list of fixed issues. | Framegrabber SDK 5.11.3 |

## 17.1. Fixed Issues

### 17.1.1. Fixed in Version 2.4.7.0

- Solved communication issues with Opto-Coupled Trigger 5 board. (Ticket ID: 306774)
- Before fixing this issue, in some cases CXP status parameters showed wrong and misleading values. This has been fixed. (Ticket ID: 282288)

### 17.1.2. Fixed in Version 2.2.7.0

- The following CXP status parameters did not work correctly and have been fixed:
  - *FG_SYSTEMMONITOR_USED_CXP_CONNECTIONS*
  - *FG_PACKET_TAG_ERROR_COUNT*
  - *FG_CORRECTED_ERROR_COUNT*
  - *FG_UNCORRECTED_ERROR_COUNT*
  - *FG_SYSTEMMONITOR_PACKETBUFFER_OVERFLOW_COUNT*
  - *FG_SYSTEMMONITOR_PACKETBUFFER_OVERFLOW_SOURCE*
  - *FG_SYSTEMMONITOR_CXP_IMAGE_LINE_MODE*
  - **FG_TRIGGER_EVENT_COUNT**
  - **FG_TRIGGER_ACKNOWLEDGEMENT_COUNT**
  - **FG_TRIGGER_WAVE_VIOLATION**

### 17.1.3. Fixed in Version 2.0.5.0

Bitrate switching has been fixed. Prior to fixing this issue, switching the bitrate of a CXP link could put your frame grabber into a failure state. This has been fixed.

## 17.2. Known Issues

- Setting the tap geometry might fail. If you set tap geometry to **FG_GEOMETRY_1X_2YE**, you have to ensure that *FG_SENSORWIDTH* is greater or equal to *FG_WIDTH* and *FG_SENSORHEIGHT* is greater or equal to *FG_HEIGHT*. The order of setting the parameter is important. Otherwise, the tap geometry sorting will fail. (Ticket-ID: 270154)
- In rare cases, the log warning “Failed to read packet. Error code: -1997” can occur with boost v cameras. The camera communication and image acquisition are not affected. This warning can be ignored. (Ticket-ID: 262668)
- In rare cases, loading an applet can fail with the error message “Fg_init(...) : -2050 (Design is invalid)”. In this case, re-load the applet and contact the Basler Support [https://www.baslerweb.com/en/sales-support/support-contact/]. (Ticket-ID: 259458)

# Glossary

| | |
|---|---|
| Area of Interest (AOI) | See Region of Interest. |
| Board | A Basler hardware. Usually, a board is represented by a interface card. Boards might comprise multiple devices. |
| Board ID Number | An identification number of a Basler board in a PC system. The number is not fixed to a specific hardware but has to be unique in a PC system. |
| Camera Index | The index of a camera connected to a interface card. The first camera will have index zero. Mind the difference between the camera index and the interface card camera port.<br>See also Camera Port. |
| Camera Port | The Basler interface card connectors for cameras are called camera ports. They are numbered {0, 1, 2, ...} or enumerated {A, B, C, ... }. Depending on the interface one camera could be connected to multiple camera ports. Also, multiple cameras could be connected to one camera port. |
| Camera Tap | See Tap. |
| Device | A board can consist of multiple devices. Devices are numbered. The first device usually has number one. |
| Direct Memory Access (DMA) | A DMA transfer allows hardware subsystems within the computer to access the system memory independently of the central processing unit (CPU).<br><br>Basler uses DMAs for data transfer such as image data between a board e.g. a interface card and a PC. Data transfers can be established in multiple directions i.e. from a interface card to the PC (download) and from the PC to a interface card (upload). Multiple DMA channels may exist for one board. Control and configuration data usually do not use DMA channels. |
| DMA Channel | See DMA Index. |
| DMA Index | The index of a DMA transfer channel.<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 interface card internal functionality.<br><br>Basler uses hardware interrupts for the event transfer and processing is absolutely optimized for low latency. These interrupts are only produced by the interface card if an event is registered and activated by software. If an event is fired at a very high frequency this may influence the system performance.<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 interface card cuts the region of interest from the camera image. A region of interest might reduce or increase the required bandwidth and the corresponding image dimension. |
| Sensor Tap | See Tap. |
| Software Callback | See Event. |
| Tap | Some cameras have multiple taps. This means, they can acquire or transfer more than one pixel at a time which increses the camera's acquisition speed. The camera sensor tap readout order varies. Some cameras read the pixels interlaced using multiple taps, while some cameras read the pixel simultaneously from different locations on the sensor. The reconstruction of the frame is called sensor readout correction.<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


## A
Area of Interest, 23

## B
Bandwidth, 3
Boardstatus, 131

## C
Camera, 17
  Events, 17
  Format, 7
  Interface, 4, 17
Camera Simulator, 100, 100
Camera::Events, 17
CoaXPress, 7
CoaXPress::Test, 13
Color Converter, 92
CXP Source Tag, 4

## E
Errors, 138
Errors::CRC, 148
Errors::LengthErrors, 149
Errors::ReceivedPacketsCorrected, 150
Errors::ReceivedPacketsUncorrected, 154
Errors::UnsupportedPackets, 158
Events
  Camera, 17
  Digital Inputs, 31, 55
  Overflow, 87
  Trigger, 31
  Trigger Lost Detection, 76
  Trigger Output, 80
  Trigger Queue, 61

## F
Features, 1
FG_APPLET_BUILD_TIME, 112
FG_APPLET_ID, 111
FG_APPLET_REVISION, 115
FG_APPLET_VERSION, 115
FG_AREATRIGGERMODE, 46
FG_BITALIGNMENT, 97
FG_CAMERASIMULATOR_ACTIVE, 109
FG_CAMERASIMULATOR_ENABLE, 100
FG_CAMERASIMULATOR_FRAMERATE, 107
FG_CAMERASIMULATOR_FRAME_GAP, 103
FG_CAMERASIMULATOR_HEIGHT, 102
FG_CAMERASIMULATOR_LINERATE, 107
FG_CAMERASIMULATOR_LINE_GAP, 102
FG_CAMERASIMULATOR_PASSIVE, 109
FG_CAMERASIMULATOR_PATTERN, 104
FG_CAMERASIMULATOR_PATTERN_OFFSET, 104
FG_CAMERASIMULATOR_PIXEL_FREQUENCY, 106
FG_CAMERASIMULATOR_ROLL, 105

FG_CAMERASIMULATOR_SELECT_MODE, 106
FG_CAMERASIMULATOR_TRIGGER_MODE, 108
FG_CAMERASIMULATOR_WIDTH, 101
FG_CAMERA_STREAM_STATUS0, 17
FG_CAMSTATUS, 112
FG_CAMSTATUS_EXTENDED, 113
FG_CORRECTED_ERROR_COUNT, 154
FG_CUSTOM_BIT_SHIFT_RIGHT, 98
FG_CXP_CAMERA_FRAME_CORRUPT_COUNT, 147
FG_CXP_CAMERA_FRAME_LOST_COUNT, 147
FG_CXP_CAMERA_MARKER_ERROR_COUNT, 146
FG_CXP_CAMERA_UNEXPECTED_STARTUP_DATA, 146
FG_CXP_CLEAR_TEST_STATISTIC_PORT, 15
FG_CXP_CONTROL_ACK_INCOMPLETE_COUNT, 142
FG_CXP_CONTROL_ACK_LOST_COUNT, 141
FG_CXP_CONTROL_ACK_PACKET_CRC_ERROR, 149
FG_CXP_CONTROL_TAG_ERROR_COUNT, 141
FG_CXP_CORRUPTED_WORD_COUNT, 14
FG_CXP_ERROR_CORRECTED, 150
FG_CXP_ERROR_CORRECTED_CONTROL_ACK, 152
FG_CXP_ERROR_CORRECTED_HEARTBEAT, 153
FG_CXP_ERROR_CORRECTED_LINKTEST, 153
FG_CXP_ERROR_CORRECTED_STREAM, 152
FG_CXP_ERROR_CORRECTED_TRIGGER, 151
FG_CXP_ERROR_CORRECTED_TRIGGER_ACK, 151
FG_CXP_ERROR_UNCORRECTED, 154
FG_CXP_ERROR_UNCORRECTED_CONTROL_ACK, 156
FG_CXP_ERROR_UNCORRECTED_HEARTBEAT, 157
FG_CXP_ERROR_UNCORRECTED_LINKTEST, 157
FG_CXP_ERROR_UNCORRECTED_STREAM, 156
FG_CXP_ERROR_UNCORRECTED_TRIGGER, 155
FG_CXP_ERROR_UNCORRECTED_TRIGGER_ACK, 155
FG_CXP_HEARTBEAT_INCOMPLETE_COUNT, 142
FG_CXP_HEARTBEAT_MAX_PERIOD_VIOLATION_COUNT, 143
FG_CXP_IMAGETAG_ERROR_COUNT, 145
FG_CXP_INPUT_MAPPED_FW_PORT_PORT, 137
FG_CXP_OVERTRIGGER_REQUEST_PULSECOUNT, 140
FG_CXP_PACKET_LENGTH_ERROR_COUNT, 15
FG_CXP_RECEIVED_PACKET_COUNT, 14
FG_CXP_STREAMID_ERROR_COUNT, 145
FG_CXP_STREAMPACKET_CRC_ERROR, 148
FG_CXP_STREAMPACKET_LENGTH_ERROR, 150
FG_CXP_STREAM_PACKET_COUNT, 10
FG_CXP_TRANSMITTED_PACKET_COUNT, 14
FG_CXP_TRIGGER_ACK_MISSING_COUNT, 140
FG_CXP_TRIGGER_PACKET_MODE, 116
FG_CXP_UNSUPPORTED_EVENT_RECEIVED, 159
FG_CXP_UNSUPPORTED_GPIO_ACK_RECEIVED, 160
FG_CXP_UNSUPPORTED_GPIO_RECEIVED, 159
FG_CXP_UNSUPPORTED_GPIO_REQUEST_RECEIVED, 160
FG_CXP_UNSUPPORTED_HEARTBEAT_RECEIVED, 159
FG_DEBUGFILE, 123
FG_DEBUGINENABLE, 123
FG_DEBUGINSERT, 123
FG_DEBUGOUTENABLE, 127
FG_DEBUGOUTPIXEL, 128
FG_DEBUGOUTXPOS, 127
FG_DEBUGOUTYPOS, 128

FG_DEBUGREADY, 125
FG_DEBUGSAVECONFIG, 119
FG_DEBUGSOURCE, 119
FG_DEBUGSOURCENAME, 119
FG_DEBUGWRITEFLAG, 125
FG_DEBUGWRITEPIXEL, 124
FG_DEBUG_FORCE_FRAMEID, 125
FG_DEBUG_FRAMEID, 126
FG_DEBUG_FRAMEID_TO_FIRSTPIXEL, 122
FG_DEBUG_PWM_SLOWRATE, 121
FG_DEBUG_SLOWMODE, 120
FG_DEBUG_SOFTWRAE_SLOWGATE, 120
FG_DEBUG_VERSION, 122
FG_DMASTATUS, 132
FG_END_OF_FRAME_CAM_PORT_0, 19
FG_FILLLEVEL, 82
FG_FORMAT, 93
FG_FRONT_GPI, 50
FG_GENTL_INFO_IGNOREFGFORMAT, 129
FG_GENTL_INFO_OVERFLOWCAPABLE, 129
FG_GENTL_INFO_VERSION, 129
FG_HAP_FILE, 112
FG_HEIGHT, 25
FG_IMAGE_TAG, 5
FG_IMG_SELECT, 89
FG_IMG_SELECT_PERIOD, 88
FG_MISSING_CAM0_FRAME_RESPONSE, 80
FG_MISSING_CAMERA_FRAME_RESPONSE, 78
FG_MISSING_CAMERA_FRAME_RESPONSE_CLEAR, 79
FG_OVERFLOW, 83
FG_OVERFLOW_CAM0, 87
FG_OVERFLOW_EVENT_SELECT, 85
FG_OVERFLOW_OFF_THRESHOLD, 83
FG_OVERFLOW_ON_SYNC_THRESHOLD, 85
FG_OVERFLOW_ON_THRESHOLD, 84
FG_PACKET_TAG_ERROR_COUNT, 143
FG_PIXELDEPTH, 97
FG_PIXELFORMAT, 10
FG_SCALINGFACTOR_BLUE, 91
FG_SCALINGFACTOR_GREEN, 90
FG_SCALINGFACTOR_RED, 90
FG_SENDSOFTWARETRIGGER, 53
FG_SENSORHEIGHT, 21
FG_SENSORWIDTH, 20
FG_SOFTWARETRIGGER_IS_BUSY, 54
FG_SOFTWARETRIGGER_QUEUE_FILLLEVEL, 54
FG_START_OF_FRAME_CAM_PORT_0, 19
FG_SYSTEMMONITOR_BYTE_ALIGNMENT_8B_10B_LOCKED, 138
FG_SYSTEMMONITOR_CHANNEL_CURRENT, 131
FG_SYSTEMMONITOR_CHANNEL_VOLTAGE, 131
FG_SYSTEMMONITOR_CURRENT_LINK_SPEED, 135
FG_SYSTEMMONITOR_CURRENT_LINK_WIDTH, 135
FG_SYSTEMMONITOR_CXP_IMAGE_LINE_MODE, 12
FG_SYSTEMMONITOR_CXP_STANDARD, 9
FG_SYSTEMMONITOR_DECODER_8B_10B_ERROR, 138
FG_SYSTEMMONITOR_EXTERNAL_POWER, 137
FG_SYSTEMMONITOR_FPGA_DNA_HIGH, 114
FG_SYSTEMMONITOR_FPGA_DNA_LOW, 114

FG_SYSTEMMONITOR_FPGA_TEMPERATURE, 133
FG_SYSTEMMONITOR_FPGA_VCC_AUX, 134
FG_SYSTEMMONITOR_FPGA_VCC_BRAM, 134
FG_SYSTEMMONITOR_FPGA_VCC_INT, 133
FG_SYSTEMMONITOR_MAPPED_TO_FG_PORT, 132
FG_SYSTEMMONITOR_PACKETBUFFER_OVERFLOW_COUNT, 144
FG_SYSTEMMONITOR_PACKETBUFFER_OVERFLOW_SOURCE, 144
FG_SYSTEMMONITOR_PCIE_TRAINED_PAYLOAD_SIZE, 136
FG_SYSTEMMONITOR_PCIE_TRAINED_REQUEST_SIZE, 136
FG_SYSTEMMONITOR_PORT_BIT_RATE, 8
FG_SYSTEMMONITOR_POWER_OVER_CXP_CONTROLLER_ENABLED, 7
FG_SYSTEMMONITOR_POWER_OVER_CXP_STATE, 7
FG_SYSTEMMONITOR_RX_LENGTH_ERROR_COUNT, 149
FG_SYSTEMMONITOR_RX_PACKET_CRC_ERROR_COUNT, 148
FG_SYSTEMMONITOR_RX_STREAM_INCOMPLETE_COUNT, 139
FG_SYSTEMMONITOR_RX_UNKNOWN_DATA_RECEIVED_COUNT, 139
FG_SYSTEMMONITOR_RX_UNSUPPORTED_PACKET_COUNT, 158
FG_SYSTEMMONITOR_STREAM_PACKET_SIZE, 8
FG_SYSTEMMONITOR_USED_CXP_CONNECTIONS, 11
FG_TAPGEOMETRY, 12
FG_TIMEOUT, 111
FG_TRIGGERCAMERA_OUT_SELECT, 117
FG_TRIGGERCAMERA_SOURCE_CXP0, 68
FG_TRIGGERCAMERA_SOURCE_CXP1, 70
FG_TRIGGERCAMERA_SOURCE_CXP2, 71
FG_TRIGGERCAMERA_SOURCE_CXP3, 72
FG_TRIGGERIN_DEBOUNCE, 49
FG_TRIGGERIN_DOWNSCALE, 52
FG_TRIGGERIN_DOWNSCALE_PHASE, 52
FG_TRIGGERIN_POLARITY, 51
FG_TRIGGERIN_SRC, 51
FG_TRIGGERIN_STATS_FREQUENCY, 57
FG_TRIGGERIN_STATS_MAXFREQUENCY, 58
FG_TRIGGERIN_STATS_MINFREQUENCY, 58
FG_TRIGGERIN_STATS_MINMAXFREQUENCY_CLEAR, 59
FG_TRIGGERIN_STATS_POLARITY, 56
FG_TRIGGERIN_STATS_PULSECOUNT, 56
FG_TRIGGERIN_STATS_PULSECOUNT_CLEAR, 57
FG_TRIGGERIN_STATS_SOURCE, 55
FG_TRIGGEROUT_SELECT_FRONT_GPO_0, 73
FG_TRIGGEROUT_SELECT_FRONT_GPO_1, 75
FG_TRIGGEROUT_STATS_PULSECOUNT, 77
FG_TRIGGEROUT_STATS_PULSECOUNT_CLEAR, 78
FG_TRIGGEROUT_STATS_SOURCE, 77
FG_TRIGGERQUEUE_FILLLEVEL, 62
FG_TRIGGERQUEUE_MODE, 61
FG_TRIGGERSTATE, 47
FG_TRIGGER_EXCEEDED_PERIOD_LIMITS, 76
FG_TRIGGER_EXCEEDED_PERIOD_LIMITS_CAM0, 80
FG_TRIGGER_EXCEEDED_PERIOD_LIMITS_CLEAR, 76
FG_TRIGGER_FRAMESPERSECOND, 35, 48
FG_TRIGGER_FRONT_GPI0_FALLING, 60
FG_TRIGGER_FRONT_GPI0_RISING, 59
FG_TRIGGER_MULTIPLY_PULSES, 38, 60
FG_TRIGGER_OUTPUT_CAM0, 81
FG_TRIGGER_OUTPUT_EVENT_SELECT, 80
FG_TRIGGER_PULSEFORMGEN0_DELAY, 66
FG_TRIGGER_PULSEFORMGEN0_DOWNSCALE, 64

FG_TRIGGER_PULSEFORMGEN0_DOWNSCALE_PHASE, 65
FG_TRIGGER_PULSEFORMGEN0_WIDTH, 67
FG_TRIGGER_PULSEFORMGEN1_DELAY, 66
FG_TRIGGER_PULSEFORMGEN1_DOWNSCALE, 64
FG_TRIGGER_PULSEFORMGEN1_DOWNSCALE_PHASE, 65
FG_TRIGGER_PULSEFORMGEN1_WIDTH, 67
FG_TRIGGER_PULSEFORMGEN2_DELAY, 66
FG_TRIGGER_PULSEFORMGEN2_DOWNSCALE, 64
FG_TRIGGER_PULSEFORMGEN2_DOWNSCALE_PHASE, 65
FG_TRIGGER_PULSEFORMGEN2_WIDTH, 67
FG_TRIGGER_PULSEFORMGEN3_DELAY, 66
FG_TRIGGER_PULSEFORMGEN3_DOWNSCALE, 64
FG_TRIGGER_PULSEFORMGEN3_DOWNSCALE_PHASE, 65
FG_TRIGGER_PULSEFORMGEN3_WIDTH, 67
FG_TRIGGER_QUEUE_FILLLEVEL_EVENT_OFF_THRESHOLD, 63
FG_TRIGGER_QUEUE_FILLLEVEL_EVENT_ON_THRESHOLD, 62
FG_TRIGGER_QUEUE_FILLLEVEL_THRESHOLD_CAM0_OFF, 63
FG_TRIGGER_QUEUE_FILLLEVEL_THRESHOLD_CAM0_ON, 63
FG_UNCORRECTED_ERROR_COUNT, 158
FG_VANTAGEPOINT, 20
FG_VISUALAPPLETS_BUILD_VERSION, 116
FG_WIDTH, 24
FG_XOFFSET, 25
FG_YOFFSET, 26
Format, 93
Frame ID, 4

## G

Generator, 100
GPI, 50

## I

Image Select, 88
Image Selector, 88
Image Tag, 5
Image Transfer, 5

## M

Miscellaneous, 111
Miscellaneous::Debug, 119
Miscellaneous::Debug::Input, 122
Miscellaneous::Debug::Output, 127
Miscellaneous::GenTL, 129
Miscellaneous::Legacy, 116
Miscellaneous::Version, 115

## O

Output Format, 93
Overflow, 82, 82
  Events, 87
Overflow::Events, 86

## P

PC Interface, 4
Pixel Format, 7

## R

Region of Interest, 23

ROI, 23

## S

Sensor Geometry, 20, 20
Software Interface, 6
Source Tag, 4
Specifications, 1

## T

Trigger, 27, 27
- Activate, 47
- Busy, 54
- Bypass, 44
- Camera Signal Mapping, 68
- Debounce, 49
- Debugging, 45
- Digital Input, 31, 50
- Digital Input Output Mapping, 31
- Digital Output, 31, 34, 73
- Downscale Input, 52
- Encoder, 34
- Error Detection, 45
- Events, 31
- Exceeded Period Limits, 76
- Exsync, 34
- External, 34, 46, 49
- Flash, 34, 36
- Frame Rate, 32
- Framerate, 48
- Generator, 32, 46
- GPI, 50
- Grabber Controlled, 32
- Image Trigger, 34
- Input, 49, 50
- Input Statistics, 55
- IO Triggered, 34
- Length, 33
- Lost Trigger, 45, 76
- Missing Frame Response, 78
- Mode, 46
- Multi Camera, 44
- Multiply Pulses, 60
- Output, 73
- Output Event, 80
- Output Statistics, 76
- Period, 48
- Pin Allocation, 31
- Polarity Input, 51, 56
- Pulse Form Generator, 63
- Pulse Multiplication, 36
- Queue, 41, 43, 61
- Sequencer, 36, 60
- Signal Length, 33
- Signal Width, 33
- Software Controlled, 53
- Software Trigger, 39, 46, 53
- Start, 47
- Stop, 47

Synchronized, 44, 46
Synchronized Cameras, 44
System Analysis, 45
Trigger IO, 31
Width, 33
Trigger::Camera Out Signal Mapping, 68
Trigger::Digital Output, 73
Trigger::Digital Output::Statistics, 76
Trigger::Output Event, 80
Trigger::Pulse Form Generator 0, 63
Trigger::Pulse Form Generator 1, 67
Trigger::Pulse Form Generator 2, 67
Trigger::Pulse Form Generator 3, 68
Trigger::Queue, 61
Trigger::Sequencer, 60
Trigger::Trigger Input, 49
Trigger::Trigger Input::External, 49
Trigger::Trigger Input::Software Trigger, 53
Trigger::Trigger Input::Statistics, 55

## W

White Balance, 90, 90